Unit Testing Framework

Running under Igor Pro 6/7
License: 3-Clause BSD

www.byte-physics.de

Version 1.06 compiled on Tue Mar 21 2017 20:43:35

Contents

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 fullfills 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 fullfills specific properties. Test assertions are defined for strings, variables and waves and have ALL CAPS names. They usually come in tripletts 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-uncaught-aborts.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 fullfills 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 distinctive module per procedure file.

A single test case from a test suite can be run using the optional `testCase` parameter of RunTest. This is also true when multiple test suites are executed in a test run and each has such a test case. If the given test case does not exist in any test suite it is treated as error.

Example: In test suite `TestSuite_1.ipf` test cases `static Step1` and `static Step2` are defined and in test suite `TestSuite_2.ipf` test cases `static Step1` , `static Step2` and `static Step3` are defined. Calling

```
Runtest("TestSuite_1.ipf;TestSuite_2.ipf", testCase="Step1")
```

executes two test cases, `Step1` in test suite 1 and `Step1` in test suite 2.

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 one command using the parameter `procWinList` of RunTest.

1.4 Test Hooks

At certain points in the execution of a test run the user can add own code by including functions with reserved names.

The following functions are reserved:

`TEST_BEGIN_OVERRIDE(string name)`	Executed at the begin of a test run name is a string with the tests name
`TEST_END_OVERRIDE(string name)`	Executed at the end of a test run Name is a string with the tests name note: As this function is executed at the very end of a test run the Igor debugger state is already reset to the state it had before RunTest was executed.
`TEST_SUITE_BEGIN_OVERRIDE(string testSuiteName)`	Executed at the begin of a test suite (test functions that are defined within a single procedure) TestSuiteName is a string with the test suites name (procedure name) This function can also be defined locally in a test suite with the `static` keyword.
`TEST_SUITE_END_OVERRIDE(string testSuiteName)`	Executed at the end of a test suite (test functions that are defined within a single procedure) TestSuiteName is a string with the test suites name (procedure name) This function can also be defined locally in a test suite with the `static` keyword.
`TEST_CASE_BEGIN_OVERRIDE(string testCaseName)`	Executed at the begin of a test case TestCaseName is a string with the test case name (function name) This function can also be defined locally in a test suite with the `static` keyword.
`TEST_CASE_END_OVERRIDE(string testCaseName)`	Executed at the end of a test case TestCaseName is a string with the test case name (function name) This function can also be defined locally in a test suite with the `static` keyword.

When defined these function are executed globally, e.g.

```
TEST_CASE_BEGIN_OVERRIDE
```

gets executed at the beginning of each test case. For the case that a user would like to have code executed only in a specific test suite these function can be defined locally within the test suite by declaring them `static`. Then the local function replaces the globally defined function. If the local defined function should only extend a global function the user can call the global function with:

```
FUNCREF USER_HOOK_PROTO tcbegin_global = $"ProcGlobal#TEST_CASE_BEGIN_OVERRIDE"
tcbegin_global(TestCaseName)
```

While a test run is executed the Igor debugger is by default disabled.

To give one example: By default each test case is executed in its own temporary data folder.

`TEST_CASE_BEGIN_OVERRIDE`

can now be used to set the data folder to root: such that each test case gets executed in root and no cleanup is executed afterwards. The next test case starts with the data the previous test case left in root:.

The functionality with additional user code at certain points of a test run is demonstrated in example5-extensionhooks.ipf.

1.5 JUNIT Output

The igor unit testing framework supports output of test run results in JUNIT compatible format. The output can be enabled by adding the optional parameter `enableJU=1` to `RunTest`. The XML output files are written to the experiments `home` directory with naming `JU_Experiment_Date_Time.xml`. If a file with the same name already exists a three digit number is added to the name. The JUNIT Output also contains the history log of each test case and test suite.

1.6 Test Anything Protocol Output

Output according to the Test Anything Protocol (TAP) standard 13 can be enabled with the optional parameter enableTAP = 1 of RunTest. The output is written into a file in the experiment folder with a unique generated name tap_'time'.log. This prevents accidental overwrites of previous test runs. A TAP output file combines all Test Cases from all Test Suites given in RunTest.
Additional TAP compliant descriptions and directives for each Test Case can be added in the two lines preceding the function of a Test Case.

```
// #TAPDescription: My description here
// #TAPDirective: My directive here
```

For directives two additional key words are defined that can be written at the beginning of the directive message.

- `TODO`

  indicates a Test that includes a part of the program still in development. Failures here will be ignores by a TAP consumer.

- `SKIP`

  indicates a Test that should be skipped. A test with this directive key word is not executed and reported always as 'ok'.

Examples:

```
// #TAPDirective: TODO routine that should be tested is still under development
```

or

```
// #TAPDirective: SKIP this test gets skipped
```

See the Experiment in the TAP_Example folder for reference.

1.7 Optional Parameters for RunTest

`enableJU = 1`	A JUNIT compatible XML file is written at the end of the test run. It allows the combination of this framework with continious integration servers like Atlassian Bamboo.
`enableTAP = 1`	A Test Anything Protocol (TAP) version 13 compatible file is written at the end of the test run.
`allowDebug = 1`	RunTest will leave the Igor debugger in its current state. This is useful for a brief look where user code generates an exception.
`keepDataFolder = 1`	The temporary data folder wherein each test case is executed is not cleaned up at the end of the test case. This allows to review the data produced.

1.8 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
#pragma TextEncoding="UTF-8"

#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_IGN↩ ORE() is not considered a test case.

```
#pragma rtGlobals=3
#pragma TextEncoding="UTF-8"
#pragma ModuleName=Example2

#include "unit-testing"

// Command: run_IGNORE()
// Show the use of optional arguments
// testCase: specify only a single function to execute as Test Case
// name: Name the whole test run

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 TextEncoding="UTF-8"
```

```
#pragma ModuleName=Example3

#include "unit-testing"

// Command: RunTest("example3-plain.ipf")

// Shows the differences between WARN, CHECK and REQUIRE
// The error count this test suite returns is 2
// This can be shown by: print RunTest("example3-plain.ipf")

// 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 "If I'm reached math is wrong !"
End
```

See also Assertion Types.

2.4 example4-wavechecking.ipf

Test suite showing some test assertions Xfor waves.

```
#pragma rtGlobals=3
#pragma TextEncoding="UTF-8"
#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 myWave
    CHECK_WAVE(myWave,TEXT_WAVE)
    CHECK_EQUAL_VAR(DimSize(myWave,0),128)

    Duplicate/T myWave, myWaveCopy
    CHECK_EQUAL_WAVES(myWave,myWaveCopy)
End
```

2.5 example5-extensionhooks.ipf

Two test suites showing how to use test hook overrides.

```
#pragma rtGlobals=3
#pragma TextEncoding="UTF-8"
#pragma ModuleName=Example5

#include "unit-testing"

// To run this example:
// RunTest("example5-extensionhooks.ipf;example5-extensionhooks-otherSuite.ipf")

// Here is shown how own code can be added to the Test Run at certain points.
// In this Test Suite additional code can be executed at the beginning and end
// of Test Cases defined in this procedure aka Test Suite This is done by
// declaring the TEST_CASE_BEGIN_OVERRIDE / TEST_CASE_END_OVERRIDE function
// 'static'

// Be aware that this overrides any global TEST_CASE_BEGIN_OVERRIDE functions
// for this Test Suite If you want to execute the global
// TEST_CASE_BEGIN_OVERRIDE as well add this code: FUNCREF USER_HOOK_PROTO
// tcbegin_global = $"ProcGlobal#TEST_CASE_BEGIN_OVERRIDE" tcbegin_global(name)

// Each hook will output a message starting with >> After the Test Run you can
// see at which points the additional User code was executed.

static Function TEST_CASE_BEGIN_OVERRIDE(name)
    string name

    printf ">> Begin of Test Case %s was extended in this test suite only <<\r", name
End

static Function TEST_CASE_END_OVERRIDE(name)
    string name

    printf ">> End of Test Case %s was extended in this test suite only <<\r", 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
#pragma TextEncoding="UTF-8"

#include "unit-testing"

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

Function TEST_BEGIN_OVERRIDE(name)
    string name

    print ">> The global Test Begin is extended by this output <<"
End

// note: At the point where TEST_END_OVERRIDE is called the Igor Debugger is
// already reset to the state before the Test Run
Function TEST_END_OVERRIDE(name)
    string name

    print ">> The global Test End is extended by this output <<"
End

Function TEST_CASE_END_OVERRIDE(name)
    string name

    print ">> This is the global extension for the End of Test Cases <<"
End

Function TEST_SUITE_BEGIN_OVERRIDE(name)
    string name

    print ">> The Test Suite Begin is globally extended by this output <<"
End

Function TEST_SUITE_END_OVERRIDE(name)
    string name

    print ">> The Test Suite End is globally extended by this output <<"
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 TextEncoding="UTF-8"
#pragma ModuleName=Example6

#include "unit-testing"

// Shows automatic execution of Test Runs
// Command: Call "autorun-test-xxx.bat" from the helper folder
// Details:
// the autorun command script executes Test Runs for all pxp experiment files
// in the current folder open a command line in the example6 folder execute
// "..\..\..\helper\autorun-test-IP7-64bit.bat" for e.g. Igor Pro 7 64 bit.
// Igor Pro is expected to be in the default installation folder.  After the
// run a log file in the example6 folder with the history can be found.

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
#pragma TextEncoding="UTF-8"

#include "unit-testing"

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

2.7 example7-uncaught-aborts.ipf

Test suite showing how unhandled aborts in test cases are handled.

```
#pragma rtGlobals=3
#pragma TextEncoding="UTF-8"
#pragma ModuleName=Example7

#include "unit-testing"

// Command: RunTest("example7-uncaught-aborts.ipf")
// Showing the error message from uncaught aborts in User code
// The Test environment catches such conditions and treats them accordingly
// works with: Abort, AbortOnValue

// PASS() just increases the assertion counter

Function CheckNumber(a)
    variable a

    PASS()

    if(numType(a) == 2)
        Abort
```

```
    endif

    return 1
End

static Function CheckNumber_not_nan()

    CheckNumber(1.0)
End

static Function CheckNumber_nan()

    CheckNumber(NaN)
End
```

2.8 example8-uncaught-runtime-errors.ipf

Test suite showing how runtime errors are treated.

```
#pragma rtGlobals=3
#pragma TextEncoding="UTF-8"
#pragma ModuleName=Example8

#include "unit-testing"

// Command: RunTest("example8-uncaught-runtime-errors.ipf")
// Shows when User code generates an uncaught Runtime Error.
// The test environment catches this condition and gives
// a detailed error message in the history
// The Runtime Error is of course treated as FAIL

Function TestWaveOp()
        WAVE/Z/SDFR=$"I dont exist" wv;
End

// There might be situations where the user wants to catch a runtime error
// (RTE) himself. This function shows how to catch the RTE before the test
// environment handles it. The test environment is controlled manually by
// PASS() and FAIL() PASS() increases the assertion counter and FAIL() treats
// this assertion as fail when a RTE was caught.
//
// Note: As this code can hide critical errors from the test environment it may
// render test runs unreliable.
Function TestWaveOpSelfCatch()
    try
        PASS()
        WAVE/Z/SDFR=$"I dont exist" wv;AbortOnRTE
    catch
        // Do not forget to clear the RTE
        variable err = getRTError(1)
        FAIL()
    endtry
End
```

3 Module Documentation

3.1 Helper functions

Functions

- variable DisableDebugOutput ()
- variable EnableDebugOutput ()
- variable RunTest (string procWinList, string name=defaultValue, string testCase=defaultValue, variable enableJU=defaultValue, variable enableTAP=defaultValue, variable allowDebug=defaultValue, variable keepDataFolder=defaultValue)

3.1.1 Detailed Description

Runner and helper functions.

3.1.2 Function Documentation

3.1.2.1 DisableDebugOutput()

```
variable DisableDebugOutput ( )
```

Turns debug output off.

3.1.2.2 EnableDebugOutput()

```
variable EnableDebugOutput ( )
```

Turns debug output on.

3.1.2.3 RunTest()

```
variable RunTest (
             string procWinList,
             string name = defaultValue,
             string testCase = defaultValue,
             variable enableJU = defaultValue,
             variable enableTAP = defaultValue,
             variable allowDebug = defaultValue,
             variable keepDataFolder = defaultValue )
```

Main function to execute one or more test suites.

Parameters

procWinList	semicolon (";") separated list of procedure files
name	(optional) descriptive name for the executed test suites
testCase	(optional) function name, resembling one test case, which should be executed only for each test suite
enableJU	(optional) enables JUNIT xml output when set to 1
enableTAP	(optional) enables Test Anything Protocol (TAP) output when set to 1
allowDebug	(optional) when set != 0 then the Debugger does not get disabled while running the tests
keepDataFolder	(optional) when set != 0 then the temporary Data Folder where the Test Case is executed in is not removed after the Test Case finishes

Returns

total number of errors

3.2 Test Assertions

Functions

- variable CHECK (variable var)
- variable CHECK_CLOSE_CMPLX (variable/c var1, variable/c var2, variable tol=defaultValue, variable strong_or_weak=defaultValue)
- variable CHECK_CLOSE_VAR (variable var1, variable var2, variable tol=defaultValue, variable strong_or_↩ weak=defaultValue)
- variable CHECK_EMPTY_FOLDER ()
- variable CHECK_EMPTY_STR (string ∗str)
- variable CHECK_EQUAL_STR (string ∗str1, string ∗str2, variable case_sensitive=defaultValue)
- variable CHECK_EQUAL_TEXTWAVES (WaveText wv1, WaveText wv2, variable mode=defaultValue)
- variable CHECK_EQUAL_VAR (variable var1, variable var2)
- variable CHECK_EQUAL_WAVES (WaveOrNull wv1, WaveOrNull wv2, variable mode=defaultValue, variable tol=defaultValue)
- variable CHECK_NEQ_STR (string ∗str1, string ∗str2, variable case_sensitive=defaultValue)
- variable CHECK_NEQ_VAR (variable var1, variable var2)
- variable CHECK_NON_EMPTY_STR (string ∗str)
- variable CHECK_NON_NULL_STR (string ∗str)
- variable CHECK_NULL_STR (string ∗str)
- variable CHECK_PROPER_STR (string ∗str)
- variable CHECK_SMALL_CMPLX (variable/c var, variable tol=defaultValue)
- variable CHECK_SMALL_VAR (variable var, variable tol=defaultValue)
- variable CHECK_WAVE (WaveOrNull wv, variable majorType, variable minorType=defaultValue)
- variable FAIL ()
- variable PASS ()
- variable REQUIRE (variable var)
- variable REQUIRE_CLOSE_CMPLX (variable/c var1, variable/c var2, variable tol=defaultValue, variable strong_or_weak=defaultValue)
- variable REQUIRE_CLOSE_VAR (variable var1, variable var2, variable tol=defaultValue, variable strong_↩ or_weak=defaultValue)
- variable REQUIRE_EMPTY_FOLDER ()
- variable REQUIRE_EMPTY_STR (string ∗str)
- variable REQUIRE_EQUAL_STR (string ∗str1, string ∗str2, variable case_sensitive=defaultValue)
- variable REQUIRE_EQUAL_TEXTWAVES (WaveText wv1, WaveText wv2, variable mode=defaultValue)
- variable REQUIRE_EQUAL_VAR (variable var1, variable var2)
- variable REQUIRE_EQUAL_WAVES (WaveOrNull wv1, WaveOrNull wv2, variable mode=defaultValue, variable tol=defaultValue)
- variable REQUIRE_NEQ_STR (string ∗str1, string ∗str2, variable case_sensitive=defaultValue)
- variable REQUIRE_NEQ_VAR (variable var1, variable var2)
- variable REQUIRE_NON_EMPTY_STR (string ∗str)
- variable REQUIRE_NON_NULL_STR (string ∗str)
- variable REQUIRE_NULL_STR (string ∗str)
- variable REQUIRE_PROPER_STR (string ∗str)
- variable REQUIRE_SMALL_CMPLX (variable/c var, variable tol=defaultValue)
- variable REQUIRE_SMALL_VAR (variable var, variable tol=defaultValue)
- variable REQUIRE_WAVE (WaveOrNull wv, variable majorType, variable minorType=defaultValue)
- variable WARN (variable var)
- variable WARN_CLOSE_CMPLX (variable/c var1, variable/c var2, variable tol=defaultValue, variable strong↩ _or_weak=defaultValue)
- variable WARN_CLOSE_VAR (variable var1, variable var2, variable tol=defaultValue, variable strong_or_↩ weak=defaultValue)
- variable WARN_EMPTY_FOLDER ()
- variable WARN_EMPTY_STR (string ∗str)

- variable WARN_EQUAL_STR (string *str1, string *str2, variable case_sensitive=defaultValue)
- variable WARN_EQUAL_TEXTWAVES (WaveText wv1, WaveText wv2, variable mode=defaultValue)
- variable WARN_EQUAL_VAR (variable var1, variable var2)
- variable WARN_EQUAL_WAVES (WaveOrNull wv1, WaveOrNull wv2, variable mode=defaultValue, variable tol=defaultValue)
- variable WARN_NEQ_STR (string *str1, string *str2, variable case_sensitive=defaultValue)
- variable WARN_NEQ_VAR (variable var1, variable var2)
- variable WARN_NON_EMPTY_STR (string *str)
- variable WARN_NON_NULL_STR (string *str)
- variable WARN_NULL_STR (string *str)
- variable WARN_PROPER_STR (string *str)
- variable WARN_SMALL_CMPLX (variable/c var, variable tol=defaultValue)
- variable WARN_SMALL_VAR (variable var, variable tol=defaultValue)
- variable WARN_WAVE (WaveOrNull wv, variable majorType, variable minorType=defaultValue)

3.2.1 Detailed Description

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

3.2.2 Function Documentation

3.2.2.1 CHECK()

```
variable CHECK (
             variable var )
```

Tests if var is true (1).

Parameters

var	variable to test

3.2.2.2 CHECK_CLOSE_CMPLX()

```
variable CHECK_CLOSE_CMPLX (
             variable/c var1,
             variable/c var2,
             variable tol = defaultValue,
             variable strong_or_weak = defaultValue )
```

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

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

Variant for complex numbers.

3.2.2.3 CHECK_CLOSE_VAR()

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

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

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

3.2.2.4 CHECK_EMPTY_FOLDER()

```
variable CHECK_EMPTY_FOLDER ( )
```

Tests if the current data folder is empty.

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

3.2.2.5 CHECK_EMPTY_STR()

```
variable CHECK_EMPTY_STR (
             string * str )
```

Tests if str is empty.

A null string is never empty.

Parameters

str	string to test

3.2.2.6 CHECK_EQUAL_STR()

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

Compares two strings for equality.

Parameters

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

3.2.2.7 CHECK_EQUAL_TEXTWAVES()

```
variable CHECK_EQUAL_TEXTWAVES (
             WaveText wv1,
             WaveText wv2,
             variable mode = defaultValue )
```

Tests two text waves for equality.

Parameters

wv1	first text wave, can be invalid for Igor Pro 7 or later
wv2	second text wave, can be invalid for Igor Pro 7 or later
mode	(optional) features of the waves to compare, defaults to all modes, defined at Wave equality flags

3.2.2.8 CHECK_EQUAL_VAR()

```
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 fullfill semantic correctness this assertion treats two variables with both holding NaN as equal.

Parameters

var1	first variable
var2	second variable

3.2.2.9 CHECK_EQUAL_WAVES()

```
variable CHECK_EQUAL_WAVES (
             WaveOrNull wv1,
             WaveOrNull wv2,
             variable mode = defaultValue,
             variable tol = defaultValue )
```

Tests two waves for equality.

Parameters

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

3.2.2.10 CHECK_NEQ_STR()

```
variable CHECK_NEQ_STR (
             string * str1,
             string * str2,
             variable case_sensitive = defaultValue )
```

Compares two strings for unequality.

Parameters

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

3.2.2.11 CHECK_NEQ_VAR()

```
variable CHECK_NEQ_VAR (
             variable var1,
             variable var2 )
```

Tests two variables for inequality.

Parameters

var1	first variable
var2	second variable

3.2.2.12 CHECK_NON_EMPTY_STR()

```
variable CHECK_NON_EMPTY_STR (
             string * str )
```

Tests if str is not empty.

A null string is a non empty string too.

Parameters

str	string to test

3.2.2.13 CHECK_NON_NULL_STR()

```
variable CHECK_NON_NULL_STR (
             string * str )
```

Tests if str is not null.

An empty string is always non null.

Parameters

str	string to test

3.2.2.14 CHECK_NULL_STR()

```
variable CHECK_NULL_STR (
             string * str )
```

Tests if str is null.

An empty string is never null.

Parameters

str	string to test

3.2.2.15 CHECK_PROPER_STR()

```
variable CHECK_PROPER_STR (
             string * str )
```

Tests if str is a "proper" string, i.e. a string with a length larger than zero.

Neither null strings nor empty strings are proper strings.

Parameters

str	string to test

3.2.2.16 CHECK_SMALL_CMPLX()

```
variable CHECK_SMALL_CMPLX (
             variable/c var,
             variable tol = defaultValue )
```

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

Parameters

var	variable
tol	(optional) tolerance, defaults to 1e-8

Variant for complex numbers

3.2.2.17 CHECK_SMALL_VAR()

```
variable CHECK_SMALL_VAR (
             variable var,
             variable tol = defaultValue )
```

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

Parameters

var	variable
tol	(optional) tolerance, defaults to 1e-8

3.2.2.18 CHECK_WAVE()

```
variable CHECK_WAVE (
             WaveOrNull wv,
             variable majorType,
             variable minorType = defaultValue )
```

Tests a wave for existence and its type.

Parameters

wv	wave reference
majorType	major wave type
minorType	(optional) minor wave type

See also

Wave existence flags

3.2.2.19 FAIL()

```
variable FAIL ( )
```

Force the test case to fail.

3.2.2.20 PASS()

```
variable PASS ( )
```

Increase the assertion counter only.

3.2.2.21 REQUIRE()

```
variable REQUIRE (
            variable var )
```

3.2.2.22 REQUIRE_CLOSE_CMPLX()

```
variable REQUIRE_CLOSE_CMPLX (
            variable/c var1,
            variable/c var2,
            variable tol = defaultValue,
            variable strong_or_weak = defaultValue )
```

3.2.2.23 REQUIRE_CLOSE_VAR()

```
variable REQUIRE_CLOSE_VAR (
            variable var1,
            variable var2,
            variable tol = defaultValue,
            variable strong_or_weak = defaultValue )
```

3.2.2.24 REQUIRE_EMPTY_FOLDER()

```
variable REQUIRE_EMPTY_FOLDER ( )
```

3.2.2.25 REQUIRE_EMPTY_STR()

```
variable REQUIRE_EMPTY_STR (
            string * str )
```

3.2.2.26 REQUIRE_EQUAL_STR()

```
variable REQUIRE_EQUAL_STR (
            string * str1,
            string * str2,
            variable case_sensitive = defaultValue )
```

3.2.2.27 REQUIRE_EQUAL_TEXTWAVES()

```
variable REQUIRE_EQUAL_TEXTWAVES (
            WaveText wv1,
            WaveText wv2,
            variable mode = defaultValue )
```

3.2.2.28 REQUIRE_EQUAL_VAR()

```
variable REQUIRE_EQUAL_VAR (
             variable var1,
             variable var2 )
```

3.2.2.29 REQUIRE_EQUAL_WAVES()

```
variable REQUIRE_EQUAL_WAVES (
             WaveOrNull wv1,
             WaveOrNull wv2,
             variable mode = defaultValue,
             variable tol = defaultValue )
```

3.2.2.30 REQUIRE_NEQ_STR()

```
variable REQUIRE_NEQ_STR (
             string * str1,
             string * str2,
             variable case_sensitive = defaultValue )
```

3.2.2.31 REQUIRE_NEQ_VAR()

```
variable REQUIRE_NEQ_VAR (
             variable var1,
             variable var2 )
```

3.2.2.32 REQUIRE_NON_EMPTY_STR()

```
variable REQUIRE_NON_EMPTY_STR (
             string * str )
```

3.2.2.33 REQUIRE_NON_NULL_STR()

```
variable REQUIRE_NON_NULL_STR (
             string * str )
```

3.2.2.34 REQUIRE_NULL_STR()

```
variable REQUIRE_NULL_STR (
             string * str )
```

3.2.2.35 REQUIRE_PROPER_STR()

```
variable REQUIRE_PROPER_STR (
             string * str )
```

3.2.2.36 REQUIRE_SMALL_CMPLX()

```
variable REQUIRE_SMALL_CMPLX (
             variable/c var,
             variable tol = defaultValue )
```

3.2.2.37 REQUIRE_SMALL_VAR()

```
variable REQUIRE_SMALL_VAR (
             variable var,
             variable tol = defaultValue )
```

3.2.2.38 REQUIRE_WAVE()

```
variable REQUIRE_WAVE (
            WaveOrNull wv,
            variable majorType,
            variable minorType = defaultValue )
```

3.2.2.39 WARN()

```
variable WARN (
            variable var )
```

3.2.2.40 WARN_CLOSE_CMPLX()

```
variable WARN_CLOSE_CMPLX (
            variable/c var1,
            variable/c var2,
            variable tol = defaultValue,
            variable strong_or_weak = defaultValue )
```

3.2.2.41 WARN_CLOSE_VAR()

```
variable WARN_CLOSE_VAR (
            variable var1,
            variable var2,
            variable tol = defaultValue,
            variable strong_or_weak = defaultValue )
```

3.2.2.42 WARN_EMPTY_FOLDER()

```
variable WARN_EMPTY_FOLDER ( )
```

3.2.2.43 WARN_EMPTY_STR()

```
variable WARN_EMPTY_STR (
            string * str )
```

3.2.2.44 WARN_EQUAL_STR()

```
variable WARN_EQUAL_STR (
            string * str1,
            string * str2,
            variable case_sensitive = defaultValue )
```

3.2.2.45 WARN_EQUAL_TEXTWAVES()

```
variable WARN_EQUAL_TEXTWAVES (
            WaveText wv1,
            WaveText wv2,
            variable mode = defaultValue )
```

3.2.2.46 WARN_EQUAL_VAR()

```
variable WARN_EQUAL_VAR (
            variable var1,
            variable var2 )
```

3.2.2.47 WARN_EQUAL_WAVES()

```
variable WARN_EQUAL_WAVES (
            WaveOrNull wv1,
```

```
            WaveOrNull wv2,
            variable mode = defaultValue,
            variable tol = defaultValue )
```

3.2.2.48 WARN_NEQ_STR()

```
variable WARN_NEQ_STR (
            string * str1,
            string * str2,
            variable case_sensitive = defaultValue )
```

3.2.2.49 WARN_NEQ_VAR()

```
variable WARN_NEQ_VAR (
            variable var1,
            variable var2 )
```

3.2.2.50 WARN_NON_EMPTY_STR()

```
variable WARN_NON_EMPTY_STR (
            string * str )
```

3.2.2.51 WARN_NON_NULL_STR()

```
variable WARN_NON_NULL_STR (
            string * str )
```

3.2.2.52 WARN_NULL_STR()

```
variable WARN_NULL_STR (
            string * str )
```

3.2.2.53 WARN_PROPER_STR()

```
variable WARN_PROPER_STR (
            string * str )
```

3.2.2.54 WARN_SMALL_CMPLX()

```
variable WARN_SMALL_CMPLX (
            variable/c var,
            variable tol = defaultValue )
```

3.2.2.55 WARN_SMALL_VAR()

```
variable WARN_SMALL_VAR (
            variable var,
            variable tol = defaultValue )
```

3.2.2.56 WARN_WAVE()

```
variable WARN_WAVE (
            WaveOrNull wv,
            variable majorType,
            variable minorType = defaultValue )
```

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 DATAFOLDER_WAVE = 0x100
- const variable DOUBLE_WAVE = 0x04
- const variable FLOAT_WAVE = 0x02
- const variable INT16_WAVE = 0x10
- const variable INT32_WAVE = 0x20
- const variable INT64_WAVE = 0x80
- const variable INT8_WAVE = 0x08
- const variable NUMERIC_WAVE = 1
- const variable TEXT_WAVE = 2
- const variable UNSIGNED_WAVE = 0x40
- const variable WAVE_WAVE = 0x4000

3.4.1 Detailed Description

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

3.4.2 Variable Documentation

3.4.2.1 COMPLEX_WAVE

```
const variable COMPLEX_WAVE = 0x01
```

3.4.2.2 DATAFOLDER_WAVE

```
const variable DATAFOLDER_WAVE = 0x100
```

3.4.2.3 DOUBLE_WAVE

```
const variable DOUBLE_WAVE = 0x04
```

3.4.2.4 FLOAT_WAVE

```
const variable FLOAT_WAVE = 0x02
```

3.4.2.5 INT16_WAVE

```
const variable INT16_WAVE = 0x10
```

3.4.2.6 INT32_WAVE

```
const variable INT32_WAVE = 0x20
```

3.4.2.7 INT64_WAVE

```
const variable INT64_WAVE = 0x80
```

3.4.2.8 INT8_WAVE

```
const variable INT8_WAVE = 0x08
```

3.4.2.9 NUMERIC_WAVE

```
const variable NUMERIC_WAVE = 1
```

3.4.2.10 TEXT_WAVE

```
const variable TEXT_WAVE = 2
```

3.4.2.11 UNSIGNED_WAVE

```
const variable UNSIGNED_WAVE = 0x40
```

3.4.2.12 WAVE_WAVE

```
const variable WAVE_WAVE = 0x4000
```

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

3.5.2.1 DATA_FULL_SCALE

```
const variable DATA_FULL_SCALE = 256
```

3.5.2.2 DATA_UNITS

```
const variable DATA_UNITS = 8
```

3.5.2.3 DIMENSION_LABELS

```
const variable DIMENSION_LABELS = 32
```

3.5.2.4 DIMENSION_SIZES

```
const variable DIMENSION_SIZES = 512
```

3.5.2.5 DIMENSION_UNITS

```
const variable DIMENSION_UNITS = 16
```

3.5.2.6 WAVE_DATA

```
const variable WAVE_DATA = 1
```

3.5.2.7 WAVE_DATA_TYPE

```
const variable WAVE_DATA_TYPE = 2
```

3.5.2.8 WAVE_LOCK_STATE

```
const variable WAVE_LOCK_STATE = 128
```

3.5.2.9 WAVE_NOTE

```
const variable WAVE_NOTE = 64
```

3.5.2.10 WAVE_SCALING

```
const variable WAVE_SCALING = 4
```

Index

T

U

W