TestApe

Unit Testing for embedded software

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

Using the TestApe unit testing tool

Release 791

Contents

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, CygWin GCC or Microsoft compilers. It will run in both Windows and Linux operating systems. 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 manualy, the framework will automatically create a default TestApe mock.

- Mock

  A mock is a function the replaces or extends existing functionallity 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 framed and in `italic`.

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

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 and in black.

```
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 tests uses the macros available from the testape framework. These are defined in `testape.h`.

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.

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

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

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

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

See also Section 4 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()` and `subtract()`. The calculator program is normaly assembled of `calc.c`, `add.c`, `subtract.c`, `multiply.c` and `divide.c`, but `calc.c` is the only unit being tested in the following samples.

```
int validate(int a, int b)  { return 1; }

int calculate(int operand1, int operand2, char operation)
  {
  if ( invalid(operand1) || invalid(operand2) )
    return ERROR;

  switch(operation)
    {
    case '+':
      return add(operand1, operand2);
    case '-':
      return subtract(operand1, operand2);
    case '*':
      return multiply(a,b);
    case '/':
```

```
      return divide(a,b);
    }
  return ERROR;
 }
```

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 CygWIN/GCC environment like it is shown below. Notice that `add.c`, `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 sample1.c testape.a

TestApe instrumentation tool, release beta-Mar--7-2010-linux.
Unit testing for embedded software - http://testape.com


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

 Analyzing ...
   Generating default mocks -> add subtract multiply divide
```

[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'.

```
  Generating hooks ->  None

c:\>test_calculator
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.

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`.

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

If the validation succeeds the following output are generated

```
Executing test testmain

  Executing test test_addition

  PASSED verify result
    expected .......... 10 (10)
    actual ............ 10

  PASSED test test_addition

PASSED test testmain
```

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.

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

The framework includes additional macros for validating strings, arrays, structs and file contents. Detailed description can be found in chapter 5 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.

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

will change the log output to

```
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]

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

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

```
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 actual and expected function call resolves to a function that 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 expected function call resolves to a function that is present in the unit, the framework will simply report that it is called and afterwards call the function.

2.6 Simulating return values

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

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

As sample5 shows above[1], the `SIMULATE` 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 resolves to a function that 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 resolves to a function that 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 function resolves to a function that is present in the unit, the framework will not interfere. The unit will then simply call that function.

2.7 Validating parameters to functions

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

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

The EXPECT_VALIDATE 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.

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

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

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

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

2.8 Simulating output from mock functions, part one

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 return 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 give the return value by the use of the macro `SIMULATE`, if the default value of zero used by `EXPECT` is not adequate.

There is also an equivalent EXPECT_SIMULATE macro, that is used by the test to indicate that a mock function should be used to validate the parameters.

The simulate value given to EXPECT_SIMULATE can be read from within the mock function using the macro SIMULATE_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_SIMULATE ( 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 SIMULATE_VALUE;
  }
```

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

2.10 Mocking without validating 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.

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

Functions that are present in the unit and not listed in a EXPECT, SIMULATE, EXPECT_VALIDATE or EXPECT_VALIDATE is 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 resolves to a function that is not present in the unit, the framework will return 0 to the unit whenever the mocked function is called.

With the use of ALLOW_SIMULATE 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.

```
void test_addition(void)
  {
  ALLOW_SIMULATE ( invalid, TRUE );
  VALIDATE(calculate(3,7,'+'), ERROR);
  }
```

It is also possible to replace the default mock entirely for the duration of the test. By the use of ALLOW_VALIDATE 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.

```
int mock_invalid(int parameter)
  {
  return FALSE;
  }

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

 void test_addition(void)
   {
   ALLOW_VALIDATE  ( invalid, mock_invalid );
   EXPECT_SIMULATE ( add, check_add , EXPECTED_SUM);
   VALIDATE(calculate(3,7,'+'), EXPECTED_SUM);
   }
```

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 - typicaly 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.

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

void test_printf(void)
  {
  SIMULATE(invalid, TRUE);
  ALLOW_VALIDATE( printf, mock_printf);
  calculate(3,7,'+');
  }
```

2.12 Calling mocked functions from a 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]

```
void mock_printf(char *fmt, ...)
  {
  va_list args;
  static int count = 0;
  printf("*** printf called %d times *** \n", count++);
  va_start(args,fmt);
  vprintf(fmt, args);
  }

void test_printf(void)
  {
  SIMULATE(invalid, TRUE);
  ALLOW_VALIDATE  ( printf, mock_printf);
  calculate(3,7,'+');
  }
```

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

```
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 possibility 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

```
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

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 );
  SIMULATE  ( invalid, PARAMETER_VALUE );

  if (!PARAMETER_VALUE)
    EXPECT_VALIDATE ( 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_SIMULATE ( 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*)SIMULATE_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_SIMULATE ( add, check_add , PARAMETER_VALUE);
  VALIDATE(calculate(tmp->left,tmp->right,'+'), tmp->result);
  }

void testmain(void)
  {
  mytest_t  test_calc_add_paramters[] = {{1,1,2},{3,4,7},{6,6,12}};
```

```
mytest_t  test5_5_10[] = {5,5,10};


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

2.17 Signal handling

TestApe framework will catch and report illegal memory access. If the unit attempts to access memory at an illegal adresses, the framework will abort the test. The log output will contain a message like this

```
Executing test testmain

  Executing test test_division

    Expecting function call to divide

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

    FAILED ( crashed with exception )

    FAILED verify calculate(3,0,'/')
      expected ......... 0 (0)
      actual ........... 7153560

  FAILED test test_division (2 errors)
    test exception
    Verify integer

FAILED test testmain (1 error)
  Execution of test
```

and the test will fail. TestApe will try to resume test execution. sample19 in sample directory contains a division-by-zero example.[1]

2.18 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.

```
void test_addition(void)
  {
  SIMULATE(add,10);
  VALIDATE(calculate(3,7,'+'), 10);
  }
```

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

2.19 Using the framework without the instrumenter

The instrumenter will help assemble the test executable and it will fill in missing functionality from the object files that are not present in the unit. If you try to link without the instrumenter you will get unresolved externals. Normally the instrumenter will fill in mock functions for these, but without it you will have to do it manually. The framework allows you to write very simple default mocks for these using the macro `CREATE_MOCK`. For example

```
CREATE_MOCK(add);
CREATE_MOC(invalid);
```

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.

```
 -------------
------ 000 ----
        ++
    ,,, ++
   (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] ld [linker arguments] testape.a
testape [-o] gcc [compiler parameters] testape.a
testape [-o] LINK [linker arguments] testape.lib
testape [-o] 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 put it all together.

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

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

3.2 Using TestApe instrumenter with GCC

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. To install TestApe on a Debian based system run

```
dpkg -i testape_r556_i386.deb
```

On a Redhat based system run

```
rpm -i testape-r556-2.i386.rpm
```

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

```
libstdc++.so.6
libm.so.6
libgcc_s.so.1
libc.so.6
```

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++ see the forum at http://testape.com.

3.3 Using TestApe instrumenter 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
```

3.4 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.

- `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.

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 log will contain no error and test identifiers. This default behavior can be changed by giving one or more of the following options :

```
testexe [-q | --quiet]
        [-idt | --show-test-identifiers]
        [-ide | --show-error-identifiers]
        [-h | --help]

testexe [-i identifier | --ignore identifier]

testexe [identifier | -e identifier | --execute identifier]
```

The name of a test is 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
```

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

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
```

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

```
void testape_error_handler(void)
  {
  }
```

Conditional breakpoints can be set using on one of the predefined macros, `testape_current_test`, `testape_current_test_id`, `testape_current_error`, or `testape_current_error_id` as shown below

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

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

During execution TestApe will log test progress to the screen. 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.

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

```
          testape: TestApe test executable
          testape: Unit testing for embedded software - http://testape.com
          testape:
          testape: Test executable initializing
          testape:   -ide ( error identifiers shown in log )
          testape:   -idt ( test identifiers shown in log )
          testape:
t000 e000 testape:
t000 e000 testape: Executing test testmain
t000 e000 testape:
t001 e000 testape:
t001 e000 testape:   Executing test test_addition
t001 e000 testape:
t001 e000 testape:
t001 e000 testape:   PASSED test test_addition
```

```
t000 e000 testape:
t000 e000 testape: PASSED test testmain
          testape:
          testape:
          testape: Test exeutable terminating
          testape:   exitcode 0
```

When an error is detected TestApe will log information about the error. Generally 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

Reference

5.1 ALLOW

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

Syntax: ALLOW(function)

Parameters:

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

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 0.

ALLOW can be used together with EXPECT, EXPECT_VALIDATE or SIMULATE. 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, EXPECT_VALIDATE or SIMULATE 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.

5.2 ALLOW_SIMULATE

This macro will add a function to the list of functions, for which function calls are allowed and always considered valid. Whenever the function is called a fixed value are returned to the unit.

Syntax: ALLOW_SIMULATE(function, retval)

Parameters:

> ***function*** A function used by the unit. The function can have any prototype.
>
> ***retval*** The value that will be returned to the unit whenever *function* is called

Returns: -

All function calls to *function* are considered valid and will return *retval* for the duration of the test. ALLOW_SIMULATE can be used together with EXPECT, EXPECT_VALIDATE or SIMULATE. 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, EXPECT_VALIDATE or SIMULATE 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_SIMULATE on *function* will override any earlier allow macros on that function.

5.3 ALLOW_VALIDATE

This macro will add a function to the list of functions, for which function 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_VALIDATE(function, mock)

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.

Returns: -

All function calls to *function* are considered valid, and they will be redirected to *mock* for the duration of the test.

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 ALLOW, ALLOW_SIMULATE, ALLOW_VALIDATE, EXPECT, EXPECT_VALIDATE or SIMULATE. If *function* are not present in unit the default mock will be called. The default mock will return 0.

ALLOW_VALIDATE can be used together with EXPECT, EXPECT_VALIDATE or SIMULATE. 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_VALIDATE 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_SIMULATE on *function* will override any earlier allow macros on that function.

5.4 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.

5.5 EXECUTE

Executes a test.

Syntax: EXECUTE(test)

Parameters:

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

5.6 EXPECT

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

Syntax: EXPECT(function)

Parameters:

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

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 mock function will as default return zero, but several other options are available - See the description of the EXPECT or SIMULATE macros. 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.

5.7 EXPECT_VALIDATE

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_VALIDATE(function, mock)

Parameters:

function The function that should be called

mock The mock that will validate parameters and determine return value

Returns: -

The EXPECT_VALIDATE macro is used to instruct the framework that a function call to *function* is expected during the test. The unit is unaware, that it is actually calling a mock function. If the unit calls the function at the right time, the framework will pass control to *mock* function. 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. Upon return from the mock, the return value is passed back to the unit.

5.8 EXPECT_SIMULATE

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_SIMULATE(function, mock, retval)

Parameters:

function The function that should be called

mock The mock that will validate parameters and determine return value

retval The value that is proposed to be returned to the unit. The value is available to the mock using the macro SIMULATE_VALUE

Returns: -

The EXPECT_SIMULATE macro is used to instruct the framework that a function call to *function* is expected during the test. The unit is unaware, that it is actually calling a mock function. If the unit calls the function at the right time, the framework will pass control to the *mock* function. The value *retval* can be read in the mock using the macro SIMULATE_VALUE. The mock will typical implement validation of the parameters, perhaps guided by the value of \ retval, but it can also execute tests.

In order for the mock to validate the parameters, it is important that the mock have the same function prototype as the function it replaces.

The value *retval* can be read in the mock using the macro SIMULATE_VALUE, and may be used as guidance on what to return. Upon return from the mock, the return value is passed back to the unit.

5.9 MOCK

Generates a default mock.

Syntax: MOCK(name)

Parameters:

name The name of the function that is mocked

Returns: -

The MOCK macro will generate a mock function called *name*. The mock function replaces and simulates functions, that are called by the unit but not available during the test. It reports to the framework, whenever a function call to *name* is detected. This allows the framework to validate, that the parameters are correct, and that the function call occurs at the right time.

The prototype for a mock function can be a function prototype of any kind. The MOCK macro will generate functions taking no parameters and returning nothing, e.g. void *name(void)*. The function it replaces will most likely have another prototype, but the differences are not important to the test.

The instrumenter will normally generate default mock functions for you, but this macro is available, if you are not using the instrumenter.

This macro cannot be used together with earlier prototype declarations of *name*. If you experience compilation problems due to this, there are two solutions - either use the correct prototype (for the test it does not matter), or use the MOCK macro from within a file, that does not include the real function declaration.

5.10 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.

5.11 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.

5.12 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.

5.13 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.

5.14 SIMULATE

This macro will add a function to the list of expected function calls. If the function is called at the expected time, the framework will return the provided value to the unit.

Syntax: SIMULATE(function, retval)

Parameters:

> ***function*** The function that should be called
>
> ***retval*** The value that will be returned to the unit

Returns: -

The SIMULATE macro is used to instruct the framework that a function call to *function* is expected during the test. The unit is unaware, that it is actually calling a mock function. If the unit calls the function at the right time, the framework will use the value *retval* and return it back to the unit. 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 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.

5.15 SIMULATE_VALUE

The return value proposed to the mock by the test.

Syntax: SIMULATE_VALUE

Parameters: -

Returns: The simulation value propose to the mock by the currently executing test.

The SIMULATE_VALUE holds the value assigned to the mock function by the currently executing test. The test uses the macro EXPECT_SIMULATE to assign this value. The value can be read from anywhere, but it is undefined, unless it is read from within a mock. If the mock are called in a mock, that was setup using the EXPECT_VALIDATE macro, the macro SIMULATE_VALUE will be zero. The value of this macro is undefined, unless it is called within a mock.

5.16 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.

5.17 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.

5.18 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.

5.19 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.

5.20 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.

5.21 Deprecated macros

`STUB` This macro now replaced by `MOCK`. Parameters and functionality are the same.

`EXPECT_AND_VALIDATE` This macro now replaced by `EXPECT_VALIDATE`. 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.