

Unit Testing Framework

Thomas Braun © 2013
www.byte-physics.de

Version 1.01 compiled on Thu Oct 31 2013 20:56:20

Contents

1	Main Page	1
1.1	Assertion Types	1
1.2	Test Case	1
1.3	Test Suite	1
1.4	Test Hooks	1
1.5	Automate Test Runs	2
2	Example Documentation	2
2.1	example1-plain.ipf	2
2.2	example2-plain.ipf	3
2.3	example3-plain.ipf	4
2.4	example4-wavechecking.ipf	4
2.5	example5-overridehooks.ipf	5
2.6	example6-automatic-invocation.ipf	6
2.7	example7-fail-pass.ipf	7
3	Module Documentation	8
3.1	Helper functions	8
3.2	Test Assertions	9
3.3	Assertions flags	13
3.4	Wave existence flags	14
3.5	Wave equality flags	15
3.6	Default hook functions	16

1 Main Page

This package empowers a programmer to utilize unit testing for Igor Pro procedures and XOPs. For a quick start have a look at the [first example](#).

The basic building blocks of this package are [Assertions](#) (for checking if an entity fulfills specific properties), [Test Cases](#) (group of assertions) and [Test Suites](#) (group of test cases).

Interface design and naming is inspired by the [Boost Test Library](#).

1.1 Assertion Types

An assertion checks that a given condition is true. Or in more general terms that an entity fulfills specific properties. Test assertions are defined for strings, variables and waves and have ALL CAPS names. They usually come in triplets which differ only in how they react on a failed assertion. Comparing two variables for example can be done with [WARN_EQUAL_VAR](#), [CHECK_EQUAL_VAR](#) or [REQUIRE_EQUAL_VAR](#).

The following table summarizes the action on a failed assertion:

Type	Create Log Message	Increment Error Count	Abort execution immediately
WARN	YES	NO	NO
CHECK	YES	YES	NO
REQUIRE	YES	YES	YES

If in doubt use the CHECK variant. For the sake of clarity only the CHECK variants are documented, as the interface for REQUIRE and WARN is equivalent. The assertions with only one variant are [PASS](#) and [FAIL](#), see also [example7-fail-pass.ipf](#).

1.2 Test Case

A test case is one of the basic building blocks grouping assertions together. A function is considered a test case if it fulfills all of the following properties:

- takes no parameters
- its name does not end on `_IGNORE`
- is either non-static or static and part of a regular module

The second rule allows advanced users to add their own helper functions. It is advised to define all test cases as static functions and to create one regular module per procedure file.

A single test case from a test suite can be run using the optional `testCase` parameter of [RunTest](#).

1.3 Test Suite

A test suite is a group of test cases which should belong together and is equal to a procedure file. Therefore tests suites can not be nested, although multiple test suites can be run with once command using the parameter `procWinList` of [RunTest](#).

1.4 Test Hooks

To ensure proper test case execution and enable book keeping, specific hook functions are called before/after distinct events. These hook functions always come in pairs with their names ending on `_BEGIN` and `_END`. Before the first test case of the first test suite is executed, the hook [TEST_BEGIN](#) is called, therefore [TEST_END](#) marks

the last function being called immediately before `RunTest` returns. Similarly the hooks `TEST_SUITE_BEGIN` and `TEST_SUITE_END` are called before and after every test suite, `TEST_CASE_BEGIN` and `TEST_CASE_END` before and after every test case.

In case the default hook functions don't suite your needs, it is explicitly **not** advised to just adapt them. Instead use [test hook overrides](#) and override them on a global or per test suite level.

1.4.1 Override Test Hooks

The default test hooks can be overridden by defining your own version of the hooks suffixed with `_OVERRIDE`. The override hooks for `TEST_BEGIN` and `TEST_END` can only be overridden by functions in ProcGlobal. The override hooks for test suites/cases can be overridden globally if they reside in ProcGlobal context, or for a specific test suite only if they are defined in the same regular module as that test suite. Overriding here means that the default test hook is **not** executed. In case you still want to have the default test hook executed, you have to call it yourself in the override function as done in [example 5](#).

The override test hooks have to accept exactly one string parameter, which is the name of the test suite group, test suite name or test case name.

1.5 Automate Test Runs

To further simplify test execution it is possible to automate test runs from the command line.

Steps to do that include:

- Implement a function called `run` in ProcGlobal context taking no parameters. This function must perform all necessary steps for test execution, which is at least one call to `RunTest`.
- Put the test experiment together with your test suites (procedure files) and the script helper/autorun-test.bat into its own folder
- Run the batch file autorun-test.bat
- Inspect the created log file

See also [example6-automatic-invocation.ipf](#).

2 Example Documentation

2.1 example1-plain.ipf

Test suite showing the basic working principles.

```
#pragma rtGlobals=3

#include "unit-testing"

// Execute the test suite, same named as this procedure file
// with RunTest("example1-plain.ipf")

Function TestModulo()

    CHECK_EQUAL_VAR(abs(1.5), 1.5)
    CHECK_EQUAL_VAR(abs(-1.5), 1.5)
    CHECK_EQUAL_VAR(abs(NaN), NaN)
    // remember that NaN is not equal to NaN
```

```

// this check will generate a warning message but due
// to the usage of WARN instead of CHECK not increment the error count
WARN(abs(NaN) == NaN)
CHECK_EQUAL_VAR(abs(INF), INF)
CHECK_EQUAL_VAR(abs(-INF), INF)
End

```

2.2 example2-plain.ipf

Test suite with run routine and module/static usage. See the section about [test cases](#) why the function run_IGNORE() is not considered a test case.

```

#pragma rtGlobals=3
#pragma ModuleName=Example2

#include "unit-testing"

// Command: run_IGNORE()
// Shows how to use ignore routines

Function run_IGNORE()
    // All of these commands run the test suite "example2-plain.ipf"

    // executes all test cases of this file
    RunTest("example2-plain.ipf")
    // execute only one test case at a time
    RunTest("example2-plain.ipf", testCase="VerifyDefaultStringBehaviour")
    // Give all test suites a descriptive name
    RunTest("example2-plain.ipf", name="My first test")
End

// Making the function static prevents name clashes with other
// procedure files. Using static functions requires also the
// line "#pragma ModuleName" from above.
static Function VerifyDefaultStringBehaviour()

    string nullString
    string emptyString = ""
    string strLow       = "1234a"
    string strUP        = "1234A"

    // by default string comparison is done case insensitive
    CHECK_EQUAL_STR(strLow, strUP)
    CHECK_EQUAL_STR(strLow, strUP, case_sensitive=0)
    // the next test fails
    WARN_EQUAL_STR(strLow, strUP, case_sensitive=1)

    CHECK_NEQ_STR(emptyString, nullString)
    CHECK_NEQ_STR(strLow, nullString)
    CHECK_EMPTY_STR(emptyString)
    CHECK_NULL_STR(nullString)
    CHECK_EQUAL_VAR(strlen(strLow), 5)
End

```

2.3 example3-plain.ipf

Test suite emphasising the difference between the [WARN\(\)](#), [CHECK\(\)](#) and [REQUIRE\(\)](#) assertion variants.

```
#pragma rtGlobals=3
#pragma ModuleName=Example3

#include "unit-testing"

// Command: RunTest("example3-plain.ipf")
// The error count of this test suite is 2

// WARN_* does not increment the error count
Function WarnTest()

    WARN_EQUAL_VAR(1.0,0.0)
End

// CHECK_* increments the error count
Function CheckTest()

    CHECK_EQUAL_VAR(1.0,0.0)
End

// REQUIRE_* increments the error count and will stop execution
// of the test case immediately.
// Nevertheless the test end hooks are still executed.
Function RequireTest()

    REQUIRE_EQUAL_VAR(1.0,0.0)
    print "I'm never reached :("
End
```

See also [Assertion Types](#).

2.4 example4-wavechecking.ipf

Test suite showing some test assertions Xfor waves.

```
#pragma rtGlobals=3
#pragma ModuleName=Example4

#include "unit-testing"

// Command: RunTest("example4-wavechecking.ipf")
// Helper functions to check wave types and compare with
// reference waves are also provided

static Function CheckMakeDouble()
    CHECK_EMPTY_FOLDER() // checks that the cdf is completely empty

    Make/D myWave
    CHECK_WAVE(myWave, NUMERIC_WAVE, minorType=DOUBLE_WAVE)
    CHECK_EQUAL_VAR(DimSize(myWave, 0), 128)
```

```

// as this test case is always executed in a fresh datafolder
// we don't have to use the overwrite /O option for Duplicate
Duplicate myWave, myWaveCopy
CHECK_EQUAL WAVES (myWave, myWaveCopy)

End

static Function CheckMakeText ()
CHECK_EMPTY_FOLDER ()

Make/T/D myWave
CHECK_WAVE (myWave, TEXT_WAVE)
CHECK_EQUAL_VAR (DimSize (myWave, 0), 128)

Duplicate myWave, myWaveCopy
CHECK_EQUAL WAVES (myWave, myWaveCopy)
End

```

2.5 example5-overridehooks.ipf

Two test suites showing how to use test hook overrides.

```

#pragma rtGlobals=3
#pragma ModuleName=Example5

#include "unit-testing"

// RunTest ("example5-overridehooks.ipf;example5-overridehooks-otherSuite.ipf")

static Function TEST_CASE_BEGIN_OVERRIDE (name)
string name

print "I'm for all test cases in *this* test suite"
End

static Function TEST_CASE_END_OVERRIDE (name)
string name

printf "I'm overriding test case end for (%s) in this test suite only\r", name
TEST_CASE_END (name)
End

static Function CheckSquareRoot ()

CHECK_EQUAL_VAR (sqrt (4.0), 2.0)
CHECK_CLOSE_VAR (sqrt (2.0), 1.4142, tol=1e-4)
End

```

```

#pragma rtGlobals=3

#include "unit-testing"

// As this procedure file is in ProcGlobal context
// the test hook overrides are global.

```

```

Function TEST_BEGIN_OVERRIDE (name)
    string name

    print "I can only be overridden globally"
End

Function TEST_END_OVERRIDE (name)
    string name

    print "I can only be overridden globally, too"
End

Function TEST_CASE_END_OVERRIDE (name)
    string name

    print "I'm for all test suites overriding the test case end"
    TEST_CASE_END (name)
End

Function TEST_SUITE_BEGIN_OVERRIDE (name)
    string name

    print "Global test suite begin override"
End

Function TEST_SUITE_END_OVERRIDE (name)
    string name

    print "Global test suite end override"
    TEST_SUITE_END (name)
End

Function CheckBasicMath()

    CHECK_EQUAL_VAR (1+2, 3)
End

```

2.6 example6-automatic-invocation.ipf

Test suite showing how to automate testing from the command line. See also [Automate Test Runs](#).

```

#pragma rtGlobals=3
#pragma ModuleName=Example6

#include "unit-testing"

// Command: Call "autorun-test.bat" without Igor Pro running

static Function CheckTrigonometricFunctions()
    CHECK_EQUAL_VAR (sin(0.0), 0.0)
    CHECK_EQUAL_VAR (cos(0.0), 1.0)
    CHECK_EQUAL_VAR (tan(0.0), 0.0)
End

```

```
#pragma rtGlobals=3

#include "unit-testing"

Function run()
    RunTest("example6-automatic-invocation.ipf")
End
```

2.7 example7-fail-pass.ipf

Test suite showing how to check for aborts using **PASS** and **FAIL** assertions.

```
#pragma rtGlobals=3
#pragma ModuleName=Example7

#include "unit-testing"

// Function to add two numbers which aborts on NaN in either a or b
Function AddNormalNumbers(a, b)
    variable a, b

    if(numType(a) == 2 || numType(b) == 2)
        Abort
    endif

    return a + b
End

// Command: RunTest("example7-fail-pass.ipf")
// Helper functions to use with try/catch
static Function CheckAddNormalNumbers_a_nan()

    variable a = NaN
    variable b = 1.0
    try
        AddNormalNumbers(a,b)
    catch
        PASS()
        return 0
    endtry
    FAIL()
End

static Function CheckAddNormalNumbers_b_nan()

    variable a = 1.0
    variable b = NaN
    try
        AddNormalNumbers(a,b)
    catch
        PASS()
        return 0
    endtry
    FAIL()
End
```

```

static Function CheckAddNormalNumbers_both_nan()

    variable a = NaN
    variable b = NaN
    try
        AddNormalNumbers(a,b)
    catch
        PASS()
        return 0
    endtry
    FAIL()

End

```

3 Module Documentation

3.1 Helper functions

Functions

- variable `DisableDebugOutput()`
- variable `EnableDebugOutput()`
- variable `RunTest` (string *procWinList*, string *name*, string *testCase*)

3.1.1 Detailed Description

Runner and helper functions.

3.1.2 Function Documentation

variable `DisableDebugOutput()`

Turns debug output off.

variable `EnableDebugOutput()`

Turns debug output on.

variable `RunTest` (string *procWinList*, string *name*, string *testCase*)

Main function to execute one or more test suites.

Parameters

<i>procWinList</i>	semicolon ";" separated list of procedure files
<i>name</i>	(optional) descriptive name for the executed test suites
<i>testCase</i>	(optional) function name, resembling one test case, which should be executed only

Returns

total number of errors

3.2 Test Assertions

Functions

- variable `CHECK` (variable `var`)
- variable `CHECK_CLOSE_VAR` (variable `var1`, variable `var2`, variable `tol`, variable `strong_or_weak`)
- variable `CHECK_EMPTY_FOLDER` ()
- variable `CHECK_EMPTY_STR` (string `*str`)
- variable `CHECK_EQUAL_STR` (string `*str1`, string `*str2`, variable `case_sensitive`)
- variable `CHECK_EQUAL_VAR` (variable `var1`, variable `var2`)
- variable `CHECK_EQUAL_WAVES` (Wave/Z `wv1`, Wave/Z `wv2`, variable `mode`, variable `tol`)
- variable `CHECK_NEQ_STR` (string `*str1`, string `*str2`, variable `case_sensitive`)
- variable `CHECK_NEQ_VAR` (variable `var1`, variable `var2`)
- variable `CHECK_NULL_STR` (string `*str`)
- variable `CHECK_SMALL_VAR` (variable `var`, variable `tol`)
- variable `CHECK_WAVE` (Wave/Z `wv`, variable `majorType`, variable `minorType`)
- variable `FAIL` ()
- variable `PASS` ()
- variable `REQUIRE` (variable `var`)
- variable `REQUIRE_CLOSE_VAR` (variable `var1`, variable `var2`, variable `tol`, variable `strong_or_weak`)
- variable `REQUIRE_EMPTY_FOLDER` ()
- variable `REQUIRE_EMPTY_STR` (string `*str`)
- variable `REQUIRE_EQUAL_STR` (string `*str1`, string `*str2`, variable `case_sensitive`)
- variable `REQUIRE_EQUAL_VAR` (variable `var1`, variable `var2`)
- variable `REQUIRE_EQUAL_WAVES` (Wave/Z `wv1`, Wave/Z `wv2`, variable `mode`, variable `tol`)
- variable `REQUIRE_NEQ_STR` (string `*str1`, string `*str2`, variable `case_sensitive`)
- variable `REQUIRE_NEQ_VAR` (variable `var1`, variable `var2`)
- variable `REQUIRE_NULL_STR` (string `*str`)
- variable `REQUIRE_SMALL_VAR` (variable `var`, variable `tol`)
- variable `REQUIRE_WAVE` (Wave/Z `wv`, variable `majorType`, variable `minorType`)
- variable `WARN` (variable `var`)
- variable `WARN_CLOSE_VAR` (variable `var1`, variable `var2`, variable `tol`, variable `strong_or_weak`)
- variable `WARN_EMPTY_FOLDER` ()
- variable `WARN_EMPTY_STR` (string `*str`)
- variable `WARN_EQUAL_STR` (string `*str1`, string `*str2`, variable `case_sensitive`)
- variable `WARN_EQUAL_VAR` (variable `var1`, variable `var2`)
- variable `WARN_EQUAL_WAVES` (Wave/Z `wv1`, Wave/Z `wv2`, variable `mode`, variable `tol`)
- variable `WARN_NEQ_STR` (string `*str1`, string `*str2`, variable `case_sensitive`)
- variable `WARN_NEQ_VAR` (variable `var1`, variable `var2`)
- variable `WARN_NULL_STR` (string `*str`)
- variable `WARN_SMALL_VAR` (variable `var`, variable `tol`)
- variable `WARN_WAVE` (Wave/Z `wv`, variable `majorType`, variable `minorType`)

3.2.1 Detailed Description

Test assertions for variables, strings, waves and helper functions.

3.2.2 Function Documentation

variable `CHECK` (variable `var`)

Tests if `var` is true (1).

Parameters

<i>var</i>	variable to test
------------	------------------

variable `CHECK_CLOSE_VAR (variable var1, variable var2, variable tol, variable strong_or_weak)`

Compares two variables and determines if they are close.

Based on the implementation of "Floating-point comparison algorithms" in the C++ Boost unit testing framework.

Literature:

The art of computer programming (Vol II). Donald. E. Knuth. 0-201-89684-2. Addison-Wesley Professional; 3 edition, page 234 equation (34) and (35).

Parameters

<i>var1</i>	first variable
<i>var2</i>	second variable
<i>tol</i>	(optional) tolerance, defaults to 1e-8
<i>strong_or_weak</i>	(optional) type of condition, can be 0 for weak or 1 for strong (default)

variable `CHECK_EMPTY_FOLDER ()`

Tests if the current data folder is empty.

Counted are objects with type waves, strings, variables and folders

variable `CHECK_EMPTY_STR (string * str)`

Tests if *str* is empty.

A null string is never empty.

Parameters

<i>str</i>	string to test
------------	----------------

variable `CHECK_EQUAL_STR (string * str1, string * str2, variable case_sensitive)`

Compares two strings for equality.

Parameters

<i>str1</i>	first string
<i>str2</i>	second string
<i>case_sensitive</i>	(optional) should the comparison be done case sensitive (1) or case insensitive (0, the default)

variable `CHECK_EQUAL_VAR (variable var1, variable var2)`

Tests two variables for equality.

For variables holding floating point values it is often more desirable use `CHECK_CLOSE_VAR` instead. To fulfill semantic correctness this assertion treats two variables with both holding NaN as equal.

Parameters

<i>var1</i>	first variable
<i>var2</i>	second variable

variable CHECK_EQUAL_WAVES (Wave/Z *wv1*, Wave/Z *wv2*, variable *mode*, variable *tol*)

Tests two waves for equality.

Parameters

<i>wv1</i>	first wave
<i>wv2</i>	second wave
<i>mode</i>	(optional) features of the waves to compare, defaults to all modes, defined at Wave equality flags
<i>tol</i>	(optional) tolerance for comparison, by default 0.0 which does byte-by-byte comparison (relevant only for mode=WAVE_DATA)

variable CHECK_NEQ_STR (string * *str1*, string * *str2*, variable *case_sensitive*)

Compares two strings for unequality.

Parameters

<i>str1</i>	first string
<i>str2</i>	second string
<i>case_sensitive</i>	(optional) should the comparison be done case sensitive (1) or case insensitive (0, the default)

variable CHECK_NEQ_VAR (variable *var1*, variable *var2*)

Tests two variables for inequality.

Parameters

<i>var1</i>	first variable
<i>var2</i>	second variable

variable CHECK_NULL_STR (string * *str*)

Tests if *str* is null.

An empty string is never null.

Parameters

<i>str</i>	string to test
------------	----------------

variable CHECK_SMALL_VAR (variable *var*, variable *tol*)

Tests if a variable is small using the inequality $|var| < |tol|$.

Parameters

<i>var</i>	variable
<i>tol</i>	(optional) tolerance, defaults to 1e-8

variable CHECK_WAVE (Wave/Z *wv*, variable *majorType*, variable *minorType*)

Tests a wave for existence and its type.

Parameters

<i>wv</i>	wave reference
<i>majorType</i>	major wave type

<i>minorType</i>	(optional) minor wave type
------------------	----------------------------

See also

[Wave existence flags](#)

variable FAIL ()

Force the test case to fail.

variable PASS ()

Increase the assertion counter only.

variable REQUIRE (variable *var*)

variable REQUIRE_CLOSE_VAR (variable *var1*, variable *var2*, variable *tol*, variable *strong_or_weak*)

variable REQUIRE_EMPTY_FOLDER ()

variable REQUIRE_EMPTY_STR (string * *str*)

variable REQUIRE_EQUAL_STR (string * *str1*, string * *str2*, variable *case_sensitive*)

variable REQUIRE_EQUAL_VAR (variable *var1*, variable *var2*)

variable REQUIRE_EQUAL WAVES (Wave/Z *wv1*, Wave/Z *wv2*, variable *mode*, variable *tol*)

variable REQUIRE_NEQ_STR (string * *str1*, string * *str2*, variable *case_sensitive*)

variable REQUIRE_NEQ_VAR (variable *var1*, variable *var2*)

variable REQUIRE_NULL_STR (string * *str*)

variable REQUIRE_SMALL_VAR (variable *var*, variable *tol*)

variable REQUIRE_WAVE (Wave/Z *wv*, variable *majorType*, variable *minorType*)

variable WARN (variable *var*)

variable WARN_CLOSE_VAR (variable *var1*, variable *var2*, variable *tol*, variable *strong_or_weak*)

variable WARN_EMPTY_FOLDER ()

variable WARN_EMPTY_STR (string * *str*)

variable WARN_EQUAL_STR (string * *str1*, string * *str2*, variable *case_sensitive*)

variable WARN_EQUAL_VAR (variable *var1*, variable *var2*)

variable WARN_EQUAL WAVES (Wave/Z *wv1*, Wave/Z *wv2*, variable *mode*, variable *tol*)

variable WARN_NEQ_STR (string * *str1*, string * *str2*, variable *case_sensitive*)

variable WARN_NEQ_VAR (variable *var1*, variable *var2*)

variable WARN_NULL_STR (string * *str*)

variable WARN_SMALL_VAR (variable *var*, variable *tol*)

variable WARN_WAVE (Wave/Z *wv*, variable *majorType*, variable *minorType*)

3.3 Assertions flags

Modules

- [Wave existence flags](#)
- [Wave equality flags](#)

3.3.1 Detailed Description

Constants for assertion test tuning.

3.4 Wave existence flags

Variables

- const variable `COMPLEX_WAVE` = 0x01
- const variable `DOUBLE_WAVE` = 0x04
- const variable `FLOAT_WAVE` = 0x02
- const variable `INT16_WAVE` = 0x16
- const variable `INT32_WAVE` = 0x20
- const variable `INT8_WAVE` = 0x08
- const variable `NUMERIC_WAVE` = 1
- const variable `TEXT_WAVE` = 2
- const variable `UNSIGNED_WAVE` = 0x40

3.4.1 Detailed Description

Values for `majorType` / `minorType` of `WARN_WAVE`, `CHECK_WAVE` and `REQUIRE_WAVE`.

3.4.2 Variable Documentation

const variable `COMPLEX_WAVE` = 0x01

const variable `DOUBLE_WAVE` = 0x04

const variable `FLOAT_WAVE` = 0x02

const variable `INT16_WAVE` = 0x16

const variable `INT32_WAVE` = 0x20

const variable `INT8_WAVE` = 0x08

const variable `NUMERIC_WAVE` = 1

const variable `TEXT_WAVE` = 2

const variable `UNSIGNED_WAVE` = 0x40

3.5 Wave equality flags

Variables

- const variable `DATA_FULL_SCALE` = 256
- const variable `DATA_UNITS` = 8
- const variable `DIMENSION_LABELS` = 32
- const variable `DIMENSION_SIZES` = 512
- const variable `DIMENSION_UNITS` = 16
- const variable `WAVE_DATA` = 1
- const variable `WAVE_DATA_TYPE` = 2
- const variable `WAVE_LOCK_STATE` = 128
- const variable `WAVE_NOTE` = 64
- const variable `WAVE_SCALING` = 4

3.5.1 Detailed Description

Values for `mode` in `WARN_EQUAL_WAVES`, `CHECK_EQUAL_WAVES` and `REQUIRE_EQUAL_WAVES`.

3.5.2 Variable Documentation

const variable `DATA_FULL_SCALE` = 256

const variable `DATA_UNITS` = 8

const variable `DIMENSION_LABELS` = 32

const variable `DIMENSION_SIZES` = 512

const variable `DIMENSION_UNITS` = 16

const variable `WAVE_DATA` = 1

const variable `WAVE_DATA_TYPE` = 2

const variable `WAVE_LOCK_STATE` = 128

const variable `WAVE_NOTE` = 64

const variable `WAVE_SCALING` = 4

3.6 Default hook functions

Functions

- variable `TEST_BEGIN` (string name)
- variable `TEST_CASE_BEGIN` (string testCase)
- variable `TEST_CASE_END` (string testCase)
- variable `TEST_END` (string name)
- variable `TEST_SUITE_BEGIN` (string testSuite)
- variable `TEST_SUITE_END` (string testSuite)

3.6.1 Detailed Description

Default implementation of test hook functions.

3.6.2 Function Documentation

variable `TEST_BEGIN` (string *name*)

Default test begin hook.

The hook is immediately called after RunTest starts.

Parameters

<i>name</i>	name of the test suite group
-------------	------------------------------

variable `TEST_CASE_BEGIN` (string *testCase*)

Default hook for test case begin.

The hook is called before executing the test case.

Parameters

<i>testCase</i>	name of the test case
-----------------	-----------------------

variable `TEST_CASE_END` (string *testCase*)

Default hook for test case end.

The hook is called after executing the test case.

Parameters

<i>testCase</i>	name of the test case
-----------------	-----------------------

variable `TEST_END` (string *name*)

Default test end hook.

The hook is called after all tests suites.

Parameters

<i>name</i>	name of the test suite group
-------------	------------------------------

variable `TEST_SUITE_BEGIN (string testSuite)`

Default hook for test suite begin.

The hook is called before executing the first test case of every test suite.

Parameters

<i>testSuite</i>	name of the test suite
------------------	------------------------

variable `TEST_SUITE_END (string testSuite)`

Default hook for test suite end.

The hook is called after executing the last test case of every test suite.

Parameters

<i>testSuite</i>	name of the test suite
------------------	------------------------

Index

A	
Assertions flags	13
C	
CHECK	
Test Assertions	9
CHECK_CLOSE_VAR	
Test Assertions	10
CHECK_EMPTY_FOLDER	
Test Assertions	10
CHECK_EMPTY_STR	
Test Assertions	10
CHECK_EQUAL_STR	
Test Assertions	10
CHECK_EQUAL_VAR	
Test Assertions	10
CHECK_EQUAL_WAVES	
Test Assertions	10
CHECK_NEQ_STR	
Test Assertions	11
CHECK_NEQ_VAR	
Test Assertions	11
CHECK_NULL_STR	
Test Assertions	11
CHECK_SMALL_VAR	
Test Assertions	11
CHECK_WAVE	
Test Assertions	11
COMPLEX_WAVE	
Wave existence flags	14
D	
DATA_FULL_SCALE	
Wave equality flags	15
DATA_UNITS	
Wave equality flags	15
DIMENSION_LABELS	
Wave equality flags	15
DIMENSION_SIZES	
Wave equality flags	15
DIMENSION_UNITS	
Wave equality flags	15
DOUBLE_WAVE	
Wave existence flags	14
Default hook functions	16
TEST_BEGIN	16
TEST_CASE_BEGIN	16
TEST_CASE_END	16
TEST_END	16
TEST_SUITE_BEGIN	16
TEST_SUITE_END	17
DisableDebugOutput	
Helper functions	8
E	
EnableDebugOutput	
Helper functions	8
F	
FAIL	
Test Assertions	12
FLOAT_WAVE	
Wave existence flags	14
H	
Helper functions	8
DisableDebugOutput	8
EnableDebugOutput	8
RunTest	8
I	
INT16_WAVE	
Wave existence flags	14
INT32_WAVE	
Wave existence flags	14
INT8_WAVE	
Wave existence flags	14
N	
NUMERIC_WAVE	
Wave existence flags	14
P	
PASS	
Test Assertions	12
R	
REQUIRE	
Test Assertions	12
REQUIRE_CLOSE_VAR	
Test Assertions	12
REQUIRE_EMPTY_FOLDER	
Test Assertions	12
REQUIRE_EMPTY_STR	
Test Assertions	12
REQUIRE_EQUAL_STR	
Test Assertions	12
REQUIRE_EQUAL_VAR	
Test Assertions	12
REQUIRE_EQUAL_WAVES	
Test Assertions	12
REQUIRE_NEQ_STR	
Test Assertions	12
REQUIRE_NEQ_VAR	
Test Assertions	12
REQUIRE_NULL_STR	
Test Assertions	12
REQUIRE_SMALL_VAR	
Test Assertions	12
REQUIRE_WAVE	
Test Assertions	12
RunTest	
Helper functions	8

T		W	
TEST_BEGIN		WARN	
Default hook functions.....	16	Test Assertions.....	12
TEST_CASE_BEGIN		WARN_CLOSE_VAR	
Default hook functions.....	16	Test Assertions.....	12
TEST_CASE_END		WARN_EMPTY_FOLDER	
Default hook functions.....	16	Test Assertions.....	12
TEST_END		WARN_EMPTY_STR	
Default hook functions.....	16	Test Assertions.....	12
TEST_SUITE_BEGIN		WARN_EQUAL_STR	
Default hook functions.....	16	Test Assertions.....	12
TEST_SUITE_END		WARN_EQUAL_VAR	
Default hook functions.....	17	Test Assertions.....	12
TEXT_WAVE		WARN_EQUAL_WAVES	
Wave existence flags.....	14	Test Assertions.....	12
Test Assertions.....	9	WARN_NEQ_STR	
CHECK.....	9	Test Assertions.....	12
CHECK_CLOSE_VAR.....	10	WARN_NEQ_VAR	
CHECK_EMPTY_FOLDER.....	10	Test Assertions.....	12
CHECK_EMPTY_STR.....	10	WARN_NULL_STR	
CHECK_EQUAL_STR.....	10	Test Assertions.....	12
CHECK_EQUAL_VAR.....	10	WARN_SMALL_VAR	
CHECK_EQUAL_WAVES.....	10	Test Assertions.....	12
CHECK_NEQ_STR.....	11	WARN_WAVE	
CHECK_NEQ_VAR.....	11	Test Assertions.....	12
CHECK_NULL_STR.....	11	WAVE_DATA	
CHECK_SMALL_VAR.....	11	Wave equality flags.....	15
CHECK_WAVE.....	11	WAVE_DATA_TYPE	
FAIL.....	12	Wave equality flags.....	15
PASS.....	12	WAVE_LOCK_STATE	
REQUIRE.....	12	Wave equality flags.....	15
REQUIRE_CLOSE_VAR.....	12	WAVE_NOTE	
REQUIRE_EMPTY_FOLDER.....	12	Wave equality flags.....	15
REQUIRE_EMPTY_STR.....	12	WAVE_SCALING	
REQUIRE_EQUAL_STR.....	12	Wave equality flags.....	15
REQUIRE_EQUAL_VAR.....	12	Wave equality flags.....	15
REQUIRE_EQUAL_WAVES.....	12	DATA_FULL_SCALE.....	15
REQUIRE_NEQ_STR.....	12	DATA_UNITS.....	15
REQUIRE_NEQ_VAR.....	12	DIMENSION_LABELS.....	15
REQUIRE_NULL_STR.....	12	DIMENSION_SIZES.....	15
REQUIRE_SMALL_VAR.....	12	DIMENSION_UNITS.....	15
REQUIRE_WAVE.....	12	WAVE_DATA.....	15
WARN.....	12	WAVE_DATA_TYPE.....	15
WARN_CLOSE_VAR.....	12	WAVE_LOCK_STATE.....	15
WARN_EMPTY_FOLDER.....	12	WAVE_NOTE.....	15
WARN_EMPTY_STR.....	12	WAVE_SCALING.....	15
WARN_EQUAL_STR.....	12	Wave existence flags.....	14
WARN_EQUAL_VAR.....	12	COMPLEX_WAVE.....	14
WARN_EQUAL_WAVES.....	12	DOUBLE_WAVE.....	14
WARN_NEQ_STR.....	12	FLOAT_WAVE.....	14
WARN_NEQ_VAR.....	12	INT16_WAVE.....	14
WARN_NULL_STR.....	12	INT32_WAVE.....	14
WARN_SMALL_VAR.....	12	INT8_WAVE.....	14
WARN_WAVE.....	12	NUMERIC_WAVE.....	14
		TEXT_WAVE.....	14
		UNSIGNED_WAVE.....	14
U			
UNSIGNED_WAVE			
Wave existence flags.....	14		