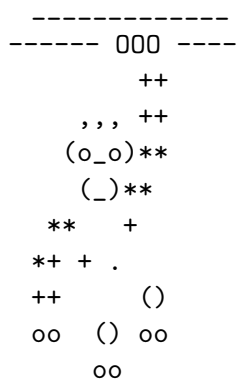
Using the TestApe framework

Martin Steen Nielsen, 2007



Contents

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 is designed to allow 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 being 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 over and over again. A TestApe executable is typically run whenever one of the modules in the unit have changed. The test can be used to verify 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.

The TestApe instrumenter was invented by him to avoid the tedious work of writing stub functions. 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 uses the test primitives available from the testape framework. The interface to the framework can be found in `testape.h`. The interface also contains several macros that might be used to make test cases more readable and easier to write.

2.1 Creating and executing a basic testcase

A full unit test consist 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 consist 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 do not see any difference compared to its normal environment. The value 0 is returned from every function call validated by the EXPECT macro. By the use of SIMULATE macro any non-zero value can be returned to the unit after the function validation.

2.4 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_AND_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_AND_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
```

2.5 Simulating output from validator functions

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;
    }
```

2.6 Organizing tests

When running the above test, 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 kind of 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);
    }
```

2.7 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 are 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 are 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

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.