Guidelines for test developers

How to add recipes

For any test that you want to perform, you write a script located in `test/recipes/`, named `{nn}-test_{name}.t`, where `{nn}` is a two digit number and `{name}` is a unique name of your choice.

Please note that if a test involves a new testing executable, you will need to do some additions in test/build.info. Please refer to the section "Changes to test/build.info" below.

Naming conventions

A test executable is named `test/{name}test.c`

A test recipe is named `test/recipes/{nn}-test_{name}.t`, where `{nn}` is a two digit number and `{name}` is a unique name of your choice.

The number `{nn}` is (somewhat loosely) grouped as follows:

```
00-04  sanity, internal and essential API tests
05-09  individual symmetric cipher algorithms
10-14  math (bignum)
15-19  individual asymmetric cipher algorithms
20-24  openssl commands (some otherwise not tested)
25-29  certificate forms, generation and verification
30-35  engine and evp
60-79  APIs:
   60  X509 subsystem
   61  BIO subsystem
   65  CMP subsystem
   70  PACKET layer
80-89  "larger" protocols (CA, CMS, OCSP, SSL, TSA)
90-98  misc
99     most time consuming tests [such as test_fuzz]
```

A recipe that just runs a test executable

A script that just runs a program looks like this:

```
#! /usr/bin/env perl

use OpenSSL::Test::Simple;

simple_test("test_{name}", "{name}test", "{name}");
```

`{name}` is the unique name you have chosen for your test.

The second argument to `simple_test` is the test executable, and `simple_test` expects it to be located in `test/`

For documentation on `OpenSSL::Test::Simple`, do `perldoc util/perl/OpenSSL/Test/Simple.pm`.

A recipe that runs a more complex test

For more complex tests, you will need to read up on Test::More and OpenSSL::Test. Test::More is normally preinstalled, do `man Test::More` for documentation. For OpenSSL::Test, do `perldoc util/perl/OpenSSL/Test.pm`.

A script to start from could be this:

```
#! /usr/bin/env perl

use strict;
use warnings;
use OpenSSL::Test;

setup("test_{name}");

plan tests => 2;                # The number of tests being performed

ok(test1, "test1");
ok(test2, "test1");

sub test1
{
    # test feature 1
}

sub test2
{
    # test feature 2
}
```

Changes to test/build.info

Whenever a new test involves a new test executable you need to do the following (at all times, replace {NAME} and {name} with the name of your test):

- add `{name}` to the list of programs under `PROGRAMS_NO_INST`
- create a three line description of how to build the test, you will have to modify the include paths and source files if you don't want to use the basic test framework:

  ```
  SOURCE[{name}]={name}.c
  INCLUDE[{name}]=.. ../include ../apps/include
  DEPEND[{name}]=../libcrypto libtestutil.a
  ```

Generic form of C test executables

```
#include "testutil.h"

static int my_test(void)
```

```
{
    int testresult = 0;                 /* Assume the test will fail    */
    int observed;

    observed = function();              /* Call the code under test     */
    if (!TEST_int_eq(observed, 2))      /* Check the result is correct  */
        goto end;                       /* Exit on failure - optional   */

    testresult = 1;                     /* Mark the test case a success */
end:
    cleanup();                          /* Any cleanup you require      */
    return testresult;
}

int setup_tests(void)
{
    ADD_TEST(my_test);                  /* Add each test separately     */
    return 1;                           /* Indicate success             */
}
```

You should use the `TEST_xxx` macros provided by `testutil.h` to test all failure conditions. These macros produce an error message in a standard format if the condition is not met (and nothing if the condition is met). Additional information can be presented with the `TEST_info` macro that takes a `printf` format string and arguments. `TEST_error` is useful for complicated conditions, it also takes a `printf` format string and argument. In all cases the `TEST_xxx` macros are guaranteed to evaluate their arguments exactly once. This means that expressions with side effects are allowed as parameters. Thus,

```
if (!TEST_ptr(ptr = OPENSSL_malloc(..)))
```

works fine and can be used in place of:

```
ptr = OPENSSL_malloc(..);
if (!TEST_ptr(ptr))
```

The former produces a more meaningful message on failure than the latter.

Note that the test infrastructure automatically sets up all required environment variables (such as `OPENSSL_MODULES`, `OPENSSL_CONF`, etc.) for the tests. Individual tests may choose to override the default settings as required.