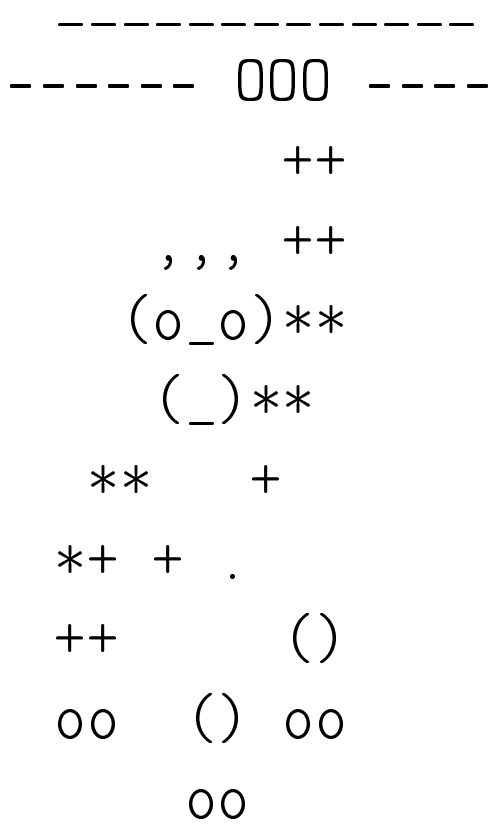

Contents

1 TestApe **2**

1.1 Theory of usage 2

1.2 The author 3

1.3 Terms of use 3

2 Using the TestApe library **5**

2.1 Creating and executing a basic testcase 5

2.2 Validating output 6

2.3 Validating function calls 6

2.4 Simulating return values 7

2.5 Validating parameters to functions 8

2.6 Simulating output from validator functions, part one 9

2.7 Simulating output from validator function, part two 10

2.8 Organizing tests 10

2.9 Ranged tests with parameters 12

2.10 Validating ranged tests with parameters 13

2.11 Using the framework without the instrumenter 14

3 Using the TestApe instrumenter **15**

3.1 Using TestApe instrumenter with GCC 15

3.2 Using TestApe instrumenter with Visual Studio 16

3.3 Integrating with Linux/GCC 16

3.4 Integrating with Visual Studio 8.0 17

3.5 Integrating with Visual Studio 6.0 17

4 Invoking the test executable **18**

5 Coverage and Memory analysis **19**

5.1 Memory analysis 19

5.2 Coverage analysis 19

6 BREW extension **20**

7 Reference **21**

7.1 STUB(name) 23

7.2 VALIDATE_FILE(actual, expected) 23

7.3 VALIDATE_STRUCT(actual, expected) 24

7.4 VALIDATE_MEMORY(actual, expected, size) 24

7.5 VALIDATE_STRING(actual, expected) 24

7.6 VALIDATE(actual, expected) 25

7.7 EXPECT(function) 25

7.8 SIMULATE(function, retval) 25

7.9 EXPECT_VALIDATE(function, validator) 26

7.10 EXPECT_SIMULATE(function, validator, retval) 26

7.11 SIMULATE_VALUE 27

7.12 EXECUTE_VALUE 27

7.13 EXECUTE(test) 27

7.14 EXECUTE_ADD(value) 28

7.15 EXECUTE_ADD_RANGE(start, stop) 28

7.16 EXECUTE_ADD_SET(arr) 29

7.17 COMMENT 29

Chapter 1

TestApe

TestApe is a free unit-testing package that can be used to test C programs. It can be used like many other frameworks to make a function call and test the return value. However, its 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 powerful instrumenter for code generated with GCC or Microsoft compilers. It will run in both Windows and Linux operating systems. The instrumenter generates stubs in order to test and simulate the data flow between the unit that is tested and the stub 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 stubs 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, he invented the TestApe instrumenter. It meant that new unit test projects could be launched quickly. The principles governing the TestApe framework, as well as the freedom in structure it supports, is heavily inspired by all the the frameworks that the author have had his hands on during these years.

1.3 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 tests are plain C modules that are compiled and linked with the unit(s) being tested in order to generate the test executable. The tests use the test primitives available from the testape framework. The interface to the framework can be found in `testape.h`. The interface contains several macros that will make test cases more readable and easier to write.

2.1 Creating and executing a basic testcase

A full unit test 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 consists of one plain C function that will call one or more functions in the unit. In the sample below the test 'test_load_my_app' will call the function `my_brew_app_Load`.

```
struct _IModule *test_my_app;

void test_load_my_app(void)
    {
    my_brew_app_Load(&my_ishell,0,&test_my_app);
    }
```

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

```
  Executing test test_load_my_app
  PASSED test test_load_my_app
```

Even though there is not a lot of validation going on in this test, it will enable coverage analysis, debugging and memory checking of the code in `my_brew_app_Load`.

2.2 Validating output

To improve the test one or more validations can be added. The `VALIDATE` macro that is used below will validate the data returned from `my_brew_app_Load` against the expected value `CONSTRUCT_OK`.

```
struct _IModule *test_my_app;

void test_load_my_app(void)
    {
    int ret = my_brew_app_Load(&my_ishell,0,&test_my_app);
    VALIDATE(ret,CONSTRUCT_OK);
    }
```

If the data is validated the following output are generated

```
 PASSED verify ret
   expected .......... CONSTRUCT_OK (1)
   actual ............ 1
```

If the data is not validated the log will look something like below and the test will be listed as failed in the summary at the end of the report.

```
 FAILED verify ret
   expected .......... CONSTRUCT_OK (1)
   actual ............ 0
```

The framework includes additional macros for validating strings, arrays, structs and file contents.

2.3 Validating function calls

Most likely `my_brew_app_Load` will depend upon other functions in order to implement it's behavior. For a given test, there is a certain order in which these are executed. This is tested using the macro `EXPECT` as shown below.

```
struct _IModule *test_my_app;

void test_load_my_app(void)
    {
    int ret;

    EXPECT( AEEStaticMod_New);
    EXPECT( AEEApplet_New);
    EXPECT( IShell_QueryClass );
```

```
    ret = my_brew_app_Load(&my_ishell,0,&test_my_app);

    VALIDATE(ret,CONSTRUCT_OK);
    }
```

The `EXPECT` macro is used to indicate to the framework, that this test expects function calls to be made to `AEEStaticMod_New`, `AEEApplet_New` and `IShell_QueryClass`. The framework will detect whatever functions are called by `my_brew_app_Load` and if everything works as expected, it will generate the output shown below

```
  Executing test test_load_my_app

    Expecting function call to AEEStaticMod_New
    Expecting function call to AEEApplet_New
    Expecting function call to IShell_QueryClass

    In stub AEEStaticMod_New

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

    In stub AEEApplet_New

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

    In stub IShell_QueryClass

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

    PASSED verify ret
      expected .......... CONSTRUCT_OK (1)
      actual ............ 1

  PASSED test test_load_my_app
```

The unit does not see any difference compared to its normal environment. The value 0 is returned from every function call validated by the `EXPECT` macro.

2.4 Simulating return values

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

```
struct _IModule *test_my_app;
struct _AEEApplet my_applet;

void test_load_my_app(void)
    {
    int ret;

    EXPECT  ( AEEStaticMod_New);
    SIMULATE( AEEApplet_New, &my_applet);
    EXPECT  ( IShell_QueryClass );

    ret = my_brew_app_Load(&my_ishell,0,&test_my_app);

    VALIDATE(ret,CONSTRUCT_OK);
    }
```

As the sample shows above, the `SIMULATE` macro is used to indicate to the framework, that this test expects a series of function calls. The second function call must be be made to `AEEApplet_New`. If the function call is detected, a value of `&my_applet` will be returned to `my_brew_app_Load`, upon return from `AEEApplet_New`.

2.5 Validating parameters to functions

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

```
struct _IModule *test_my_app;

void test_load_my_app(void)
    {
    int ret;

    EXPECT_VALIDATE ( AEEStaticMod_New, check_AEEStaticMod_New);
    EXPECT          ( AEEApplet_New );
    EXPECT          ( IShell_QueryClass );

    ret = my_brew_app_Load(&my_ishell,0,&test_my_app);

    VALIDATE(ret,CONSTRUCT_OK);
    }
```

The `EXPECT_VALIDATE` macro shown above instructs the framework to expect a function call to `AEEStaticMod_New`. `check_AEEStaticMod_New` is a reference to a validator function that will verify the arguments being passed to `AEEStaticMod_New`. This function must have the same prototype as the function it simulates. The sample function `check_AEEStaticMod_New` shown below has the same prototype as the function `AEEStaticMod_New` called by `my_brew_app_load`.

```
int check_AEEStaticMod_New(struct _IShell *my_shell,
                           int par,
                           struct _IModule *my_app)
    {
    VALIDATE(my_shell,NULL);
    VALIDATE(par,0);
    VALIDATE(my_app,&test_my_app);
    }
```

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 validator function. The output from the validator will look something like this

```
PASSED verify my_shell
  expected ..... NULL (0)
  actual ..... 0

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

PASSED verify my_app
  expected ..... &test_my_app ( 0x12345678 )
  actual ..... 0x12345678
```

2.6 Simulating output from validator functions, part one

The validator function has the same prototype as the function is simulates, so it is possible for it to return a value that the framework will guide to back to `my_brew_app_load`.

This sample validator function shown below will simulate that `AEEStaticMod_New` returns `CONSTRUCT_OK`, so this is the value that the validator function will return. Other tests may require a different return value, so they will use another validator function.

```
int check_AEEStaticMod_New(struct _IShell *my_shell,
                           int par,
                           struct _IModule *my_app)
    {
    VALIDATE(my_shell,NULL);
    VALIDATE(par,0);
    VALIDATE(my_app,&test_my_app);
    return CONSTRUCT_OK;
    }
```

2.7 Simulating output from validator function, part two

As described in section 2.4 the test can give the return value by the use of the macro `SIMULATE`, if the default value of zero returned by `EXPECT` is not adequate.

There is also an equivalent `EXPECT_SIMULATE` macro, that is used by the test to give a validator function to validate the parameters as well as a return value. The value being returned is still determined by the validator function but the value proposed by the test can be read from within the function using the macro `SIMULATE_VALUE`.

```
struct _IModule *test_my_app;

void test_load_my_app(void)
    {
    int ret;

    EXPECT_SIMULATE( AEEStaticMod_New,
                          check_AEEStaticMod_New,
                          CONSTRUCT_FAIL);
    ret = my_brew_app_Load(&my_ishell,0,&test_my_app);
    VALIDATE(ret,CONSTRUCT_FAIL);
    }
```

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

```
int check_AEEStaticMod_New(struct _IShell *my_shell,
                          int par,
                          struct _IModule *my_app)
    {
    VALIDATE(my_shell,NULL);
    VALIDATE(par,0);
    VALIDATE(my_app,&test_my_app);
    return SIMULATE_VALUE;
    }
```

The same thing can be achived by having several different validator functions each returning their own unique value, but by the use of `SIMULATE_VALUE` the same validator function can be reused between several tests.

2.8 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_load_my_app

    Expecting function call to AEEStaticMod_New
    Expecting function call to AEEApplet_New
    Expecting function call to IShell_QueryClass

    In stub AEEStaticMod_New

    PASSED verify my_shell
      expected .......... NULL (0)
      actual ............ 0

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

    PASSED verify my_app
      expected .......... &test_my_app ( 0x12345678 )
      actual ............ 0x12345678

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

    In stub AEEApplet_New

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

    In stub IShell_QueryClass

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

    PASSED verify ret
      expected .......... CONSTRUCT_OK (1)
      actual ............ 1

  PASSED test test_load_my_app
```

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);
    }
```

2.9 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 validator functions. However - it is much easier to use the possibility to execute a ranged test with parameters using the macro `EXECUTE_ADD`, `EXECUTE_ADD_RANGE`, and `EXECUTE_ADD_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 `EXECUTE_ADD(true);` `EXECUTE_ADD(false);` and execute the test as using `EXECUTE(myfunc)`

Some more examples are shown in the the code below

```
void interesting_integers(void)
    {
    EXECUTE_ADD(0);
    EXECUTE_ADD(-1);
    EXECUTE_RANGE(128,255);
    EXECUTE_ADD("some input string");
    EXECUTE_ADD("");
    EXECUTE_ADD(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

2.10 Validating ranged tests with parameters

Each time an input parameter is changed a new test is executed. The TestApe will invoke all these tests one at a time, until the list of values is exhausted. With the previous example in mind, the macro `EXECUTE_VALUE` can be used to extract the values as shown below

```
void test_load_my_app(void)
    {
    int ret;

    EXPECT( AEEStaticMod_New);
    EXPECT( AEEApplet_New);
    EXPECT( IShell_QueryClass );

    ret = my_brew_app_Load(&my_ishell,EXECUTE_VALUE,&test_my_app);

    if (0 == EXECUTE_VALUE)
      VALIDATE(ret,CONSTRUCT_FAIL);
    else
      VALIDATE(ret,CONSTRUCT_OK);
    }
```

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 `EXECUTE_VALUE`. An example is shown below.

```
void test_load_my_app(void)
    {
    int ret;

    if (0 != EXECUTE_VALUE)
      {
      EXPECT( AEEStaticMod_New);
      EXPECT( AEEApplet_New);
```

```
    EXPECT( IShell_QueryClass );
    ret=my_brew_app_Load(&my_ishell,EXECUTE_VALUE,&test_my_app);
    VALIDATE(ret, CONSTRUCT_OK);
    }
  else
    {
    EXPECT( AEEStaticMod_New);
    ret=my_brew_app_Load(&my_ishell,EXECUTE_VALUE,&test_my_app);
    VALIDATE(ret, CONSTRUCT_FAIL);
    }
  }
```

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

```
void test_load_my_app(void)
  {
  int ret;

  EXPECT_SIMULATE( AEEStaticMod_New,
                   check_AEEStaticMod_New, EXECUTE_VALUE);
  EXPECT( AEEApplet_New);
  EXPECT( IShell_QueryClass );
  ret=my_brew_app_Load(&my_ishell,EXECUTE_VALUE,&test_my_app);
  VALIDATE(ret, CONSTRUCT_OK);
  }
```

The validator function can pick up the value as described in 2.4

2.11 Using the framework without the instrumenter

The instrumenter will help you 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 stub functions for these, but without it you will have to do it manually. The framework allows you to write very simple substitutes for these using the macro `CREATE_STUB`. For example

```
CREATE_STUB(AEEStaticMod_New);
CREATE_STUB(AEEApplet_New);
CREATE_STUB(IShell_QueryClass);
```

Chapter 3

Using the TestApe instrumenter

The TestApe instrumenter is designed to generate stub replacement code for functions that cannot be resolved by the linker. The generated code allows simulating and testing of data flowing between the external functions and the test executable.

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 stub generation will be initiated and the TestApe logo will be displayed.

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

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 Using TestApe instrumenter 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 g++ unit_a.cpp unit_d.cpp ad_test.c /path/testape.a
```

3.2 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
```

3.3 Integrating with Linux/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. On Debian based system run

```
dpkg -i testape_1.0_i386.deb
```

On a Redhat based system run

```
rpm -i testape-1.0-2.i386.rpm
```

If you want to do it manually, unpack the tar ball and put the instrumenter, interface and library to the proper directories. The library and instrumenter will depend of the following libraries

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

3.4 Integrating with Visual Studio 8.0

N/A

3.5 Integrating with Visual Studio 6.0

The compiler needs access to interface file and the linker needs access to the library file. The instructions below fits a Microsoft Visual Studio installation on a Windwos XP operating system.

- Add the library `testape.lib` to library directory. In MSVC 6.0 default is

  ```
  c:\Program␣files\Microsoft␣Visual␣Studio\vc98\lib
  ```

- Add interface `testape.h` to include directory. In MSVC 6.0 default is

  ```
  c:\Program␣files\Microsoft␣Visual␣Studio\vc98\include
  ```

If you plan to use the instrumenter you need to put it where your linker is located or somewhere else in the path of executables. The instrumenter will help you assemble the test executable and it will fill in missing functionality for those modules not included in your unit. Instructions for Microsoft Visual studio are given below

- Add `testape.exe` to Microsoft Visual Studio binary directory. In MSVC 6.0 default is

  ```
  c:\Program␣files\Microsoft␣Visual␣Studio\vc98\bin
  ```

- Make sure that `testape.exe` is invoked every time `link.exe` is called. In MSVC 6.0 this can be done by opening regedit and changing `'Executable Path'` from `link.exe` to `testape.exe link.exe` Entry `'Executable Path'` is located in

  ```
  \HKEY_CURRENT_USER\Software\Microsoft\Devstudio\6.0\
  ␣Build␣System\Components\Platforms\Win32␣(x86)\Tools\
  ␣COFF␣Linker␣for␣80x86
  ```

The instrumenter will stay transparent until it is invoked. Linking with testape.lib will automatically invoke the instrumenter and display "Instrumenting ..." in the build window. This method of instrumenting is well suited for running the tests from within visual studio.

Chapter 4

Invoking the test executable

The name of a test is used as identifier on the commandline. 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 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
```

```
testexe [-S | -stop-on-error ]
         [-B | --break-on-error ]
         [-L | --log-only-errors]

testexe [-I identifier | --ignore identifier]

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

Chapter 5

Coverage and Memory analysis

5.1 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`

5.2 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
```

Chapter 6

BREW extension

With respect to TestApe there is no difference between BREW applications and any other software program. It is easily possible to instrument and create TestApe test without the TestApe BREW extension. However - All BREW applications share several characteristics that makes it possible to reuse and share much of the test cases and test software. Also BREW applications requires certification. The certification test cases can be simulated with the TestApe BREW extension framework. The TestApe BREW extension available for download only after contacting the author.

Chapter 7

Reference

- **STUB**(name)

  Generates a stub function.

- **VALIDATE_FILE**(actual, expected)

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

- **VALIDATE_STRUCT**(actual, expected)

  Validates the contents of struct against a reference struct.

- **VALIDATE_MEMORY**(actual, expected, size)

  Validates size *byte of memory pointer to by* actual *against the reference data* expected.

- **VALIDATE_STRING**(actual, expected)

  Validates the contents of a string against a reference string.

- **VALIDATE**(actual, expected)

  Validates a variable against a reference value.

- **EXPECT**(function)

  Validates that a function call is made to function.

- **SIMULATE**(function, retval)

  Validates that a function call is made to function. *If function is called, value* retval *is returned to the unit.*

- **EXPECT_VALIDATE**(function, validator)

  Validates that a function call is made to function. *If function is called, function* validator *is called.*

- **EXPECT_SIMULATE**(function, validator, retval)

  Validates that a function call is made to function. *If function is called, function* validator *is called.*

- **SIMULATE_VALUE**

  The return value proposed to the stub function by the test.

- **EXECUTE_VALUE**

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

- **EXECUTE**(test)

  Executes a test.

- **EXECUTE_ADD**(value)

  Adds a variant to the next executed test.

- **EXECUTE_ADD_RANGE**(start, stop)

  Adds a range of variants to the next executed test.

- **EXECUTE_ADD_SET**(arr)

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

- **COMMENT**

  The command will generated a comment in the log file.

7.1 STUB(name)

Generates a stub function.

Parameters: ***name*** The name of the function that this stub intends to replace

The STUB macro will generate a stub function called *name*. The stub 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 stub function can be a function prototype of any kind. The STUB macro will generate stub 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 stub 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 STUB macro from within a file, that does not include the real function declaration.

7.2 VALIDATE_FILE(actual, expected)

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

Parameters: ***actual*** The filename that will be checked

expected String containing the expected file contents

If the unit is supposed to generate a file during the test, you can use the VALIDATE_FILE macroto 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 ’

’ in string *expected* to insert linefeeds. These will be translated the same way as

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.

7.3 VALIDATE_STRUCT(actual, expected)

Validates the contents of struct against a reference struct.

Parameters: ***actual*** The struct data that will be checked

expected The reference struct data

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.

7.4 VALIDATE_MEMORY(actual, expected, size)

Validates *size* byte of memory pointer to by *actual* against the reference data *expected*.

Parameters: ***actual*** A pointer to memory that will be checked

expected A pointer to memory containing reference data

size Size of memory block

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.

7.5 VALIDATE_STRING(actual, expected)

Validates the contents of a string against a reference string.

Parameters: ***actual*** The string data that will be checked

expected The reference string

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.

7.6 VALIDATE(actual, expected)

Validates a variable against a reference value.

Parameters: ***actual*** The variable that will be checked

expected The reference value

This macro is used to validate a single value. The macro can be used within tests or validator functions. 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.

7.7 EXPECT(function)

Validates that a function call is made to *function*.

Parameters: ***function*** The function that should be called

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 stub functions outside the unit.

For each test, a unique set of stub 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 stub 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 stub 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 stub 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 stub function. The stub 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 validator functions - i.e. during execution of the unit.

7.8 SIMULATE(function, retval)

Validates that a function call is made to *function*. If function is called, value *retval* is returned to the unit.

Parameters: ***function*** The function that should be called

retval The value that will be returned to the unit

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 stub 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 stub 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 validator functions - i.e. during execution of the unit.

7.9 EXPECT_VALIDATE(function, validator)

Validates that a function call is made to *function*. If function is called, function *validator* is called.

Parameters: ***function*** The function that should be called

validator The validator function that will calidate parameters and determine return value

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 stub function. If the unit calls the function at the right time, the framework will pass control to *validator* function.

The validator will typical implement validation of the parameters, but it can also execute tests. In order for the validator to validate the parameters, it is important that the validator has the same function prototype as the function it replaces. Upon return from the validator, the return value is passed back to the unit.

7.10 EXPECT_SIMULATE(function, validator, retval)

Validates that a function call is made to *function*. If function is called, function *validator* is called.

Parameters: ***function*** The function that should be called

validator The validator function that will calidate parameters and determine return value

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

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 stub function. If the unit calls the function at the right time, the framework will pass control to the *validator* function.

The value *retval* can be read in the validator using the macro SIMULATE_VALUE.

The validator will typical implement validation of the parameters, perhaps guided by the value of \ retval, but it can also execute tests.

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

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

7.11 SIMULATE_VALUE

The return value proposed to the stub function by the test.

Returns: The simulation value for the stub assigned by the currently executing test.

The SIMULATE_VALUE holds the value assigned to the stub 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 validator function. If the validator function are called in a validator function, 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 validator function.

7.12 EXECUTE_VALUE

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

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 EXECUTE_VALUE will in turn hold each of these values. The test can read this macro, setup the list of expected stub 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 validator functions - i.e. during execution of the unit.

7.13 EXECUTE(test)

Executes a test.

Parameters: ***test*** A function implementing the test. *test* is void function returning void.

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 normally used from 'void main(void)' to start a test, but it can also be nested and used during a test, or be called from a validator function - i.e. during execution of the unit.

7.14 EXECUTE_ADD(value)

Adds a variant to the next executed test.

Parameters: ***value*** A test value for a variant of a test.

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 *EXECUTE_ADD* 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 *EXECUTE_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.

7.15 EXECUTE_ADD_RANGE(start, stop)

Adds a range of variants to the next executed test.

Parameters: ***start*** First test value for a variant of a test.

stop Last test value for a variant of a test.

With the previous macro, it was possible to add a single variant value to a test. With the macro *EXECUTE_ADD_RANGE*, the same thing can be done with a range of values. Each consequent call to *EXECUTE_ADD_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 *EXECUTE_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.

7.16 EXECUTE_ADD_SET(arr)

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

Parameters: ***arr*** Array holding one or more sets

It is possible to vary the test by giving more than one value at a time. To do this use the macro *EXECUTE_ADD_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 *EXECUTE_ADD_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 *EXECUTE_VALUE*. The individual members of the set can be accessed from the test, by casting EXECUTE_VALUE to a pointer to one of the sets, e.g. ((set_type*)EXECUTE_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.

7.17 COMMENT

The command will generated a comment in the log file.

Parameters: ***format*** printf style comment format string

... printf style list of arguments

Will put a "TESTAPE:" prefixed comment in the logfile.