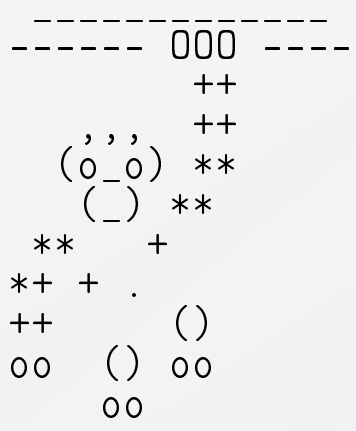
TestApe

Unit Testing for embedded software



Using the TestApe unit testing tool

Release 1171

Contents

1 TestApe **5**

1.1 Theory of usage 5
1.2 The author 6
1.3 Terminology 6
1.4 Typography 7
1.5 Terms of use 8

2 Using the TestApe library **9**

2.1 Creating and executing a basic testcase 9
2.2 testmain 10
2.3 Generating the test executable 10
2.4 Validating output 12
2.5 Validating function calls 13
2.6 Function return values 14
2.7 Function parameters 15
2.8 Return values from mock functions 15
2.9 Input to mock functions 16
2.10 Mocking function calls 17
2.11 Mocking c library 18
2.12 Calling mocked functions from the mock 18
2.13 Organizing tests 19
2.14 Ranged tests with parameters 20
2.15 Validating ranged tests with parameters 21
2.16 Parameter sets 23
2.17 Signal handling 23
2.18 Arguments to tests 25

2.19 Floating point support . . . 26
2.20 Floating point return values . . . 26
2.21 Mocking functions returning floating point values . . . 28
2.22 Using the mocking system with other test systems . . . 28
2.23 Adding and removing object files from unit . . . 29
2.24 Handling of function calls explained in detail . . . 29

3 Using the TestApe instrumenter 31
3.1 The TestApe instrumenter commandline options . . . 31
3.2 Installing TestApe on Linux . . . 32
3.3 Using TestApe with gcc . . . 33
3.4 Using TestApe with gcc arm cross compiler . . . 33
3.5 Installing TestApe on windows . . . 34
3.6 Using TestApe with Visual Studio . . . 34
3.7 Using TestApe with mingw-w64 gcc . . . 35
3.8 Troubleshooting . . . 35

4 Using the TestApe test executable 37
4.1 Commandline options . . . 37
4.2 Exit codes . . . 38
4.3 Memory analysis . . . 38
4.4 Coverage analysis . . . 38
4.5 Debugging test executable . . . 39
4.6 Log output and error messages . . . 39

5 Customizing TestApe 41
5.1 Error messages . . . 41
5.2 Startup code . . . 41
5.3 External dependencies . . . 42
5.4 Other operating systems and bare metal configurations . . . 42

6 Reference 44
6.1 ALLOW . . . 44
6.2 ALLOW_MOCK . . . 44
6.3 COMMENT . . . 45

6.4 EXECUTE 45
6.5 EXPECT 46
6.6 EXPECT_MOCK 47
6.7 MOCK 47
6.8 MOCK_VALUE 48
6.9 MOCK_VALUE_FP 48
6.10 PARAMETER 48
6.11 PARAMETER_RANGE 49
6.12 PARAMETER_RANGE_FP 49
6.13 PARAMETER_SET 50
6.14 PARAMETER_VALUE 50
6.15 PARAMETER_VALUE_FP 51
6.16 VALIDATE 51
6.17 VALIDATE_BIT 52
6.18 VALIDATE_BYTE 52
6.19 VALIDATE_DWORD 52
6.20 VALIDATE_FP 53
6.21 VALIDATE_FILE 53
6.22 VALIDATE_MEMORY 53
6.23 VALIDATE_STRUCT 54
6.24 VALIDATE_STRING 54
6.25 VALIDATE_WORD 55
6.26 Deprecated macros 55

Chapter 1

TestApe

TestApe is a free unit-testing package that can be used to test embedded software written in C. Like many other unit test frameworks, it can be used to make a function call into the unit and test the return value. However, TestApe's ability to test what goes on inside these functions is what make this framework different.

In general, the output of a function becomes a less interesting test parameter as the abstraction level of the function increases. For example, the output of a function implementing a state machine is of little importance for test, compared to the behavior inside the function. TestApe allows testing of the behavior inside functions as well as their output.

TestApe package comes with a instrumenter for code generated with GCC (minGw/CygWin) or Microsoft compilers. It will run in Windows, Linux and other operating systems on intel and arm processors. The instrumenter generates default mocks in order to test and simulate the data flow between the unit that is tested and the mock function.

1.1 Theory of usage

In classic software development, a modest complicated software program is typically decomposed into several cooperating modules. The compiler interprets the source code for each module and creates an object file holding the compiled code. The linker assembles all the object files and creates the complete executable. If one or more, of these object files were not present, the linker could not assemble the software program and it would not be able to execute. In TestApe terminology, an incomplete assembly of object files is called a unit. The TestApe instrumenter combines these units with the framework, the TestApe tests and with mocks for the functions in missing object files, in order to turn it all into a TestApe executable.

When running a TestApe executable the behavior of the unit is tested using the TestApe framework and the result of the tests are reported as shown below

```
In stub execute

PASSED verify command
  expected .......... [000] 6c 69 6e 6b 20 40 74 65 'link @te'
```

```
  actual ............ [000] 6c 69 6e 6b 20 40 74 65 'link @te'

PASSED verify file existence testape_prelink_args
  expected .......... testape_prelink_args
  actual ............ testape_prelink_args

PASSED verify file size testape_prelink_args
  expected .......... 52
  actual ............ 52

PASSED verify testape_prelink_args
  expected .......... [000] 6d 79 6f 62 6a 2e 6f 62 'myobj.ob'
  actual ............ [000] 6d 79 6f 62 6a 2e 6f 62 'myobj.ob'

PASSED verify ret_val
  expected .......... 0 (0)
  actual ............ 0

PASSED test instrument_testape_lib
```

The entire process can be automated and run repeatedly. A TestApe executable is typically run, whenever one of the modules in the unit has changed. The test verifies that the new functionality behaves as expected ,or that the unchanged part of the unit is still working.

1.2 The author

Martin Steen Nielsen holds degrees in Electronic Engineering and Computer Science. He has worked with unit testing of embedded software since 1999. Martin became hooked on automated module testing, seeing how it made significant improvements in quality and stability of his own work.

To avoid the tedious work of writing stub functions, the author invented the TestApe instrumenter, so that new unit test projects could be launched quickly. The principles in the TestApe framework, as well as the freedom in structure it supports, is inspired by experience with unit testing and test driven development during these years.

1.3 Terminology

- Unit

  An incomplete collection of modules, e.g. source files, object files and/or library files. When combined these modules form the unit that is tested. The instrumenter will fill in the missing pieces in order to form a complete test executable.

- Test executable

  The output of the instrumenter. The test executable will run the tests and report errors found.

- Test

  The test is the basic test implementation by the test designer. It is composed of one single C function, that will call the selected functions in the unit and validate the behavior.

- Testcase

  It is common to group several tests to form a more complete test - for example, software that implement some kind of state machine may require several tests to be executed, before a scenario is covered.

  It is a possibility for the test designer to form testcases by nesting tests.

- Testsuite

  It is a possibility for the test designer to form testsuites by nesting testcases.

- Stub

  A stub is a function that replaces non-existing functionality in the unit by providing an alternative test implementation for the function it replaces.

  The test developer can choose to implement a stub manually. Manually created stubs are not dynamic - they remain in scope for all tests at all times.

  If a stub is not created manually, the framework will automatically create a default TestApe mock.

- Mock

  A mock is a function the replaces or extends existing functionality by providing an implementation for the function it mocks.

  The mock implementation is defined by the test developer. The mock will typically verify parameters and simulate a return value for the unit. In addition the mock function can choose to call the function it is mocking.

  Default mocks are automatically generated for those functions that are not part of the unit. TestApe mocks dynamic scope. Each test define which functions that are mocked and which mock implementation they use for each test.

- Instrumenter

  A tool provided in the package, that will create default mocks for those functions, that are not present in the unit.

- Framework

  The TestApe framework is a library wrapped in a collection of macros. The macros are used when writing test.

1.4 Typography

The usage of the package is illustrated by text and examples. All sample code that is written with test purpose appears with linenumbers framed and in `italic`.

```
1 void test_addition(void) {
2   int result;
3   result = calculate(3,7,'+');
4   }
```

Everything that can be typed literally in testcode like macros, parameters and other code references appears in fixed type font e.g. `test_addition()`

The framework will generate log output. Samples of log output are shown framed without linenumbers.

```
FAILED verify result
  expected ..... 10 (10)
  actual ..... -1
```

Commands that are to be given on a commandline is shown unframed in `italic`. Like it is shown below.

testape gcc unit_a.c unit_d.c ad_test.c testape.a

1.5 Terms of use

It is the hope of the author, that you will find this package useful. Please observe the following terms when using the package :

> By downloading and/or using one of Martin Steen Nielsen's TestApe components you are legally bound by the following: The TestApe components may be used for personal or business but you may not put them on a diskette, CD, web site or any other medium and offer them for redistribution or resale. The TestApe components are offered, "as is" without warranty of any kind, either expressed or implied. Martin Steen Nielsen will not be liable for any damage or loss of data whatsoever due to downloading or use of these components. In no event shall Martin Steen Nielsen be liable for any damages including, but not limited to, direct, indirect, special, incidental or consequential damages or other losses arising out of the use of or inability to use the components and/or information from testape.com. Martin Steen Nielsen reserves the right to change and/or modify these terms with no prior notice. Understand this is a legally binding contract, and violation will have consequences where this document may be used against you.

Chapter 2

Using the TestApe library

TestApe test files are plain C modules. These are compiled and linked with the unit that is tested, in order to generate the test executable. The unit that is tested can be as simple as one source file, but it may also consist of any combination of c, object or library files that you find is an appropriate combination for what you want to test. The tests uses the macros available from the testape framework. These are defined in `testape.h`. The macros fall into two categories - macros that are used for test control and macros that are used for validations.

2.1 Creating and executing a basic testcase

A test project consists of a range of basic tests. These tests can be nested, run in loops, put in structures in whatever way is appropriate for the test organization. There is no hard coded arrays and/or global variables that limits how these can be organized. A TestApe test is made of one plain C function that will setup expectations, call one or more functions in the unit, and validate the result.

In the sample below the test `test_addition` will call the function `calculate` in the unit.

```
1 void test_addition(void) {
2   int result;
3   result = calculate(3,7,'+');
4   }
```

The sample is executed using `EXECUTE(test_addition)` and this would produce the following output in the log

```
Executing test test_addition
PASSED test test_addition
```

Even though there is no validation in this test, it will enable coverage analysis, debugging and memory checking of the code in `calculate`.

2.2 testmain

All test projects must define one test - the `testmain` test. This test will serve as an entry point for the framework. The framework will launch `testmain` after initialization and analysis of the commandline arguments.

In order to launch `test_addition` the following code is needed

```
1  void test_addition(void) {
2    int result;
3    result = calculate(3,7,'+');
4  }
5
6  void testmain(void) {
7    EXECUTE(test_addition);
8  }
```

See also Section 4.1 about exitcodes and command line parameters.

2.3 Generating the test executable

To generate the test executable, the following is needed : the code being tested, the test, and the testape framework. These are combined to form the test executable using the instrumenter together with the compiler or linker.

`calc.c` shown below will be used in the following examples. The calculator will perform a simple addition or subtraction. It will do that calling `invalid()`, `add()`, `subtract()`,`divide()` and `multiply()`. The calculator program is normally assembled of `calc.c`, `add.c`, `subtract.c`, `multiply.c` and `divide.c`, but `calc.c` and `add.c` are the only files in the unit being tested in the following samples.

```
1  int validate(int a, int b)  { return 1; }
2
3  int calculate(int operand1, int operand2, char operation) {
4    if ( invalid(operand1) || invalid(operand2) )
5      return ERROR;
6
7    switch(operation) {
8      case '+':
9        return add(operand1, operand2);
10     case '-':
11       return subtract(operand1, operand2);
12     case '*':
13       return multiply(a,b);
14     case '/':
15       return divide(a,b);
16     }
17   return ERROR;
18 }
```

Sample1 is shown here, [1]

```
#include "testape.h"

void test_addition(void) {
  int result;
  result = calculate(3,7,'+');
}

void testmain(void) {
  EXECUTE(test_addition);
}
```

can be compiled and run in an MinGw-w64/gcc environment like it is shown below. Notice that `multiply.c`, `divide.c`, and `subtract.c` are not compiled. Instead the instrumenter is invoked in front of the compiler. The instrumenter will launch the compiler and generate default mocks for the missing functions in the unit.

```
c:\>testape gcc calc.c add.c sample1.c testape.a

TestApe instrumentation tool, release 1171-xx-xxxx-Linux.
Unit testing for embedded software - http://testape.com


      -------------
     ------ OOO ----
            ++
         ,,, ++
        (o_o)**
         (_)**
        **   +
       *+ + .
       ++     ()
       oo  () oo
           oo

 Analyzing ...
   Generating default mocks -> subtract multiply divide
   Generating hooks ->  None

c:\>sample1 -ol
Executing test testmain

  Executing test test_addition

  PASSED test test_addition

PASSED test testmain
```

TestApe will also run with Visual Studio CL.EXE or with GCC in Linux. See more details about supported environments in section 3.

[1] The code examples shown, are given in order to illustrate the usage - they may be incomplete. The installation contains a sample directory that contains full versions of the samples from this document. To run this example on windows, select "Try examples" from menu and run 'nmake sample1'. In Linux run 'make sample1' in '/usr/share/doc/testape/samples'.

2.4 Validating output

To improve the test, one or more validations can be added, as shown in sample2 below.[1] The `VALIDATE` macro that is used, will validate the data returned from `calculate` against the expected value `10`.

```
1 void test_addition(void) {
2   int result;
3   result = calculate(3,7,'+');
4   VALIDATE(result, 10);
5 }
```

If the validation succeeds the following output are generated

```
testape: TestApe test executable
testape: Unit testing for embedded software - http://testape.com
testape:
testape: Initializing ./sample2
testape:   -ol ( output errors, summary and log )
testape:
testape: Executing test testmain
testape:
testape:   Executing test test_addition
testape:
testape:     PASSED verify result
testape:       expected .......... 10 (10)
testape:       actual ............ 10
testape:
testape:   PASSED test test_addition
testape:
testape: PASSED test testmain
testape:
testape: Terminating ./sample2 - exitcode 0
testape:   2 passed, 0 failed (0 errors), 0 skipped
```

If not, the failure is is reported in the log. In addition the test will be listed as failed in the summary at the end of the log.

```
testape: Unit testing for embedded software - http://testape.com
testape:
testape: Initializing ./sample2
testape:   -ol ( output errors, summary and log )
testape:
testape: Executing test testmain
testape:
testape:   Executing test test_addition
testape:
testape:     FAILED verify result
error e001: sample2.c:6: failed value of result. Expected 10 (10), was 9.
testape:       expected .......... 10 (10)
testape:       actual ............ 9
testape:
```

```
testape:   FAILED test test_addition (1 error)
testape:     Verify integer in test test_addition
testape:
testape: FAILED test testmain (1 error)
testape:   Verify integer in test test_addition
testape:
testape: Terminating ./sample2 - exitcode 1
testape:   0 passed, 2 failed (1 errors), 0 skipped
```

The framework includes additional macros for validating numbers, bits, bytes, words, dwords, strings, arrays, structs and file contents. Detailed description can be found in chapter 6 Readability of the log file can be improved by choosing proper test names and by using macros for values as shown below in sample3[1] If possible, the framework will try to retain the symbolic values and use these in the log output.

```
1 #define EXPECTED_SUM 10
2 void test_addition(void) {
3   VALIDATE(calculate(3,7,'+'), EXPECTED_SUM);
4 }
```

will change the log output to[2]

```
Executing test testmain

  Executing test test_addition

  PASSED verify calculate(3,7,'+')
    expected .......... EXPECTED_SUM (10)
    actual ............ 10

  PASSED test test_addition

PASSED test testmain
```

2.5 Validating function calls

The implementation of `calculate` depends upon the functions `validate`, `add`, `divide`, `multiply` and `subtract` in order to implement the calculations. For a given test, there is a certain order in which these are executed. This is tested using the macro `EXPECT` as it is shown in sample4 below.[1]

```
1 void test_addition(void) {
2   EXPECT( invalid );
3   EXPECT( invalid );
4   EXPECT( add );
5   calculate(3,7,'+');
6 }
```

[2]The log examples shown are given in order to illustrate a point. As default only errors and a summary are shown in the command prompt. A log output similar to what is shown in this document can be enabled with the options -ol and -notag.

With the `EXPECT` macro the test developer can indicate to the framework, that this test expects `invalid()` and `add()` function calls during `calculate`. The framework detects the function calls made by `calculate` and generates the output shown below[2]

```
Executing test test_addition

  Expecting function call to invalid
  Expecting function call to invalid
  Expecting function call to add

  PASSED verify function call to invalid
    expected .......... invalid
    actual ............ invalid

  PASSED verify function call to invalid
    expected .......... invalid
    actual ............ invalid

  PASSED verify function call to add
    expected .......... add
    actual ............ add

PASSED test test_addition
```

The calculate function do not see any difference compared to its normal environment.

If the expected function is not present in the unit, the framework will report that it is called and return 0 to the unit that is calling it. If the function is present in the unit, the framework will simply report that it is called and afterwards call the function.

2.6 Function return values

Zero is often an adequate value to return from mock functions. In addition - by the use of an optional paremeter to the `EXPECT` macro, any non-zero value can be returned to the unit.

```
1 void test_addition(void) {
2   EXPECT  ( invalid );
3   EXPECT  ( invalid, TRUE );
4   VALIDATE(calculate(3,7,'+'), -1);
5 }
```

As sample5 shows above[1], the `EXPECT` macro is used to indicate to the framework, that this test expects `invalid()` function calls twice. The first call to invalid should return default 0 (`FALSE`) and the second should return `TRUE`. If the second `invalid()` function call is detected, the framewrok will make the mock return the value `TRUE` to `calculate`.

If the actual and expected function call matches, and the expected function is present in the unit, the framework will not call that function, but instead return the value given by the test.

If the actual and expected function call do not match and the expected function is not present in the unit, the framework will report an error that an unexpected function call was detected.

If the actual and expected function call do not match and the actual is present in the unit, the framework will not interfere. The unit will then simply call that function.

2.7 Function parameters

As part of the function validation, it is also possible to validate the parameters passed from the unit.

```
1 void test_addition(void) {
2   EXPECT  ( invalid );
3   EXPECT  ( invalid );
4   EXPECT_MOCK ( add, check_add );
5   VALIDATE(calculate(3,7,'+'), EXPECTED_SUM);
6 }
```

The EXPECT_MOCK macro shown in sample6 above[1], instructs the framework to expect a function call to `add`. `check_add` is a reference to a mock that will verify the arguments being passed to `add`. This mock function must have the same prototype as the function it validates. The sample function `check_add` shown below has the same prototype as the function `add` called by `calculate`.

```
1 int check_add(int left_operand, int right_operand) {
2   VALIDATE(left_operand,3);
3   VALIDATE(right_operand,7);
4   return 10;
5 }
```

The function validates that each of the parameters are as expected for this test. Other tests might have other expectations to the parameters, so they will use another mock function. It is possible to have many mock functions, each of which can be used to test parameters for `add` in different scenarios. The output from the mock function `check_add` will look something like this[2]

```
PASSED verify left_operand
  expected .......... 3 (3)
  actual ............ 3

PASSED right_operand
  expected .......... 7 (7)
  actual ............ 7
```

2.8 Return values from mock functions

The mock function has the same prototype as the function is validates, so it is possible for it to return a value, that the framework returns to `calculate`.

The mock shown below will return `EXPECTED_SUM`, which the framework will return back to `calculate`. Other tests may require a different return value, so they will use another mock. It is possible to have many mocks, each of which returns different values for different test scenarios.

```
int check_add(int left_operand, int right_operand) {
  VALIDATE(left_operand,3);
  VALIDATE(right_operand,7);
  return EXPECTED_SUM;
}
```

If the function mock is mocking a function that is present in the unit, it is possible, for the mock function to call that function.

In sample7 below[1], the function `add` is present in the unit

```
  int check_add(int left_operand, int right_operand) {
    VALIDATE(left_operand,3);
    VALIDATE(right_operand,7);
    return add(left_operand, right_operand);
  }
```

Calling the mocked function is a possibillity - it is not required.

2.9 Input to mock functions

As described in section 2.6 the test can provide the return value by the use of the optional return value parameter to the macro `EXPECT`, if the default value of zero is not adequate.

There is also an optional parameter to the `EXPECT_MOCK` macro. This parameter can be used by the test to pass information to the mock function.

The parameter given to `EXPECT_MOCK` can be read from within the mock function using the macro `MOCK_VALUE`. This can be used to pass parameters between tests and mock functions as shown in sample8 below[1]

```
void test_addition(void) {
  EXPECT  ( invalid );
  EXPECT  ( invalid );
  EXPECT_MOCK ( add, check_add , EXPECTED_SUM);
  VALIDATE(calculate(3,7,'+'), EXPECTED_SUM);
}
```

If the test is constructed as shown above, the value `EXPECTED_SUM` can be extracted and returned from the mock like this:

```
int check_add(int left_operand, int right_operand) {
  VALIDATE(left_operand,3);
  VALIDATE(right_operand,7);
  return MOCK_VALUE;
}
```

The same thing can be achived by having several different mocks each returning their own unique value, but, by the use of `MOCK_VALUE`, the same mock can be shared between several tests.

2.10 Mocking function calls

Sometimes it is not of any interest for a test, wheither the unit makes a call to a particular function or not. By the use of `ALLOW` macro, shown in sample9 below,[1] a test can indicate to the framework, that function calls to this function is valid at all times during the test.

```
1 void test_addition(void) {
2   ALLOW ( invalid );
3   EXPECT_MOCK ( add, check_add , EXPECTED_SUM);
4   VALIDATE(calculate(3,7,'+'), EXPECTED_SUM);
5 }
```

Functions that are present in the unit and not listed in a `EXPECT` or `EXPECT_MOCK` are automatically ignored by the framework. In those cases the framework will not interfere and it will simply transfer control to the function.

If the function given to ALLOW is not present in the unit, the framework will return 0 to the unit whenever the function is called.

With the use of the optional second parameter the default mock can be instructed to return any value to the unit. sample10 shown below[1] illustrates this, by making the default mock for `invalid` return `TRUE` at all times during the test.

```
1 void test_addition(void) {
2   ALLOW   ( invalid, TRUE );
3   VALIDATE(calculate(3,7,'+'), ERROR);
4 }
```

It is also possible to replace the default mock entirely for the duration of the test. By the use of `ALLOW_MOCK` as shown in sample11 below,[1] the function `invalid` will be mocked by function `mock_invalid`. The mock function must have the same function prototype is the function is mocks.

```
1  int mock_invalid(int parameter) {
2    return FALSE;
3  }
4
5  int check_add(int left_operand, int right_operand) {
6    VALIDATE(left_operand,3);
7    VALIDATE(right_operand,7);
8    return MOCK_VALUE;
9  }
10
11 void test_addition(void) {
```

```
12    ALLOW_MOCK  ( invalid, mock_invalid );
13    EXPECT_MOCK ( add, check_add , EXPECTED_SUM);
14    VALIDATE(calculate(3,7,'+'), EXPECTED_SUM);
15  }
```

2.11 Mocking c library

Eventhough standard C library functions can be mocked, it is worth to notice that some of these may in fact be macros disguised as functions. This depends on the library installed, but some versions of stdlib may have this issue for `feof`, `setjmp` and `longjmp`.

The `exit` function is generally not mockable - typically the compiler will assume that `exit` never returns to the caller. Therefore it will not generate normal stack cleanup code and the mock functionality will not work correctly.

`sample12` below,[1] is an example on a test that mocks `printf` in order to make a general validation of the format string.

```
1   void mock_printf(char *fmt, ...) {
2     VALIDATE(fmt==0, FALSE);
3   }
4
5   void test_printf(void) {
6     EXPECT(invalid, TRUE);
7     ALLOW_MOCK( printf, mock_printf);
8     calculate(3,7,'+');
9   }
```

2.12 Calling mocked functions from the mock

Upon entering a mock function, the framework disables all mocking of the function being mocked. It is therefore possible to execute the function being mocked from within the mock function. Upon return from the mock, or if the mock executes another test, the mocking of that function is enabled again. An example (sample13) is shown below.[1]

```
1   void mock_printf(char *fmt, ...) {
2     va_list args;
3     static int count = 0;
4     printf("*** printf called %d times *** \n", count++);
5     va_start(args,fmt);
6     vprintf(fmt, args);
7   }
8
9   void test_printf(void) {
10    EXPECT(invalid, TRUE);
11    ALLOW_MOCK  ( printf, mock_printf);
12    calculate(3,7,'+');
13  }
```

If the mocked function is not present in the unit, calling it from a mock will simply call the default mock for that function.

2.13 Organizing tests

When running the above tests, the framework validates that all the expected functions are called in the order as defined by the test and that no unexpected function was called. The test will validate if the parameters were correct and simulate the output as defined by the test. At the very end, the framework will validate the data returned from the unit. This would look something like this in the log[2]

```
Executing test test_addition

  Expecting function call to invalid
  Expecting function call to invalid
  Expecting function call to add

  In stub invalid

  PASSED verify function call to invalid
    expected .......... invalid
    actual ............ invalid

  In stub invalid

  PASSED verify function call to invalid
    expected .......... invalid
    actual ............ invalid

  In stub add

  PASSED verify left_operand
    expected .......... 3 (3)
    actual ............ 3

  PASSED right_operand
    expected .......... 7 (7)
    actual ............ 7

  PASSED verify calculate(3,7,'+')
    expected .......... EXPECTED_SUM (10)
    actual ............ 10

PASSED test test_addition
```

In order to test the entire module and not just one function several tests must be combined - for example, software that operates in an event driven environment will typically implement some kind of state machine. In those environments, several events are required to test the unit and a complete test scenario may require several tests to be executed. TestApe post no restrictions on how the tests are organized. In fact, they can be nested to allow for whatever

test organization that is appropriate for testing the unit. e.g. the nested test below would be executed using `EXECUTE(scenario_sunshine)`

```
void scenario_sunshine(void) {
  EXECUTE(test_receive_this_event_wait_for_that_event);
  EXECUTE(test_receive_that_event_and_finish);
}
```

It is also common to group several tests to form testcases and testsuites. It is a possibility for the test designer to form testcases by nesting tests and to form testsuites by nesting testcases. This is illustrated in sample14 shown below.[1]

```
void test_mapping_minus(void) {
  EXPECT  ( invalid );
  EXPECT  ( invalid );
  EXPECT  ( subtract );
  calculate(1,1,'-');
}

void test_mapping_plus(void) {
  EXPECT  ( invalid );
  EXPECT  ( invalid );
  EXPECT  ( add );
  calculate(1,1,'+');
}

void testcase_operator_mapping(void) {
  EXECUTE(test_mapping_plus);
  EXECUTE(test_mapping_minus);
}

void testsuite_calculator(void) {
  EXECUTE(testcase_operator_mapping);
}

void testmain(void) {
  EXECUTE(testsuite_calculator);
}
```

2.14 Ranged tests with parameters

A very useful feature of the TestApe framework, is its ability to execute the same test with a given list of parameters, e.g. a range between 0 and 255, or a set of strings e.g. `""`, `NULL`, `"SAMPLE"` that is known to be of importance for the unit being tested. This also gives the posibility to vary the test slightly, in order to achieve the final increase in coverage. Of course, this can be done by copying existing tests; changing a parameter here and there and implementing new mocks. However - it is much easier to use the possibility to execute a ranged test with parameters using the macro `PARAMETER`, `PARAMETER_RANGE`, and `PARAMETER_SET`.

These macros will prepare one or more values for the next executing test. Suppose you want to test `myfunc` using a bool input parameter with values `true` and `false`, then you would

prepare the test using `PARAMETER(true);` `PARAMETER(false);` and execute the test as using `EXECUTE(myfunc)`

Some more examples are shown in the the code below. In the sample directory, sample15 has a more complete working example.[1]

```
void interesting_integers(void) {
  PARAMETER(0);
  PARAMETER(-1);
  PARAMETER_RANGE(128,255);
  PARAMETER("some input string");
  PARAMETER("");
  PARAMETER(NULL);
  EXECUTE(interesting_parameters);
}
```

This would generate the following output[2]

```
Executing test interesting_parameters (step 1 of 133 using value 0)
....
PASSED test interesting_parameters
Executing test interesting_parameters (step 2 of 133 using value -1)
....
PASSED test interesting_parameters

....

Executing test interesting_integers (step 130 of 133 using value 255)
....
PASSED test interesting_parameters
Executing test interesting_integers (step 131 of 133 using value "some input string")
....
PASSED test interesting_parameters
Executing test interesting_integers (step 132 of 133 using value "")
....
PASSED test interesting_parameters
Executing test interesting_integers (step 132 of 133 using value NULL)
....
PASSED test interesting_parameters
```

When testing a value in a range from one to 10000, the same test is literally executed 10000 times. This can take some time and generate huge logfiles. As an option, the amount of values tested, and/or the output generated, can be limited from the command line when invoking the test executable. See section 4.1

2.15 Validating ranged tests with parameters

Each time an input parameter is changed the test is executed one more time. The TestApe will invoke the test until the list of values is exhausted. With the previous example in mind, the macro `PARAMETER_VALUE` can be used to extract the values as shown in sample16 below[1]

```
void test_addition(void) {
  EXPECT    ( invalid );
  EXPECT    ( invalid );
  EXPECT    ( add );
  VALIDATE(calculate(PARAMETER_VALUE,PARAMETER_VALUE,'+'), 2*PARAMETER_VALUE);
}

void testmain(void) {
  PARAMETER_RANGE(10,20);
  EXECUTE(test_addition);
}
```

As the input is changed, most likely so too are the validations and the list of expected function calls. This can also be implemented by the use of `PARAMETER_VALUE`. An example (sample17) is shown below.[1]

```
void test_addition(void) {
  EXPECT ( invalid );
  EXPECT ( invalid, PARAMETER_VALUE );

  if (!PARAMETER_VALUE)
    EXPECT_MOCK ( add, check_add );

  if (!PARAMETER_VALUE)
    VALIDATE(calculate(3,7,'+'), EXPECTED_SUM);
  else
    VALIDATE(calculate(3,7,'+'), ERROR);
}

void testmain(void) {
  PARAMETER(TRUE);
  PARAMETER(FALSE);
  EXECUTE(test_addition);
}
```

The value being tested can also be passed to one of the mocks, e.g. the function `check_add` shown below

```
void test_addition(void) {
  EXPECT  ( invalid );
  EXPECT  ( invalid );
  EXPECT_MOCK ( add, check_add , PARAMETER_VALUE);
  VALIDATE(calculate(3,7,'+'), PARAMETER_VALUE);
}
```

The mock can pick up the value as described in 2.6 or it can just pick up the value by referencing `PARAMETER_VALUE` directly. `PARAMETER_VALUE` will remain in scope for the test and mock functions throughout the duration of the test.

2.16 Parameter sets

It is possible to give parameter sets as input to test. This can be done like it is shown in sample18 below[1]

```
typedef struct _mytest_t
   { int left; int right; int result; } mytest_t;

int check_add(int left_operand, int right_operand) {
  mytest_t  *tmp = (mytest_t*)MOCK_VALUE;
  VALIDATE(left_operand,tmp->left);
  VALIDATE(right_operand,tmp->right);
  return tmp->result;
}

void test_addition(void) {
  mytest_t  *tmp = (mytest_t*)PARAMETER_VALUE;

  EXPECT  ( invalid );
  EXPECT  ( invalid );
  EXPECT_MOCK ( add, check_add , PARAMETER_VALUE);
  VALIDATE(calculate(tmp->left,tmp->right,'+'), tmp->result);
}

void testmain(void) {
  mytest_t  test_calc_add_parameters[] = {{1,1,2},{3,4,7},{6,6,12}};
  mytest_t  test5_5_10[] = {5,5,10};

  PARAMETER_SET(test_calc_add_parameters);
  PARAMETER    (test5_5_10);
  EXECUTE(test_addition);
}
```

2.17 Signal handling

TestApe will react to signals from the operating system. If the signal is caused by one of the VALIDATE macros, for example by accessing memory through an invalid pointer, that validation will fail and execution of the test will continue.

If the signal is caused by one of the functions in the unit being tested, TestApe will terminate the current test and carry on with the next test, however if parameters are used and the test is setup to run more than once, the test will execute again with next parameter until there is no more parameters in the list.

Sample19[1] will run test `divide` twice with parameters 0 and 1. The first execution will fail because a signal is raised when the denominator is zero. The second will pass. The output looks like this[2]

```
  Executing test test_division (step 1 of 2) using range 0-1 value 0

    Expecting function call to divide
```

```
    PASSED verify function call to divide
      expected .......... divide
      actual ............ divide

  FAILED test test_division (caught exception, signal 0xc0000094)
    caught exception in test test_division

  Executing test test_division (step 2 of 2) using range 0-1 value 1

    Expecting function call to divide

    PASSED verify function call to divide
      expected .......... divide
      actual ............ divide

  PASSED test test_division
```

As default, signals will cause the test to fail, but that can be changed. TestApe will pass the signal number to the function `testape_exception_handler`. The default implementation of this will simply return 1. Any non-zero value returned from here will cause the test to fail. If the return value is zero the test will pass, so by using any of the mocking functions, e.g. `EXPECT(testape_exception_handler, 0)` at the proper place, the same test can be made to pass. For example, if sample19 is modified like this

```
1  void test_division(void) {
2    EXPECT( divide );
3    if (0 == PARAMETER_VALUE) {
4      EXPECT(testape_exception_handler, 0);
5    }
6    calculate(10,PARAMETER_VALUE,'/');
7  }
8
9  void testmain(void)  {
10   PARAMETER_RANGE(0,1);
11   EXECUTE(test_division);
12 }
```

the output becomes[2]

```
  Executing test test_division (step 1 of 2) using range 0-1 value 0

    Expecting function call to divide
    Expecting function call to testape_exception_handler

    PASSED verify function call to divide
      expected .......... divide
      actual ............ divide

    PASSED verify function call to testape_exception_handler
      expected .......... testape_exception_handler
      actual ............ testape_exception_handler
```

```
  PASSED test test_division

  Executing test test_division (step 2 of 2) using range 0-1 value 1

    Expecting function call to divide

    PASSED verify function call to divide
      expected .......... divide
      actual ............ divide

  PASSED test test_division
```

2.18 Arguments to tests

In its most simple form, a test is written like this `void test(void)`, but a test can also be written with arguments as shown in sample20[1] below

```
1  void test_deg2rad(double degrees, double radians, double *error) {
2    double result = deg2rad(degrees);
3    *error = result-radians;
4    VALIDATE_FP(result , radians);
5  }
6
7  void testmain(void) {
8    double rounding_error;
9    EXECUTE(test_deg2rad, 360, 2*PI, &rounding_error);
10   COMMENT("Rounding error is %f\n", rounding_error );
11 }
```

If the test are executed repeatedly, e.g. they have been setup with a `PARAMETER` macro, each call will use the same function arguments. By passing `PARAMETER_VALUE` as a one of these, this argument can be made to vary with each test run, e.g. in sample21[1]

```
1  void test_deg2rad(double degrees, double radians) {
2    double result = deg2rad(degrees);
3    VALIDATE_FP(result , radians);
4  }
5
6  void testmain(void) {
7    PARAMETER_RANGE_FP(2*PI, 10*PI, 2*PI);
8    EXECUTE(test_deg2rad, 360, PARAMETER_VALUE_FP);
9  }
```

TestApe will pass commandline arguments after the `--args` tag to testmain, so if the test executable are invoked like this

```
./sample22 --args 4
```

`argc` will be 1 and `argv[0]` will be "4" in the testmain function declared in sample22[1] below

```
void test_deg2rad(double degrees, double radians) {
  double result = deg2rad(degrees);
  *error = result-radians;
  VALIDATE_FP(result , radians);
}

int testmain(int argc, char *argv[]) {
  double rounding_error;
  int number = atoi(argv[0]);
  EXECUTE(test_deg2rad, 360, number*PI, &rounding_error);
  COMMENT("Rounding error for number equals %d is %f\n", number, rounding_error);
}
```

2.19 Floating point support

TestApe supports validating floating point values and mocking of functions using or returning floating point values. The floating point types float and double are supported.

To validate floating point values, use either normal `VALIDATE` function or dedicated floating point version `VALIDATE_FP`. The first one only validate the integer part of a value, where as the second one validates the full floating point value with some allowed degree of inaccuracy. This is illustrated in sample23[1] below.

```
#define PI (4.0 * atan( 1.0 ))

void test_floating_point(void) {
  VALIDATE   (PI,PI);          // will pass - accurate within limit
  VALIDATE_FP(PI,3.14);        // will fail - not accurate enough
  VALIDATE_FP(PI,3.1415926);   // will pass - accuracy within limit
}
```

The required accuracy can be changed, by supplying the `testape_validate_fp_accuracy()` function.

```
double testape_validate_fp_accuracy() {
  return 0.1f;
}
```

If this function is not present in any test, the instrumenter will provide a default returning an accuracy of 0.0001f.

2.20 Floating point return values

The following section illustrates various ways to test the return value of the function `pi` used by `rad2deg` and `deg2rad` in converter.c shown below

```
double pi() {
  return 3.14f;
```

```
}

double rad2deg(double rad) {
  return 180*rad / pi();
}

double deg2rad(double deg) {
  return deg * pi() / 180;
}
```

`EXPECT`, `ALLOW` will accept functions that returns float or doubles and all other testape macros will accept either floating point values or integer values as arguments.

So it is possible to write a test like it is shown in sample24[1] below

```
#define PI (4.0 * atan( 1.0 ))

double testape_validate_fp_accuracy() {
  return 0.01;
}

void test_rad2deg(void) {
  EXPECT ( pi );
  ALLOW  ( pi );
  VALIDATE   ( rad2deg(0), 0);
  VALIDATE   ( rad2deg(2*PI), 360);
  VALIDATE_FP( deg2rad(0), 0);
  VALIDATE_FP( deg2rad(360), 2*PI);
}
```

The first two validations will convert the floating point return value to integers before validating them against the integer limits e,g, $0 == 0$, and $360 == 360$. The later two, will directly validate the floating point return value from `deg2rad` against PI. e.g. 0.0f == 0.0f and 6.283185307f == 6.28f. These validation will both pass, as they are accurate with 0.01f as indicated in the `testape_validate_fp_accuracy` function.

It is also possible to use macros `EXPECT(function, retval)` and `ALLOW(function, retval)` with floating point values. These macros will evaluate *retval* twice each time the macro is used, as the framework will save two versions of *retval*. First version as an integer representation and second version as a floating point representation. If *function* is a function returning a floting point, the floating point version will be returned to the calling function and similar if *function* is a function returning an integer value, the integer version will be used.

In sample25[1] below, the floating point representation is used to mock the return value from `pi`.

```
#define PI (4.0 * atan( 1.0 ))

void test_deg2rad(void) {
  EXPECT  ( pi, PI );
  EXPECT  ( pi, PI );
  VALIDATE_FP( deg2rad(0), 0);
  VALIDATE_FP( deg2rad(360), 2*PI);
}
```

Notice the absence of `testape_validate_fp_accuracy` and that the simulated value used are more accurate than the actual implementaion of `pi`. Therfore both validation will pass with the default accuracy of 0.00001f.

The double evaluation of *retval* in the EXPECT and ALLOW macros, may cause cause problems if the *retval* parameter expands to a function for which its return value varies each time it is called. Make a local variable, if you have this problem.

2.21 Mocking functions returning floating point values

Sample26[1] below illustrates how to mock a function that is returning a double and how to use the macros `MOCK_VALUE` and `MOCK_VALUE_FP` to access both representations of the simulate value.

```
#define PI (4.0 * atan( 1.0 ))

double mock_pi() {
  return MOCK_VALUE_FP;
}

double integer_mock_pi() {
  return MOCK_VALUE;
}

void test_deg2rad(void) {
  EXPECT_MOCK( pi, mock_pi, PI );
  VALIDATE_FP    ( deg2rad(360), 2*PI);
  EXPECT_MOCK( pi, integer_mock_pi, PI );
  VALIDATE       ( deg2rad(360), 2*PI);
}
```

It is safe to mix the usage of these macros with both integers or floating points values, i.e if `MOCK_VALUE_FP` is used as an integer value, the compiler will cast it to an integer value, and if `MOCK_VALUE` is used as a floating point value, it will be cast to an floating point value.

2.22 Using the mocking system with other test systems

It is possible to use the mocking system only as illustrated by the example below[1]

```
void mock_printf(char *fmt, ...) {
  VALIDATE(fmt==0, FALSE);
}

void test(void) {
  MOCK(printf, mock_printf); /* enable mock  */
  ... other test code
  MOCK(printf, 0);           /* disable mock */
}
```

2.23 Adding and removing object files from unit

In large software projects the availability of object files may depend on the progress from other groups. It is possible to write the tests, so that they are invariant to the presence of an object file. As soon as the interface has been agreed upon, TestApe will make it possible to write the software using it. Later, when the implementation is ready, the object file can be added, and the same test can be reused to verify that the implementation in the object file, is in line with expectation. Imaging that `add.c` is a large and complex function that is not initially available when developing `calculator.c`. The expected behavior of `calculator.c` can be tested like it shown below.

```
1 void test_addition(void) {
2   EXPECT  (add,10);
3   VALIDATE(calculate(3,7,'+'), 10);
4 }
```

When `add.obj` becomes available it can be added to the commandline

2.24 Handling of function calls explained in detail

I order to test the behavior of a higher level function it is typically not enough to look at the output it delivers. The behavior of the function itself is also of interest.

The difference between EXPECT and ALLOW macros

The macros EXPECT and ALLOW (and their MOCK variants) allows you to test this behavior. You can validate the parameters to any helper functions that is called and you can control the program flow by deciding their return values. If you want to validate if these function call happens at the right time and in the right order, you use the EXPECT macros. If you dont care about that, you use the ALLOW variants. The ALLOW and EXPECT macros takes one optional parameter - the return value that TestApe will return to the unit when the function is called.

The MOCK variants

Many times more elborate control and validation are needed than what can be achieved with the ALLOW and EXPECT macros - for that, each of them has a MOCK variant. The EXPECT_MOCK and ALLOW_MOCK takes an additional mandatory mock function parameter. These macros behaves like their counterparties, but instead of returning 0 or a simple value to the unit, these macros pass control to this mock function whenever the function is called. With the mock function more elborate tests can be written, e.g. they check the parameters, call other functions, or execute new tests. The MOCK variants takes also takes one or more optional parameters. These can later be retrieved in the mock function when it is called.

The allow, expect, and mock function lists

During any test, three lists of function call elements are maintained - the allow list, the expect list and the mock list. Each element contains information about one function, e.g. what mock to use (if any) and the return value to use (if any) Every function call made by the unit during execution of test is handled according to the information found in one of these lists.

How the program flow is affected by the contents in the function lists

When TestApe detects afunction call, the execution of the unit are affected according to the rules outlined below.

First TestApe determines if the function called are present in the mock list. If so, the function is called according to the information in this list. If the function is not found in mock list, it is then determined if it is present at the end of of the expect list. If that is the case the element from the end of the expect list is removed. A message is logged about a sucessfull validation and the function is called according to the information in the element. If TestApe cannot find the function in the top of the expect list, it then looks in the allow list. If the function is present here, the function is called according to the information in the element found. If the function call are not in the mock or the allow list and not at the end of the expect list an error message is logged and the function is just called directly without redireting to a mock and without changing its return value.

Controlling the content of the function lists

The contents of the mock list, are manually controlled by the test. Every MOCK adds to the mock list (or removes from the list if mocking of the function is disabled).

Any of the ALLOW macros will add to the allow list. In addition the contents in the allow list are inerited from the parent test and restored when the test completes.

The expect list are also inherited from the parent test and any of the EXPECT macros will add to the expect list. Any function call matching the element at end of the list will remove this element from the list. When a test completes the list must be empty, if not an error is logged.

Initial contents of the function lists

The initial content of the mock and the expect lists are empty lists.

TestApe assumes that you are interested in testing at the boundaries of your unit, so upon startup, all internal functions are automatically added to the allow list and inherited to each test. This means internal functions has to be added explicitly to the expect list, if you you want to validate the order of function calls to these funcitons. Oppositely it also means, that external functions has to be added explicitly to the allow list, if you dont want to validate the order of function calls to these functions.

Chapter 3

Using the TestApe instrumenter

The TestApe instrumenter is designed to generate default mocks for those functions that cannot be resolved by the linker. The generated mocks allows simulating and testing of data flowing between the external functions and the unit being tested.

At the final link stage, the invocation of the linker is passed through the TestApe instrumenter. If the instrumenter determines that the testape library are not used, the instrumenter will simply invoke the linker in pass through mode. If the testape library is used, automatic mock generation will be initiated and the TestApe logo will be displayed.

```
   -------------
------ OOO ----
        ++
     ,,, ++
   (o_o)**
    (_)**
  **   +
 *+ + .
 ++     ()
 oo  () oo
     oo
```

The instrumenter will also check and notify about newer versions when these becomes available.

Supported linkers includes LINK.EXE (including when LINK.EXE is invoked though CL.EXE) and ld (including when ld in invoked through g++ or gcc)

3.1 The TestApe instrumenter commandline options

The syntax and options that can be used when invoking the instrumenter is shown below.

```
testape [-o][-nvc][-nl][-mgw] ld [linker arguments] testape.a
testape [-o][-nvc][-nl][-mgw] gcc [compiler parameters] testape.a
testape [-o][-nvc][-nl] LINK [linker arguments] testape.lib
testape [-o][-nvc][-nl] CL [compiler parameters] testape.lib
```

The instrumentation is done in three passes; pass 1 will execute compiler/linker and analyze the amount of unresolved externals, pass 2 will generate and compile instrumentation code, and pass 3 will link it all together.

It is possible to ask the instrumenter to skip pass 3, so that the linking can be done manually. This is done by using the option `-nl`. This will make the instrumenter stop after pass2 and generate the file `testape.o` that contains the instrumentation code.

The option `-o` is used to troubleshoot problems. When present, the command lines for each pass is shown, and the intermediate files are kept.

The option `-mgw` is used with gcc to activate MinGw support on windows. TestApe cannot reliably detect the precense of MinGW gcc compiler, but with this option the instrumenter can be forced to mingw gcc mode.

Upon exit, the instrumenter will alert, if a never version of the packages exists. The option `-nvc` is used to disable this check.

The `compiler parameters` or `linker arguments` can be one or more parameters normally reconized by gcc/ld/LINK or CL.

3.2 Installing TestApe on Linux

The `gcc`/`g++` needs access to interface file and `ld` needs access to the library file. In addition certain system libraries are required. This is all taken care of by the package manager.

The package manager will put the binaries in `/usr/bin`, `/usr/include`, `/usr/lib` and the documentation and examples in `/usr/share/doc/testape`

If you want to do it manually, unpack the tar ball and put the instrumenter,interface and library in the proper directories. Remember to link to testape.a from `/usr/lib/libtestape.a` if you want to use `-ltestape` option.

The library and instrumenter will depend of the following libraries

```
libc6 (>= 2.4),
libgcc1 (>= 1:4.1.1)
libstdc++6 (>= 4.6)
```

Debian

To install TestApe on a Debian based system run

```
dpkg -i testape_1171_i386.deb
```

On a Redhat based system run

```
rpm -i testape-1171-2.i386.rpm
```

Raspbian on Raspberry Pi

The TestApe mocking system supports the ARM processor. The release is tested on a QEMU and RaspBerry Pi but boards with other arm processors will work as well. Note however, that ARM thumb mode instruction set is currently not understood by the mocking system.

To install TestApe on raspbian run.

```
dpkg -i testape_1171_armhf.deb
```

3.3 Using TestApe with gcc

To use the instrumenter with the linux linker `ld` simply link with the testape library and invoke the instrumenter in front of the final linker command as shown in the examples below

```
testape ld unit_a.o unit_d.o ad_test.o testape.a
testape ld unit_a.o unit_d.o ad_test.o /path/testape.a
```

or when the linker is used through `gcc`/`g++`

```
testape gcc unit_a.c unit_d.c ad_test.c testape.a
testape gcc unit_a.c unit_d.c ad_test.c -ltestape
```

If the libray is used together with tests written in C++, or if the tests are compiled with g++, the interface needs to be put inside a `extern "C"` declaration in order to be linkable with the C++ program.

Eventhough the instrumenter cannot autogenerate mocks for code that is written in C++, it is still possible to use the the framework to test the code. All mocks will have to be written manually.

For further information on TestApe with C++ ask on the forum at http://testape.com.

3.4 Using TestApe with gcc arm cross compiler

TestApe supports cross compiling on intel platforms for the ARM processor through the use of arm-linux-gnueabi-testape.a. This library is available in the linux installation packages. Cross compilation is not supported on any of the windows packages.

The implementation uses the arm-linux-gnueabi GNU sstandard C library that ships with the gnueabi toolchain. As default this library uses the software floating point model, but technically there should be no reason why -mfloat-abi=softfp and mfloat-abi=hard should not work. Also other standard C libraries should work as long as they adhere to the EABI standard. The following example shows how to compile and execute a test with the cross compiler and run the test in QEMU arm emulator.

```
testape arm-linux-gnueabi-gcc -o sample1 sample1.c calc.c add.c arm-linux-gnueabi-testape.a
qemu-arm -L /usr/arm-linux-gnueabi/lib/libc ./sample1
```

ARM and floating point return values

The gnueabi toolchain uses the software floating point model as default. It passes floating point value through the arm registers. The stack and register aligment as well as the floating point representation are affected by your choise of float versus double.

This is in contrast to the x86 platform which passes floating points on a seperate FPU stack and represents all type internally as 80 bit floating point. So on x86 it is not important, if your test do not accurately indicate if a return value is a floating point or not. TestApe will just push an integer representation onto the normal stack and a floating point representation onto the FPU stack.

However on the arm it is important that you cast the mock return value to the same return type as the mocked function would return normally. For example on Intel platforms this will work

```
1 EXPECT(function_returning_float,10);
2 EXPECT(function_returning_double,10);
```

but on ARM platforms you need to do this

```
1 EXPECT(function_returning_float,(float)10);
2 EXPECT(function_returning_float,10.0f);
3 EXPECT(function_returning_double,(double)10);
4 EXPECT(function_returning_double,10.0);
```

Note, that this limiation only affects floating point return values - any other type of return value follows the same pattern as the x86 implementation.

3.5 Installing TestApe on windows

To install TestApe on a windows machine download and run the installer.

The installer will put the files in `C:\Program Files\Testape` and adjust the `PATH`, `LIB` and `INCLUDE` environment variables.

There will be two versions of the testape library installed - one for visual studio (testape.lib) and one for mingw-w64 gcc (testape.a).

To get started you can use the menu item 'Try Examples' that will launch a commandline where the samples can be compiled.

3.6 Using TestApe with Visual Studio

To use the instrumenter with the Visual Studio command line tools `link.exe` simply link with the testape library and invoke the instrumenter in front of the final linker command as shown in the examples below

```
testape.exe link.exe unit_a.obj unit_d.obj ad_test.obj testape.lib
testape.exe link.exe unit_a.obj unit_d.obj ad_test.obj \path\testape.lib
```

or when used through cl.exe

```
testape.exe cl.exe unit_a.c unit_d.c ad_test.c testape.lib
testape.exe cl.exe unit_a.c unit_d.c ad_test.c \path\testape.lib
```

TestApe test executable can be debugged using the Visual Studio debugging environment. The following command lines is an example on how it can be done.

```
C:\testape\sample>cmd /k ""c:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"" x86
Setting environment for using Microsoft Visual Studio 2008 x86 tools.
C:\testape\sample>testape cl sample1.c calc.c add.c testape.lib
C:\testape\sample>devenv /debugexe sample1.exe
```

3.7 Using TestApe with mingw-w64 gcc

TestApe will work with the gcc compiler that ships with cygwin and with mingw-w64. The instrumenter will link to the standard c library. Cygwin ships with its own standard library (libc), whereas MinGW-w64 uses the standard c library that comes with Windows (mscvrt)

As the instrumenter cannot reliably detect if you're using the mingw or the cygwin version of gcc, you need to help it and provide the option -mgw when running with the mingw-w64 tool chain,

3.8 Troubleshooting

The instrumenter has a `-o` commandline switch to help troubleshoot. When invoked in this mode, the instrumenter will show command lines for each pass and keep the temporary files. Some typical problems are shown below.

Problems running any samples with mingw

You need to use the mingw-w64 distribution. TestApe will not work reliably with the old version of mingw as this toolchain may stop linking prematurely before disclosing all relevant unresolved externals.

Problems mocking malloc, free, printf and other clib functions

Most likely the problems are caused be the mix of static and dynamic linking. The testape library are statically linked and to avoid problems use the `-static` option to gcc when compiling the tests.

Creating mock for EXECUTE (and other TestApe macros)

Check that `testape.h` is included in the tests.

Creating mock for getreentrnet

Check that you are linking with all required standard libraries. Try and remove `testape` from the commandline and verify that you can compile the tests without the instrumenter and that you do not see unexpected unresolved external symbols.

Creating mock for _Z20testape_validate_intPcS_ii

Check that you have named the files with the extension `.c` if these are c files. Verify that you are compiling using gcc, not g++. If you want to to use g++ in order to write the tests in c++, the `#include <testape.h>` statement must be put inside a `extern "C"` declaration.

undefined reference to 'testape_validate_int'

Check that you are linking with library (testape.a/testape.lib). The order of libraries and tests given on commandline is important. Verify that the library is given later on commandline than the tests using it.

Failures when functions returning doubles are implicitly declared

The following code will fail to return correct value to the calling function

```
1 int degree_to_miliradians(int degrees) {
2   return 1000*deg2rad(degrees)
3 }
```

The code relies on the function `deg2rad`. This function accepts an integer and returns a double. However, in this example, `deg2rad` are implicitely declared to be of type `int deg2rad(int)` and not `double deg2rad(int)` as intented, so `degree_to_miliradians` will not work correctly.

However, it is possible to write a test, to discover this e.g.

```
1 void test_degree_to_miliradians(void) {
2   EXPECT(deg2rad, 2*PI);
3   VALIDATE(degree_to_miliradians(360), 1000*2*PI);
4 }
```

When TestApe detects a call to `deg2rad`, it correctly detects that deg2rad should return an integer, and will return (int)2*PI, e.g. 6, that will make `degree_to_miliradians` return 6000, and the test will fail.

However, if the bug is corrected, e.g.

```
1 double deg2rad(int degrees);
2
3 int degree_to_miliradians(int degrees) {
4   return 1000*deg2rad(degrees)
5 }
```

The TestApe will detect the `deg2rad` should return a floating point e.g. 2*PI, 6.28..., and that will make `degree_to_miliradians` return the correct value 6282, and the test will pass

Chapter 4

Using the TestApe test executable

The test executable can be run from commandline like any other programs generated by the compiler. The framework will launch `testmain` using `EXECUTE(testmain)` after initialization and analysis of the commandline arguments.

4.1 Commandline options

If no options are given, all tests are run and the test will output a short summary and report any errors to stderr. This behavior can be changed by providing one or more of the following options

```
test executable
 [options]
 [[-run | --run-tests] identifiers[,identifiers]]
 [--args args]

-a --args                     : arguments after --args are passed to testmain
-e --first-error              : execute the test having the first error
-ide --show-test-identifiers  : enable error identifiers in testlog output.
-idt --show-error-identifiers : enable test identifiers in testlog output.
-tag xx --log-tag xx          : enable log tag xx in testlog output.
-notag --no-log-tag           : disable log tag in testlog output.
-q --quiet                    : disable all output.
-oe --output-error            : output errors only.
-ol --output-log              : output errors, summary and log.
-h --help                     : print this help.

 identifiers -id | id- | id-id : until id | from id to id | from id
 id [t]nnn | testname | ennn   : test nnn | test testname | test having error nnn
```

All arguments after `--args` or `-a` parameter are treated as arguments to `testmain`.

`-q` or `--quiet` are used to prevent all output from testape. `-oe` or `--output-error` will disable all output except error messages written to stderr. `-ol` or `--output-log` will enable detailed log output. The log report will be written to stdout.

The name of a test can be used as identifier on the commandline. [2] In addition each test is numbered. These names or numbers can be used as identifiers. Each argument that accept an identifier, also accepts a list of identifiers and /or a range of identifiers. Each identifier in a list, is seperated by ','. The start and stop values in a range is separated by '-'. If no start value is given the first test is used as start value. If no stop value are given, the last test is used as stop value. Some identifier examples are shown below

```
test_load_my_app
test_load_my_app,scenario_sunshine_all_ints
scenario_sunshine_all_ints-
-scenario_sunshine_all_ints
test_load_my_app-scenario_sunshine_all_ints
1
1-10,4,124
```

4.2 Exit codes

If any errors were detected during execution of the tests in `testmain`, the framework will make the test executable return a nonzero errorlevel or exit code.

The effect of the exit code varies depdendent on the context in which the test executable is running. If the test executable is executed as part of a rule in a makefile , the exit code will make the makefile terminate the rule.

4.3 Memory analysis

The test executable can be instrumented using standard memory analysis and coverage tools. For example to execute with Valgrind memory analysis tools simply use `valgrind testexe`

4.4 Coverage analysis

Coverage data can be meassured using any 3rd party tool. E.g. to use purecoverage simply invoke purecoverage instrumentaiton on the test executable.

```
testape cl unit_a.c unit_d.c ad_test.c testape.lib /out:testexe.exe
purecov testexe
testexe
```

`gcov` is the GCC coverage tool. This tool will automatically collect the data created by the test executable, if the test executable it was compiled with coverage enabled. e.g.

```
testape gcc -ftest-coverage unit_a.c unit_d.c ad_test.c testape.a -o testexe
testexe
gcov -a uint_a.c uint_d.c
```

[2] The usage of test names as identifiers is not implemented yet.

4.5 Debugging test executable

The test executable can be debugged like any other executable. Breakpoints can be set at any statements in tests or in mock functions. In addition, it is also possible to break on specific errors.

When single stepping during debugging, note that EXECUTE is not always a single statement. When test parameters are given, EXECUTE will actually call the test several times.

In order to break on errors, the breakpoint must be set in the error handler. This function is called immediately before the statement having the error. Upon return from the error handler, the execution will continue at the point where the error was detected.

In order to provide an error handler, the test must implement a function having the predefined name `testape_error_handler`. The following shows an simple example

```
1 void testape_error_handler(void) {
2 }
```

Conditional breakpoints can be set on one of the externaly available values `testape_current_test`, `testape_current_test_id`, `testape_current_error`, or `testape_current_error_id`. These can be used for conditional breakpoint or for general info as shown below

```
1 void testape_error_handler(void) {
2   printf("Error found in '%s', test number %d.\n",
3              testape_current_test, testape_current_test_id);
4   printf("This error is '%s', error number %d.\n",
5            testape_current_error, testape_current_error_id);
6 }
```

The error handler is intenteded for breakpoints only. Any use of the framework within the handler, may have unexpected results. If no error handler is given in one of the test files, the instrumenter will provide an empty default.

4.6 Log output and error messages

Running `./sample23` in the sample directory will generate the following output

```
TestApe test executable
Unit testing for embedded software - http://testape.com

Initializing ./sample23

error e001: sample23.c:12: failed value of PI. Expected 3.140000 (3.14), was 3.141593.

Terminating ./sample23 - exitcode 1
  0 passed, 1 failed (1 errors), 0 skipped
```

The detailed log report is disabled by default, but it can be enabled from commandline with -ol option. In the log each log statement is prefixed with the tag `TestApe` and optionally the current test and/or error identifier as shown below.

```
./sample23 -ide -idt -ol
```

which will produce the following output instead

```
          testape: TestApe test executable
          testape: Unit testing for embedded software - http://testape.com
          testape:
          testape: Initializing ./sample23
          testape:   -ol ( errors:stderr, summary:stdout, log:stdout )
          testape:   -ide ( error identifiers shown in log )
          testape:   -idt ( test identifiers shown in log )
          testape:
t000 e000 testape: Executing test testmain
t000 e000 testape:
t000 e000 testape:   PASSED verify PI
t000 e000 testape:     expected .......... 3 (PI)
t000 e000 testape:     actual ............ 3
t000 e000 testape:
t000 e001 testape:   FAILED verify PI
error e001: sample23.c:12: failed value of PI. Expected 3.140000 (3.14), was 3.141593.
t000 e001 testape:     expected .......... 3.140000 (3.14)
t000 e001 testape:     actual ............ 3.141593
t000 e001 testape:
t000 e001 testape:   PASSED verify PI
t000 e001 testape:     expected .......... 3.141593 (3.1415926)
t000 e001 testape:     actual ............ 3.141593
t000 e001 testape:
t000 e001 testape: FAILED test testmain (1 error)
t000 e001 testape:   t000 e001 Verify floating point in test testmain
t000 e001 testape:
          testape:
          testape: Terminating ./sample23 - exitcode 1
          testape:   0 passed, 1 failed (1 errors), 0 skipped
```

Every error detected and every test execution is assigned a sequential identifier. Error identifiers are 3 digits shown prefixed by e, e.g. `e001`. Test identifiers are shown prefixed by t, e.g. `t001`.

These identifiers can be used from the commandline to limit the amount of test to run.

When an error is detected TestApe print an error message to stderr and add information about the error to the log.

The log will contain a line for the actual and expected value. Only 8 values will be shown on each line, so when validating file content, strings, structs and arrays, several lines may be shown. Differences between actual and expected are indicated with the ^ sign as shown below.

```
testape:   FAILED verify s
testape:     expected .......... [000] 4e 4f 54 20             'NOT '
testape:     actual ............ [000] 57 45 52 00             'WER.'
testape:                                ^^ ^^ ^^ ^^              ^^^^
```

Chapter 5

Customizing TestApe

It is possible to modify the testape library in order to customize error messages, startup code and os dependencies.

The library is a collection of objects files and by replacing one or more of these, the behavior of testape can be changed. You need to compile our own versions of these and replace them in testape library.

If you choose to make a copy of the library, please notice that it is important that the library ends with testape.a, e.g. `my_os_testape.a` is ok, but `testape_for_my_os.a` is not ok. If the library name do not end with testape.a it will not trigger the testape instrumentation when used on commandline.

If the name is correct, the instrumenter will analyze and generate some instrumentation code, and as its final task, it will link this code (found in `testape.o`) with your tests and the testape library.

It will by default link with the standard c library, but if you're building a bare metal configuraton this is most likely not what you want. You can ask the testape instrumenter to skip the final link task by passing the option `-nl`.

5.1 Error messages

The output to stderr during test execution can be customized by implementing the functions defined in `testape_messsages.h`. These functions (`testape_error_xxx`) are called by the framework whenever errors are detected.

The format that ships with testape, will allow IDE's like Visual Studio, Codelite and Eclipse to find the files and lines having the error, but you can customize choose whatever you want to be printed on stderr by selecting among the various function arguments.

The defult implementation (`testape_messsages.c`) is available for inspiration.

5.2 Startup code

The startup code are responsible for handling the commandline arguments and for executing the first test. In some situations it might be impractical to start a test from the command line. Implementing the interface defined in `testape_init.h` will allow you to decide the various options compile time and to execute other testcases besides testmain.

5.3 External dependencies

The use of the operating system and standard c library is collected via a jump table defined in `testape_ext.h`. You can modify this and route the use of these functions to your own implementation or to the default implementation defined by the os that you want in use.

The sample directory contains all these headers as well as a sample implementation for all of the above.

5.4 Other operating systems and bare metal configurations

TestApe do not ship with a bare metal configuration, but it is possible to customize the use of the OS and standard C library by providing an alternative implementation, so that these dependencies can be replaced with handwritten equivalents.

All use of c lib functions are collected in `linux_testape_ext.o` and `testape_messsages.o`. These files can be removed from `testape.a` and replaced with your handcrafted equivalent, like it is shown below

```
ar d testape.a linux_testape_ext.o
ar d testape.a testape_messsages.o
ar r testape.a my_os.o
```

You need to fill `my_os.c` with the implementations of the functions listed in `testape_ext.h` and `testape_customize.h`.

If write a vnsprintf replacement, you should know that testape internally uses the following format specifiers %s %d %f %.8x %.3d %02x %04x %08x and %c.

To remove all file operations and reduce the footprint of the initialization code, it may be wise to customize the startup and hardwire it to a specific testcase. Just remove `testape_init.o` from `testape.a` like this

```
ar d testape.a testape_init.o
```

and implement whatever init function you need (e.g. main) in your test source code. You can call the desired test function where appropriate. The example below shows `mytest` being called from main, but you can call `testmain` or any other test from anywhere in your implementation.

```
1 void main(void) {
2    EXECUTE(mytest);
3 }
```

You also need to control the behavior normally determined by the commandline options. You do that by providing the implementation listed in `testape_init.h`.

As the majority of the libc usage comes from the default init implementation a handcrafted main function in every test excutable will reduce requirements to a tiny c library to atoi, free, malloc, memcmp, memcpy, memset, strcat, strcmp, strcpy, strlen, strstr and vsnprintf

The package contains the sample `my_os_sample.c` the contains a sample implementation for all of the above. The commandline to use it and use to build sample26 looks like this

```
cp testape.a my_os_testape.a
gcc -c my_os_sample.c
ar d my_os_testape.a linux_testape_ext.o
ar d my_os_testape.a testape_messsages.o
ar d my_os_testape.a testape_init.o
ar r my_os_testape.a my_os_sample.o
testape gcc convert.c sample26.c my_os_testape.a
```

Chapter 6

Reference

6.1 ALLOW

This macro will add a function to the list of functions, for to which calls are allowed and always considered valid.

Syntax: ALLOW(function [, retval])

Parameters:

function A function used by the unit. The function can have any prototype.

retval An optional value that will be returned to the unit. Default is 0.

Returns: -

All function calls to *function* are considered valid for the duration of the test. If *function* are not present in unit the default mock will be called. The default mock will return the optional parameter *retval*. If this parameter is not given, the default mock will return 0.

ALLOW can be used together with the EXPECT macros. If these are used with same function in the same test (or in a test executed from the test), the validation of function calls setup with EXPECT will take precedence until all of the expected function calls has been validated. The macro is valid for the duration of the test. If the test calls other tests, the macro will remain in scope also for these tests. ALLOW on *function* will override any earlier allow macros on that function.

6.2 ALLOW_MOCK

This macro will add a function to the list of functions, to which calls are allowed and always considered valid. Function calls to this function are always replaced with function calls to a specfified mock function.

Syntax: ALLOW_MOCK(function, mock [, parameter])

Parameters:

function A function used by the unit. The function can have any prototype.

mock The function that will be called instead of *function*. The mock must have same prototype as the function it mocks.

parameter An optional value that can be retieved in the mock. Default is 0.

Returns: -

All function calls to *function* are considered valid, and they will be redirected to *mock* for the duration of the test. *parameter* is an optional parameter for the mock function. If not given, the default value is zero.

The mock will typical implement validation of the parameters, but it can also execute tests.

In order for the mock to validate the parameters, it is important that the mock has the same function prototype as the function it replaces. The return value of *mock* is passed back to the unit as the return value of *function*. The unit is unaware, that it is actually calling a mock function.

While *mock* is executing it is possible for the mock function to call *function*. Function calls made from an active mock do not apply to any of the rulessetup by by ALLOW or EXPECT. If *function* are not present in unit the default mock will be called. The default mock will return 0.

ALLOW_MOCK can be used together with EXPECT macros. If these are used with same function in the same test (or in a test executed from the test), the validation of function calls, setup with these macros, will take precedence until all of the expected function calls has been validated. ALLOW_MOCK is valid for the duration of the test. If the test calls other tests, the macro will remain in scope also for these tests.

ALLOW_MOCK on *function* will override any earlier allow macros on that function.

6.3 COMMENT

This macro will generate output to the log file.

Syntax: COMMENT(format, ...)

Parameters:

format printf style comment format string.

... printf style list of arguments.

Returns: -

Will put a prefixed comment in the logfile. The prefix defaults to `TESTAPE:` but it can be omitted or changed from commandline when the test executable is running.

6.4 EXECUTE

Executes a test.

Syntax: EXECUTE(test)

Parameters:

test A function implementing the test. *test* is a any function returning void.

... Parameters to the test. *test* is any function returning void.

Returns: -

test is a plain c function that implements the test. In its most simple form, this function will call some functionality in the unit that needs to be tested. In addition, the return value can be validated and/or a list of expected function calls can be setup.

The macro is used to start execution of a new test. It is used to execute the first testcase from the main function. It can also be nested and used during a test, or be called from a mock - i.e. during execution of the unit. Note that during execution of parameterized tests this macro might call *test* several times.

If *test* are declared with function arguments, these can be passed from the EXECUTE macro, after the *function*

6.5 EXPECT

This macro will add a function to the list of expected function calls.

Syntax: EXPECT(function [, retval])

Parameters:

function The function that is expected to be called during a test

retval An optional value that will be returned to the unit. Default is 0.

Returns: -

A test will execute the unit by calling one of its functions. In return, the unit will run its code, which in turn might call external functions outside the unit. For each test, a unique set of functions will be called. The test must prepare a list of these to the framework, before executing the unit. The test can use the macro EXPECT to instruct the framework that a function call to *function* is expected during the test. If several functions are called, several EXPECT macros must be used before executing the test. The order is important - e.g. the order of the EXPECT should correspond to the order, in which, the unit will call the expected functions. If the framework detects a function call to a function that is not expected at that time, the test will fail. Each time an expected function is correctly called, the framework will remove it from the list. The test will fail, if the list is not empty after the test has completed execution.

The unit is unaware, that it is actually calling a mock function. The default mock will return the optional parameter *retval*. If this parameter is not given, the default mock will return zero.

The macro is normally used before executing the unit, but if desired, it can also be called from mocks - i.e. during execution of the unit.

6.6 EXPECT_MOCK

This macro will add a function to the list of expected function calls. If the function is called at the expected time, the provided mock function will be called instead.

Syntax: EXPECT_MOCK(function, mock [, parameter])

Parameters:

function The function that should be called

mock The mock that will validate parameters and determine return value

parameter An optional value that can be retieved in the mock. Default is 0.

Returns: -

The EXPECT_MOCK macro is used to instruct the framework that a function call to *function* is expected during the test. If the unit calls *function* at the right time, the framework will replace the function call with a function call to the *mock* function.

parameter is an optional parameter for the mock function. If not given, the default value is zero. The parameter can be read in the mock.

The mock will typical implement validation of the parameters, but it can also execute tests.

In order for the mock to validate the parameters, it is important that the mock has the same function prototype as the function it replaces. The return value of *mock* is passed back to the unit as the return value of *function*. The unit is unaware, that it is actually calling a mock function.

6.7 MOCK

Enables or disables mocking of a function.

Syntax: MOCK(function, mock)

Parameters:

function The name of the function that is mocked

mock The function that will be used as mock (or 0)

Returns: -

The MOCK macro will instruct the framework to replace all function calls to *function* with function calls to *mock*. Unlike the ALLOW_MOCK and EXPECT_MOCK macros, the mock is in effect throughout execution of all tests, or until the mocking of the function are deactivated.

During execution of the mock function all mocking of *function* are disabled, so it is possible for the mock function to call *function*.

The mock can be replaced by subsequent calls to MOCK or deactivated if the function is mocked by 0. Only the last mock setup for *function* is active.

If *function* are not present in unit the default mock will be called. The default mock will return 0.

6.8 MOCK_VALUE

Returns the integer mock parameter

Syntax: MOCK_VALUE

Parameters: -

Returns: The integer parameter assigned to the mock by the currently executing test.

The MOCK_VALUE returns the mock parameter assigned to the mock function by the currently executing test. It is possible to cast this value to any type. When casting to a floating point, use the macro MOCK_VALUE_FP to avoid losing the fractional part.

The test will assign this value using the optional parameter to the macros ALLOW or EXPECT. If this is a floating point value, this macro will return its integer equivalent.

If the macro are used in a mock, that was setup using the EXPECT_MOCK macro wihtout the optional parameter, it will return zero.

All simulation values are stored as floating point numbers. In order to remain backward compatible with earlier versions of testape, this macro is implemented as an int wrapper for the default floating point version, e.g. `(int)MOCK_VALUE_FP`.

The value of this macro is undefined, unless it is called within a mock.

6.9 MOCK_VALUE_FP

Returns the floating point mock parameter

Syntax: MOCK_VALUE_FP

Parameters: -

Returns: The floating point parameter assigned to the mock by the currently executing test.

The MOCK_VALUE_FP returns the mock parameter assigned to the mock function by the currently executing test. It is possible to cast this value to any type, however, when casting to a pointer type, use the macro MOCK_VALUE to avoid compilation errors.

The test will assign this value using the macros ALLOW or EXPECT. If this is an integer value, this macro will return its floating point equivalent.

If the macro are used in a mock, that was setup using the EXPECT_MOCK macro, it will return zero (0.0f).

The value of this macro is undefined, unless it is called within a mock.

6.10 PARAMETER

Adds a variant to the next executed test.

Syntax: PARAMETER(value)

Parameters:

value A test value for a variant of a test.

Returns: -

The same test can be executed with a range of values. This is useful when increased coverage only require slight changes to a test. The variations can be added, to the next test, using this macro. Each consequent call to PARAMETER will add another value to be tested during next test. The framework will automatically repeat the next EXECUTE command for each value that has been added.

During execution the test can retrieve and implement the current variation using the macro PARAMETER_VALUE.

Usually this macro is used during setup of the test, e.g. before the EXECUTE macro. However, it is possible to add variant to the next test from everywhere.

6.11 PARAMETER_RANGE

Adds a range of variants to the next executed test.

Syntax: PARAMETER_RANGE(start, stop)

Parameters:

start First test value for a variant of a test.

stop Last test value for a variant of a test.

Returns: -

With the previous macro, it was possible to add a single variant value to a test. With the macro PARAMETER_RANGE, the same thing can be done with a range of values. Each consequent call to PARAMETER_RANGE will add a range, which will be tested during next test. The framework will automatically repeat the next EXECUTE command for each value in the range.

During execution the test can retrieve and implement the current variation using the macro PARAMETER_VALUE.

Usually this macro is used during setup of the test, e.g. before the EXECUTE macro. However, it is possible to add variant to the next test from everywhere.

6.12 PARAMETER_RANGE_FP

Adds a range of floating point variants to the next executed test.

Syntax: PARAMETER_RANGE_FP(start, stop, delta)

Parameters:

start First floating point test value for a variant of a test.

stop Last floating point test value for a variant of a test.

delta The value to increment/decrement for each call.

Returns: -

With the macro PARAMETER_RANGE_FP, a range of floating point values can be setup for the next test. Each consequent call to PARAMETER_RANGE_FP will add a range, which will be tested during next test. The framework will automatically repeat the next EXECUTE command for each value in the range.

During execution the test can retrieve and implement the current variation using the macro *PARAMETER_VALUE_FP*.

Usually this macro is used during setup of the test, e.g. before the EXECUTE macro. However, it is possible to add variant to the next test from everywhere.

6.13 PARAMETER_SET

Adds a one or more sets containing values for the next test.

Syntax: PARAMETER_SET(arr)

Parameters:

arr Array holding one or more sets

Returns: -

It is possible to vary the test by giving more than one value at a time. To do this, use the macro `PARAMETER_SET`. This will add one or more set of values. This macro can be used many times while preparing the test. Each consequent call to `PARAMETER_SET` will add one or more set of values, and each of those sets will in turn be tested during next test. The test will implement the variant using the macro `PARAMETER_VALUE`. The individual members of the set can be accessed from the test, by casting `PARAMETER_VALUE` to a pointer to one of the sets, e.g. `((set_type*)PARAMETER_VALUE)->set_member`. Usually this macro is used during setup of the test, e.g. before the `EXECUTE` macro. However, it is possible to add sets to next test from everywhere.

6.14 PARAMETER_VALUE

If the same test are executed with a range or a list, this value represents the variant currently executing.

Syntax: PARAMETER_VALUE

Parameters: -

Returns: The value assigned to the currently executing test variant.

All tests can be executed more than once and the tests can be varied slightly between each run. For example, one test could be setup to execute 100 times using the value from 0 to 99. Another test could be setup to execute with variants ‘""‘, ‘"string"‘ and ‘NULL‘.

The macro PARAMETER_VALUE will in turn hold each of these values. The test can read this macro, setup the list of expected function calls (as well as their simulated values), and stimulate the unit according to this value.

The macro is normally used during the test, but if desired, it can also be called from mocks - i.e. during execution of the unit.

6.15 PARAMETER_VALUE_FP

If the same test are executed with a range or a list of floating point values, this value represents the variant currently executing.

Syntax: PARAMETER_VALUE_FP

Parameters: -

Returns: The floating point value assigned to the currently executing test variant.

All tests can be executed more than once and the tests can be varied slightly between each run. For example, one test could be setup to execute 9 times using the value from 0.2 to 9.2.

The macro PARAMETER_VALUE_FP will in turn hold each of these values. The test can read this macro, setup the list of expected function calls (as well as their simulated values), and stimulate the unit according to this value.

The macro is normally used during the test, but if desired, it can also be called from mocks - i.e. during execution of the unit.

6.16 VALIDATE

Validates a variable against a reference value.

Syntax: VALIDATE(actual, expected)

Parameters:

actual The variable that will be checked

expected The reference value

Returns: -

This macro is used to validate a single value. The macro can be used within tests or mocks. Parameter *actual* must be a variable. Parameter *expected* can be a variable or a constant. The symbolic names of *actual* and *expected* are shown in the log. Proper chosen names for these can improve readability a lot.

6.17 VALIDATE_BIT

Validates a variable against a bit value.

Syntax: VALIDATE_BIT(actual, expected, bitno[, expected, bitno] ...)

Parameters:

actual The variable that will be checked

expected The reference value, 0 or 1

bitno The bit number 0-31

... Additional pairs of expected values and bit number

Returns: -

6.18 VALIDATE_BYTE

Validates a variable against a byte value.

Syntax: VALIDATE_BYTE(actual, expected [, mask])

Parameters:

actual The variable that will be checked

expected The reference value, 0-0xff

mask Optional mask value to apply before comparing value

Returns: -

6.19 VALIDATE_DWORD

Validates a variable against a 32 bit double word value.

Syntax: VALIDATE_DWORD(actual, expected [, mask])

Parameters:

actual The variable that will be checked

expected The reference value, 0-0xffffffff

mask Optional mask value to apply before comparing value

Returns: -

6.20 VALIDATE_FP

Validates a floating point variable against a reference value.

Syntax: VALIDATE_FP(actual, expected)

Parameters:

actual The floating point variable that will be checked

expected The floating point reference value

Returns: -

This macro is used to validate a single floating point value. Floating point values may have inaccuracies due to limitation in their precision. TestApe will call the function `testape_validate_fp_accuracy()` to obtain the required precision. The test can supply its own version of this function or rely on the default provided by the instrumenter. The default will return 0.00001f.

The macro can be used within tests or mocks. Parameter *actual* must be a floating point variable or an integer variable that can be cast to a floating point. Parameter *expected* can be a variable or a constant. The symbolic names of *actual* and *expected* are shown in the log. Proper chosen names for these can improve readability a lot.

6.21 VALIDATE_FILE

Validates the existence and contents of a file against a reference file.

Syntax: VALIDATE_FILE(actual, expected)

Parameters:

actual The filename that will be checked

expected String containing the expected file contents

Returns: -

If the unit is supposed to generate a file during the test, you can use the VALIDATE_FILE macro to validate the existence and contents of this file. The framework will verify the existence of a file name *actual*. If the right file is found, its contents are validated against the string *expected*. Use '
n' in string *expected* to insert linefeeds. These will be translated the same way as
n in the file generated by the unit.

This macro is not intended to be used with binary files. In order to validate contents of binary files, a validate function, that validates the content in the file byte for byte, must be written.

6.22 VALIDATE_MEMORY

Validates a given chunk of memory against the given reference data.

Syntax: VALIDATE_MEMORY(actual, expected, size)

Parameters:

actual A pointer to memory that will be checked

expected A pointer to memory containing reference data

size Size of memory block

Returns: -

To validate memory contents this macro can be used. Parameter *actual* and *expected* must be variables. The symbolic names of *actual* and *expected* are shown in the log. Proper chosen names for these can improve readability a lot. The framework will compare the content 8 byte at a time. For each failed 8-byte block, the framework will display both the expected and actual 8-byte block in the log file. Exceptions are cought and reported, in case some of the memory are out of bounds.

6.23 VALIDATE_STRUCT

Validates the contents of struct against a reference struct.

Syntax: VALIDATE_STRUCT(actual, expected)

Parameters:

actual The struct data that will be checked

expected The reference struct data

Returns: -

With this macro, more data can be validated. The framework will check that the two structs are of identical size and that their content are identical. The function also check the padding bytes. You must use memset, before assigning values to the structs, in order to control the values of padding bytes.

6.24 VALIDATE_STRING

Validates the contents of a string against a reference string.

Syntax: VALIDATE_STRING(actual, expected)

Parameters:

actual The string data that will be checked

expected The reference string

Returns: -

To validate a zero terminated string, this macro can be used. If the strings have different lengths, the test will fail. Parameter *actual* must be a variable. Parameter *expected* can be a variable or a constant. The symbolic names of *actual* and *expected* are shown in the log. Proper chosen names for these can improve readability a lot. The framework will compare the content 8 byte at a time. For each failed 8-byte block, the framework will display both the expected and actual 8-byte block in the log file. Exceptions are caught and reported, in case some of the memory access is not allowed.

6.25 VALIDATE_WORD

Validates a variable against a 16 bit word value.

Syntax: VALIDATE_WORD(actual, expected [, mask])

Parameters:

actual The variable that will be checked

expected The reference value, 0-0xffff

mask Optional mask value to apply before comparing value

Returns: -

6.26 Deprecated macros

ALLOW_VALIDATE
This macro now replaced by ALLOW_MOCK. Parameters and functionality are the same.

ALLOW_SIMULATE
This macro now replaced by an optional parameter to ALLOW.

EXPECT_VALIDATE
This macro now replaced by EXPECT_MOCK. Parameters and functionality are the same.

EXPECT_SIMULATE
This macro now replaced by an optional parameter to EXPECT_MOCK.

EXPECT_AND_VALIDATE
This macro now replaced by EXPECT_MOCK. Parameters and functionality are the same.

EXECUTE_ADD
This macro now replaced by PARAMETER. Parameters and functionality are the same.

EXECUTE_ADD_RANGE
This macro now replaced by PARAMETER_RANGE. Parameters and functionality are the same.

EXECUTE_ADD_SET
This macro now replaced by PARAMETER_SET. Parameters and functionality are the same.

EXECUTE_VALUE
: This macro now replaced by PARAMETER_VALUE. Parameters and functionality are the same.

SIMULATE
: This macro now replaced by an optional parameter to EXPECT.

SIMULATE_VALUE
: This macro now replaced by MOCK_VALUE. Parameters and functionality are the same.

SIMULATE_VALUE_FP
: This macro now replaced by MOCK_VALUE_FP. Parameters and functionality are the same.