Igor Unit Testing Framework Documentation

Release (master)

UnitTestingFramework-v1.07-1-g8839d8c

byte-physics.de

Sep 17, 2018

CONTENTS

1 Reference **3**
- 1.1 What is a Unit Testing Framework? . . . 3
- 1.2 Guided Tour . . . 4
- 1.3 Basic Structure . . . 8
- 1.4 Advanced Usage . . . 11
- 1.5 Examples . . . 14
- 1.6 Code Documentation . . . 25

Index **33**

This package empowers a programmer to utilize unit testing for Igor Pro procedures and XOPs. If you do not yet know about unit tests, start by reading the introduction on *What is a Unit Testing Framework?*.

There is a *Guided Tour* that will get you started on-the-fly. If you prefer, you can skip to read about *Basic Structure* or *Advanced Usage*. Some may also find it useful to skip directly to the *Examples*.

CHAPTER

ONE

REFERENCE

1.1 What is a Unit Testing Framework?

The purpose of every program is to ensure that a specific task is performed reliably in a defined matter. Therefore, programming is all about testing and quality control of the produced source code. These two workflow tasks are entirely optional but are especially important when it comes to hazard and risk-sensitive tasks, as well as security-relevant features of software with critical to catastrophic consequences. More generally speaking, it contributes to a clean, professional look and better working experience if software works in a defined way and unit tests help to define this way.

1.1.1 Testing

A program gets tested in various ways during development: A first test usually involves the syntactic correctness and the correct usage of external libraries. It ensures that the program compiles and that it produces output for a given task. Complex scenarios typically afford a much larger codebase and a more profound investigation of the involved interfaces. The more complex the scenarios a program can handle, the more time is involved in its production. Therefore it is crucial to define the program's interface to indicate what it is capable of, and what not, to prevent it getting used in the wrong context.

One standard in quality control is the four-eyes-check by two persons. Writing professional code in a lean and agile, continuous delivery software environment, usually involves this additional peer review step. The review step is an attempt to separate code production, and testing to separate persons as the perception of the tester adds valuable input to the code leading to quicker deployment of quality software.

A review typically involves testing the functionality of the code output for different inputs. These tests are equally performed during code production and review stages. The problem, this review step is targetting onto, is that a programmer typically does not think of all critical test situations. The tester, in turn, does not know about the code and its context and therefore the reviewer needs time to understand the context of the program. In an attempt to save valuable time, review and code production have to be based on a definition for the produced functionality which can be for example the creation of a valid file format. Such a definition allows the tester to perform tests without necessarily needing to hack into the code base. Defining these tests somewhere records the current functionality of the program and protects it against changes.

Even though, the review process guarantees a higher level of quality, the additional assessment requires an assignment of double the developing resources and those resources are usually considered precious. In this context, automated test environments minimize production time and ensure a consistent level of quality. This level of quality can then consistently get maintained over time when further changes are introduced to the unit.

1.1.2 Unit Tests

To be able to perform automated tests, the code is typically organized in functional units. A unit is a part of software inside a project that performs a particular task. Typically this unit is isolated and runs on a linearly independent path inside the code. The unit communicates via an interface which accepts inputs and produces outputs.

```
          ########
input --> # unit # --> output
          ########
```

In the most simple case, a unit is a function. The parameters which get passed to the function define the input interface, and the return value is the output interface. In a more complex scenario, such a unit could be responsible for converting one file to another format.

A unit can be checked for valid output by defining a *suite of tests*. The test suite is further grouped into atomically small tests which are called *Test Cases*. A test case typically checks that an entity fulfills specific properties and a unit produces valid output for a given input. Within these checks, the result of defined inputs is compared against defined outputs. The comparisons are performed using different types of *Assertions*. As long as all test cases inside a test suite are executed correctly, the tested functionality of the unit is maintained. Performing these checks on a regular basis also ensures that a consistent level of quality and a defined functionality is maintained upon changes to the code.

1.1.3 Agile Development

When using version control systems like git, the introduced changes are typically tested with test pipelines before applying the changes by using apps like jenkins or gitlab. These automated tests introduce a step prior to the review process which makes the review more clear and transparent and allow a quicker code review. This Framework enables unit tests for continuous integration and continuous delivery environments in Igor Pro. Do not hesitate to contact us if you need further assistance in creating a professional CI/CD workflow for your Igor Pro project to ensure a higher level of quality in your code.

1.2 Guided Tour

To visualize the functionality of the unit testing framework, we will start with a guided tour in which we create our first unit and test it with the unit testing framework. The tour will cover the following steps:

- *Creating a unit*
- *Testing the unit*
- *Executing the test*
- *Extending the test*

Please make sure that the framework has been properly installed if you wish to follow the guide. For the framework to work, the files from the procedures folder should be placed into the *User Procedures* Folder of your Igor Pro setup.

1.2.1 Creating a unit

We will start by creating a simple unit.

The following formula gives the diameter d of a carbon nanotube:

$$d = \frac{a_0}{\pi} \cdot \sqrt{n^2 + m^2 + nm}$$

The natural numbers n and m define the carbon nanotube type. a_0 is the unit cell lattice constant of graphene (Understanding the background of the above formula is not required here).

The formula is easily translated into Igor Pro code:

Listing 1: Procedure

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

// calculate carbon nanotube diameters
Function diameter(n, m)
    Variable n, m

    return 0.144 / 3.1415 * (3 * (n^2 + n*m + m^2))^(0.5)
End
```

1.2.2 Testing the unit

If we want to rely on this formula with other calculations, we have to test if the output of this function is both correct and within our required accuracy range. To perform these two tests, we define a *Test Case*.

Listing 2: test0

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

#include "unit-testing"

Function testDiameter()
    // the (6,5) type is 0.757nm in diameter
    REQUIRE_CLOSE_VAR(diameter(6, 5), 0.757, tol=1e-3)
    // this is the same value as for the (9,1) type.
    REQUIRE_EQUAL_VAR(diameter(6, 5), diameter(9, 1))
End
```

The test case `testDiameter` contains two checks. Both are required to pass the test suite. In the context of this framework we will refer to them as *assertions*. The first assertion `REQUIRE_CLOSE_VAR` compares the two floating point numbers within the given tolerance of 0.001nm. The second `REQUIRE_EQUAL_VAR` uses a mathematical peculiarity of the above formula to check if the calculation gives correct output.

The test case function can be placed anywhere inside the main procedure file, but it can be considered good practice to separate test cases into a procedure file of their own. Such a separate procedure file that only contains test cases is called a *Test Suite*. A test suite can for example perform all the necessary tests for a unit.

1.2.3 Executing the test

To execute the test suite we use the *`RunTest()`* directive. It accepts the name of our test suite (the procedure window) as an argument. In our example we have named the procedure window `"test0"`.

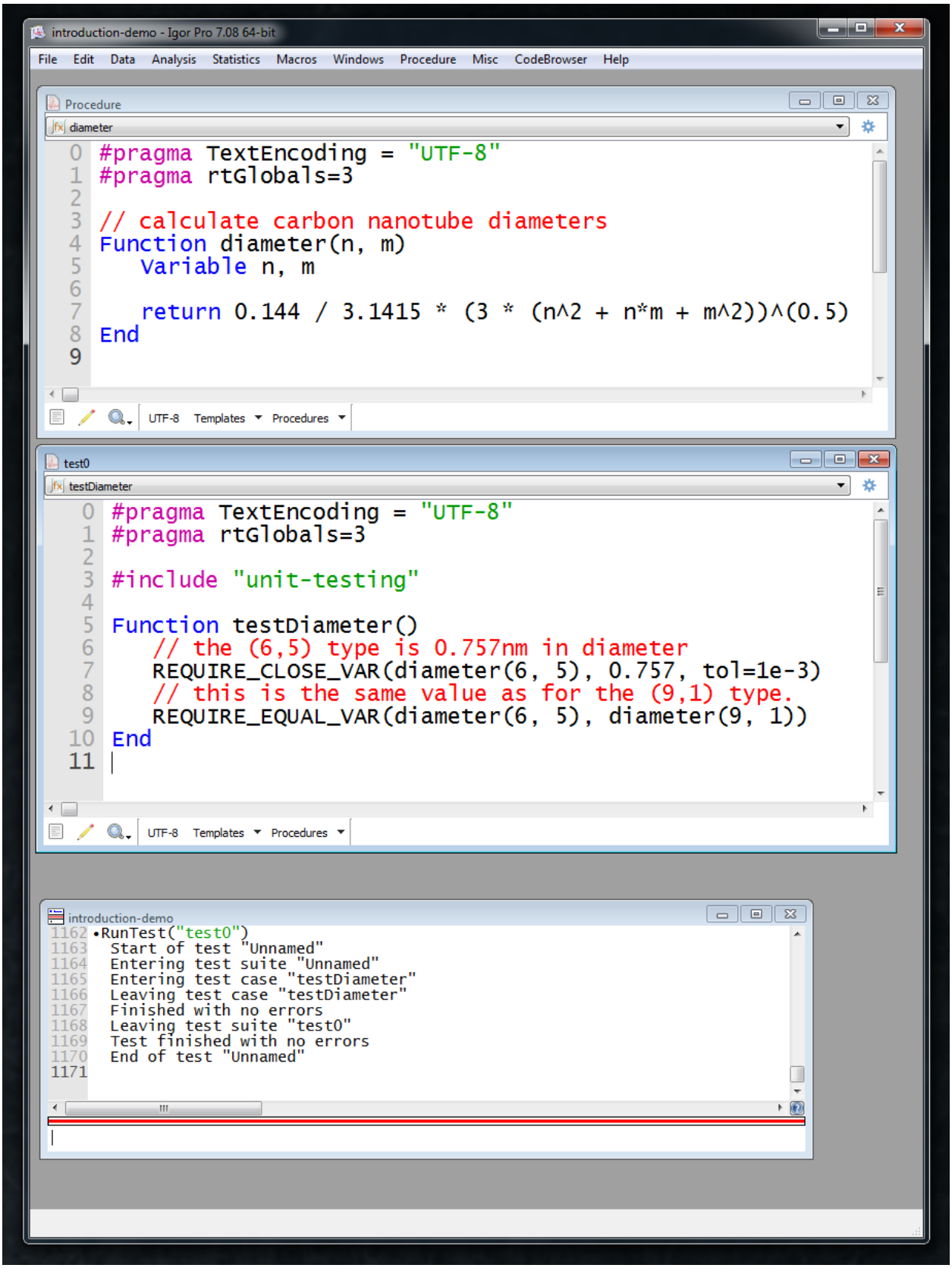
```
•RunTest("test0")
  Start of test "Unnamed"
  Entering test suite "Unnamed"
  Entering test case "testDiameter"
  Leaving test case "testDiameter"
```

(continues on next page)

(continued from previous page)

```
Finished with no errors
Leaving test suite "test0"
Test finished with no errors
End of test "Unnamed"
```

In the cosole output above, the highlighted line indicates that all tests within the current test suite have passed successfully. The unit is working properly. The full Igor Pro environment with our unit test should look like this:


introduction-demo - Igor Pro 7.08 64-bit
File Edit Data Analysis Statistics Macros Windows Procedure Misc CodeBrowser Help
Procedure
diameter
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3
// calculate carbon nanotube diameters
Function diameter(n, m)
Variable n, m
return 0.144 / 3.1415 * (3 * (n^2 + n*m + m^2))^(0.5)
End
UTF-8 Templates Procedures
test0
testDiameter
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3
#include "unit-testing"
Function testDiameter()
// the (6,5) type is 0.757nm in diameter
REQUIRE_CLOSE_VAR(diameter(6, 5), 0.757, tol=1e-3)
// this is the same value as for the (9,1) type.
REQUIRE_EQUAL_VAR(diameter(6, 5), diameter(9, 1))
End
UTF-8 Templates Procedures
introduction-demo
•RunTest("test0")
Start of test "Unnamed"
Entering test suite "Unnamed"
Entering test case "testDiameter"
Leaving test case "testDiameter"
Finished with no errors
Leaving test suite "test0"
Test finished with no errors
End of test "Unnamed"

1.2.4 Extending the test

Note, that we have defined a test case for the current capabilities of our function `diameter()`. The calculation is only exact up to the specified error range. The high error is caused by a fixated value of `pi=3.1415`. To emphasize this, we can add an assertion to the test case that will fail but will not affect the error counter. Such an assertion is done with a *WARN_** directive. Every *REQUIRE_** assertion also has a *WARN_** variant, see:ref:*AssertionTypes* for a summary.

```
Function testDiameter()
    // the (6,5) type is 0.757nm in diameter
    REQUIRE_CLOSE_VAR(diameter(6, 5), 0.757, tol=1e-3)
    // this is the same value as for the (9,1) type.
    REQUIRE_EQUAL_VAR(diameter(6, 5), diameter(9, 1))
    // warn if accuracy is not exact
    WARN_CLOSE_VAR(diameter(6, 5), 0.7573453, tol=1e-7)
End
```

The output of *RunTest()* will now include a warning assertion without failing the test case:

```
•RunTest("test0")
  Start of test "Unnamed"
  Entering test suite "Unnamed"
  Entering test case "testDiameter"
  Entering test case "testDiameter"
  0.757368 ~ 0.757345 with strong check and tol 1e-07: is false
  Assertion "WARN_CLOSE_VAR(diameter(6, 5), 0.7573453, tol=1e-7)" failed in line 11,␣
↪procedure "test0"
  Leaving test case "testDiameter"
  Finished with no errors
  Leaving test suite "test0"
  Test finished with no errors
  End of test "Unnamed"
```

If the program should be extended to a higher level of accuracy, this warning can be set to the corresponding *REQUIRE* assertion. The program `diameter` then has to be changed to reflect the new requirement. In the current example, *pi* would need to be used instead of only a handful of decimal places hardcoded.

In a test-driven workflow, the unit tests get extended before even changing anything at the code base. Defining the test case prior to any code production assures that the software development is not producing unnecessary (and untested) code.

A more elaborate example for defining a peak find functionality can be found in the *examples section*. For a quick start, also have a look at the *first example*.

1.3 Basic Structure

The interface design and naming is inspired by the Boost Test Library. Following this naming scheme, the unit testing package consists of three basic structural elements:

- *Test Suites*
- *Test Cases*
- *Assertions*

The basic building blocks of this unit testing framework are assertions. Assertions are used for checking if a condition is true. See *Assertion Types* for a clarification of the difference between the three assertion types. Assertions are grouped into single test cases and test cases are organized in test suites.

A *test suite* is a group of test cases that live in a single procedure file. You can group multiple test suites in a named test environment by using the optional parameter `name` of *`RunTest()`*.

For a list of all objects see genindex or use the search.

1.3.1 Test Run

A Test Run is executed using *`RunTest()`* with only a single mandatory parameter which is the *Test Suite*.

Function definition of RunTest

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

Main function to execute test suites with the unit testing framework.

Listing 3: usage example

```
RunTest("proc0;proc1", name="myTest")
```

This command will run the test suites *proc0* and *proc1* in a test named *myTest*.

Return total number of errors

Parameters

- `procWinList`: A list of procedure files that should be treated as test suites. The list should be given semicolon (";") separated.

  The procedure name must not include Independent Module specifications.

  This parameter can be given as a regular expression with enableRegExp set to 1.

- `name`: (optional) default "Unnamed" descriptive name for the executed test suites. This can be used to group multiple test suites into a single test run.

- `testCase`: (optional) default ".*" (all test cases in the list of test suites) function names, resembling test cases, which should be executed in the given list of test suites (procWinList).

  The list should be given semicolon (";") separated.

  This parameter can be treated as a regular expression with enableRegExp set to 1.

- `enableJU`: (optional) default disabled, enabled when set to 1: A JUNIT compatible XML file is written at the end of the Test Run. It allows the combination of this framework with continuous integration servers like Atlassian Bamboo.

- `enableTAP`: (optional) default disabled, enabled when set to 1: A TAP compatible file is written at the end of the test run.

  Test Anything Protocol (TAP) standard 13

- `enableRegExp`: (optional) default disabled, enabled when set to 1: The input for test suites (procWinList) and test cases (testCase) is treated as a regular expression.

Listing 4: Example

```
RunTest("example[1-3]-plain\\.ipf", enableRegExp=1)
```

This command will run all test cases in the following test suites:

- *example1-plain.ipf*
- example2-plain.ipf
- *example3-plain.ipf*

- `allowDebug`: (optional) default disabled, enabled when set to 1: The Igor debugger will be left in its current state when running the tests.
- `keepDataFolder`: (optional) default disabled, enabled when set to 1: The temporary data folder wherein each test case is executed is not removed at the end of the test case. This allows to review the produced data.

1.3.2 Test Suite

A Test Suite is a group of *Test Cases* which should belong together. All *test functions* are defined in a single procedure file. Generally speaking, a Test Suite is equal to a procedure file. Therefore tests suites can not be nested, although multiple test suites can be run with one command by supplying a list to the parameter `procWinList` in *`RunTest()`*.

Note: Although possible, a test suite should not live inside the main program. It should be separated from the rest of the project into its own procedure file. This also allows to load only the necessary parts of your program into the unit test.

1.3.3 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:

1. It takes no parameters.
2. Its name does not end with *_IGNORE*.
3. It is either non-static, or static and part of a regular module.

The first rule is making the test case callable in automated test environments.

The second rule is reserving the *_IGNORE* namespace to allow 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. This will keep the Test Cases in their own namespace and thus not interfere with user-defined functions in *ProcGlobal*.

A defined list of test cases in a test suite can be run using the optional parameter `testCase` of *`RunTest()`*. When executing multiple test suites and a test case is found in more than one test suite, it is executed in every matching test suite.

Example:

In Test Suite *TestSuite_1.ipf* the Test Cases *static Duplicate()* and *static Unique_1()* are defined. In Test Suite *TestSuite_2.ipf* the Test Cases *static Duplicate()*, *static Unique_2()* are defined.

```
Runtest("TestSuite_1.ipf;TestSuite_2.ipf", testCase="Unique_1;Unique_2;Duplicate")
```

The command will run the two test suites *TestSuite_1.ipf* and *TestSuite_2.ipf* separately. Within every test suites two test cases are execute: the *Unique** test case and the *Duplicate* test case. The *Duplicate* test cases do not interfere with each other since they are static to the corresponding procedure files. Since the duplicate test cases are found in both test suites, they are also executed in both.

Note: The Test Run will not execute if the one of the specified test cases can not be found in the given list of test suites. This is also applies if no test case could be found using a regular expression pattern.

1.3.4 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. The assertion group is specified with a prefix to the assertion name using one of *WARN*, *CHECK* or *REQUIRE*. Assertions usually come in these triplets which differ only in how they react on a failed assertion. The following table clarifies the difference between the three assertion prefix groups:

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

The most simple assertion is *CHECK()* which tests if its argument is true. If you do not want to increase the error count, you could use the corresponding *WARN()* function and if you want to Abort the execution of the current test case if the supplied argument is false, you can use the *REQUIRE()* variant for this.

Similar to these simple assertions there are many different checks for typical use cases. Comparing two variables, for example, can be done with *WARN_EQUAL_VAR()*, or *REQUIRE_EQUAL_VAR()*. Take a look at *Example10* for a test case with various assertions.

Note: See *Assertions* for a complete list of all available checks. If in doubt use the *CHECK* variant. Only the `CHECK_*` variants are documented, as the interface for `REQUIRE_*` and `WARN_*` is equivalent.

Assertions with only one variant are *PASS()* and *FAIL()*. If you want to know more about how to use these two special assertions, take a look at *Example7*.

1.4 Advanced Usage

1.4.1 Test Hooks

A Test Run can be extended with user-defined code at specific points during its execution. These pre-defined injection points are at the beginning and respectively at the end of a complete *Test Run*, a *Test Suite*, and a *Test Case*.

The following functions are reserved for user code injections:

`TEST_BEGIN_OVERRIDE``()`

Executed at the **begin** of a *Test Run*.

TEST_END_OVERRIDE()

Executed at the **end** of a *Test Run*.

TEST_SUITE_BEGIN_OVERRIDE()

Executed at the **begin** of a *Test Suite*.

TEST_SUITE_END_OVERRIDE()

Executed at the **end** of a *Test Suite*.

TEST_CASE_BEGIN_OVERRIDE()

Executed at the **begin** of a *Test Case*.

TEST_CASE_END_OVERRIDE()

Executed at the **end** of a *Test Case*.

Note: *TEST_END_OVERRIDE()* is executed at the very end of a test run so that the Igor debugger state is already reset to the state it had before *RunTest()* was executed.

Note: The functions *TEST_SUITE_BEGIN_OVERRIDE()* and *TEST_SUITE_END_OVERRIDE()* as well as *TEST_CASE_BEGIN_OVERRIDE()* and *TEST_CASE_END_OVERRIDE()* can also be defined locally in a test suite with the *static* keyword. example2 shows how *static* functions are called the framework.

These functions are executed automatically if they are defined anywhere in global or local context. For example, *TEST_CASE_BEGIN_OVERRIDE()* gets executed at the beginning of each *Test Case*. Locally defined functions always override globally defined ones of the same name. To visualize this behavior, take a look at the following scenario: A user would like to have code executed only in a specific *Test Suite*. Then the functions *TEST_SUITE_BEGIN_OVERRIDE()* and *TEST_SUITE_END_OVERRIDE()* can be defined locally within the current *Test Suite* by declaring them *static* to the current Test Suite. The local (*static*) functions then replace any previously defined global functions. The functionality with additional user code at certain points of a Test Run is demonstrated in *Example5*.

Note: If the locally defined function should only extend a global function the user can call the global function within the local function as follows:

```
FUNCREF USER_HOOK_PROTO tcbegin_global = TEST_CASE_BEGIN_OVERRIDE
tcbegin_global(TestCaseName)
```

To give a possible use case, take a look at the following scenario: By default, each *Test Case* is executed in its own temporary data folder. *TEST_CASE_BEGIN_OVERRIDE()* can be used to set the data folder to *root:*. This will result that each Test Case gets executed in *root:* and no cleanup is done afterward. The *next* Test Case then starts with the data the *previous* Test Case left in *root:*.

Note: By default the Igor debugger is disabled during the execution of a test run.

1.4.2 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.

1.4.3 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 preceeding the function of a Test Case:

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

For directives two additional keywords 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 ignored by a TAP consumer.
- *SKIP* indicates a Test that should be skipped. A Test with this directive keyword 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.

1.4.4 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* and the script *helper/autorun-test.bat* into its own folder.
- Run the batch file *autorun-test.bat*.
- Inspect the created log file.

The example batch files for autorun create a file named *DO_AUTORUN.TXT* before starting Igor Pro. This enables autorun mode. After the *run()* function is executed and returned the log is saved in a file on disk and Igor Pro quits.

A different autorun mode is enabled if the file is named *DO_AUTORUN_PLAIN.TXT*. In this mode no log file is saved after the test execution and Igor Pro does not quit. This mode also does not use the Operation Queue.

See also *Example6*.

1.4.5 Running in an Independent Module

The unit-testing framework can be run itself in an independent module. This can be required in very rare cases when the *ProcGlobal* procedures might not always be compiled.

See also *Example9*.

1.4.6 Handling of Abort Code

The unit-testing framework continues with the next test case after catching *Abort* and logs the abort code. Currently differentiation of different abort conditions include manual user aborts, stack overflow and an encountered *Abort* in the code. The framework is terminated when manually pressing the Abort button.

Note: Igor Pro 6 can not differentiate between manual user aborts and programmatic abort codes. Pressing the Abort button in Igor Pro 6 will therefore terminate only the current test case and continue with the next queued test case.

1.5 Examples

The example section shows the usage of the Igor Unit Testing Framework. If you are just starting to use this framework, consider taking the *Guided Tour*.

1.5.1 Example1

This example is showing the basic working principle of the compare assertion. Constant values are given as input to the unit `abs()` and the output is checked for equality.

This unit test makes sure that the function `abs()` behaves as expected. For example if you use the unit `abs()` in a function and you give `NaN` as an input value the output value will also be `NaN`. The function is also capable of handling `INF` singularities.

Listing 5: example1-plain.ipf

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

#include "unit-testing"

Function TestAbs()

        CHECK_EQUAL_VAR(abs(1.5), 1.5)
        CHECK_EQUAL_VAR(abs(-1.5), 1.5)
        CHECK_EQUAL_VAR(abs(NaN), NaN)
        WARN(abs(NaN) == NaN)
        CHECK_EQUAL_VAR(abs(INF), INF)
        CHECK_EQUAL_VAR(abs(-INF), INF)
End
```

The test suite can be executed using the following command:

Listing 6: command

```
RunTest("example1-plain.ipf")
```

By looking at line 10 in this example it becomes clear that *CHECK_EQUAL_VAR()* is a better way of comparing numeric variables then the plain *CHECK()* assertion since `NaN == NaN` is false. The error is skipped by using the *WARN()* variant and will not raise the error counter. If you want to know up to what extend those methods differ, take a look at the section on *Assertion Types* .

Note: It is recommended to take a look at the *complete list of assertions*. This will help in choosing the right assertion type for a comparison.

The definition for the assertions in this test suite:

- *CHECK_EQUAL_VAR()*
- *WARN()*

1.5.2 Example2

This test suite has its own run routine. The `run_IGNORE` function serves as an entry point for `"example2-plain.ipf"`. By using the `_IGNORE` suffix, the function itself will be ignored as a test case. This is also explained in the section about *Test Cases*. It is important to note that calling *RunTest()* would otherwise lead to a recursion error.

There are multiple calls to *RunTest()* in `run_IGNORE` to demonstrate the use of optional arguments. Calling the function without any optional argument will lead to a search for all available test cases in the procedure file. You can also execute specific test cases by supplying them with the `testCase` parameter.

The optional parameter `name` is especially useful for bundling more than one procedure file into a single test run.

The test suite itself lives in a module and all test cases are static to that module. This is the recommended environment for a test suite. When using the static keyword, you also have to define a module with `#pragma ModuleName=Example2`

Listing 7: example2-plain.ipf

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

#include "unit-testing"

Function run_IGNORE()

        // executes all test cases of this file
        RunTest("example2-plain.ipf")
        // execute only one test case at a time
        RunTest("example2-plain.ipf", testCase="VerifyStringComparison")
        // explicitly specify both tests
        RunTest("example2-plain.ipf", testCase="VerifyStringComparison;
↪VerfifyEmptyString")
        // specify with regular expression
        RunTest("example2-plain.ipf", testCase="Verify.*", enableRegExp = 1)
        // Give the test a descriptive name
        RunTest("example2-plain.ipf", name="My first test")
```

(continues on next page)

(continued from previous page)

```
End

static Function VerifyStringComparison()

        string strLow      = "123abc"
        string strUP       = "123ABC"

        // by default string comparison is done case insensitive
        CHECK_EQUAL_STR(strLow, strUP)
        // It can be specificylly enabled or disabled.
        CHECK_EQUAL_STR(strLow, strUP, case_sensitive = 0)
        // Now we use WARN because the two strings are not equal.
        WARN_EQUAL_STR(strLow, strUP, case_sensitive = 1)
        // other comparisons are also possible
        CHECK_EQUAL_VAR(strlen(strLow), 6)
End

static Function VerfifyEmptyString()

        string nullString
        string emptyString = ""
        string filledString = "filled"

        // an uninitialized string is not equal to an empty string.
        CHECK_NEQ_STR(emptyString, nullString)
        // same as for a filled string
        CHECK_NEQ_STR(filledString, nullString)
        // there is an explicit function for empty strings
        CHECK_EMPTY_STR(emptyString)
        // and also for null strings.
        CHECK_NULL_STR(nullString)
End
```

Listing 8: command

```
run_IGNORE()
```

Note: The definition for the *Assertions* in this test suite:

- *CHECK_EQUAL_STR()*
- *CHECK_NEQ_STR()*
- *CHECK_EMPTY_STR()*
- *CHECK_NULL_STR()*

1.5.3 Example3

This test suite emphasizes the difference between the *WARN()*, *CHECK()*, and *REQUIRE()* assertion variants.

The *WARN_** variant does not increment the error count if the executed assertion fails. *CHECK_** variants increase the error count. *REQUIRE_** variants also increment the error count but will stop the execution of the test case immediately if the assertion fails.

Even if a test has failed, the test end hook is still executed. See *Example5* for more details on hooks.

Listing 9: example3-plain.ipf

```
#pragma rtGlobals=3
#pragma TextEncoding="UTF-8"
#pragma ModuleName=Example3

#include "unit-testing"

// 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_* will stop execution of the test case immediately
Function RequireTest()

        REQUIRE_EQUAL_VAR(1.0,0.0)
        print "If I'm reached math is wrong !"
End
```

Listing 10: command

```
print RunTest("example3-plain.ipf")
```

The error count this test suite returns is 2

Note: See also the section on *Assertion Types*.

- *CHECK()*
- *WARN()*
- *REQUIRE()*

1.5.4 Example4

This test suite shows the use of test assertions for waves.

The type of a wave can be checked with *CHECK_EQUAL_WAVES()* and binary flags for the *MinorType* and *MajorType*. All flags are defined in *Test Wave Flags* and can be concatenated as shown in line 45. If the comparison is done against such a concatenation, it will fail if a single flag is not true. This is also shown in line 47 where the free wave does not exist but as proven before, it is definitely numeric.

It is noteworthy that each test case is executed in a fresh and empty datafolder. There is no need to use `KillWaves` or `Make/O` here.

Listing 11: example4-wavechecking.ipf

```
#pragma rtGlobals=3
#pragma TextEncoding="UTF-8"
#pragma ModuleName=Example4

#include "unit-testing"


static Function CheckMakeDouble()

        CHECK_EMPTY_FOLDER()

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

        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

static Function CheckWaveTypes()

        WAVE/Z wv
        CHECK_WAVE(wv, NULL_WAVE)

        Make/FREE/U/I wv0
        CHECK_WAVE(wv0, FREE_WAVE | NUMERIC_WAVE, minorType = UNSIGNED_WAVE | INT32_
↪WAVE)

        Make/FREE/T wv1
        CHECK_WAVE(wv1, FREE_WAVE | TEXT_WAVE)

        Make/O/U/I root:wv2/WAVE=wv2
        CHECK_WAVE(wv2, NORMAL_WAVE | NUMERIC_WAVE, minorType = UNSIGNED_WAVE | INT32_
↪WAVE)
        //The following check for a free wave is intended to fail
        WARN_WAVE(wv2, FREE_WAVE | NUMERIC_WAVE, minorType = UNSIGNED_WAVE | INT32_
↪WAVE)
End
```

Listing 12: command

```
print RunTest("example4-wavechecking.ipf")
```

Helper functions to check wave types and compare with reference waves are also provided in *Assertions*.

Note: The definition for the *Assertions* in this test suite:

- *CHECK_EMPTY_FOLDER()*
- *CHECK_WAVE()*
- *CHECK_EQUAL_VAR()*
- *CHECK_EMPTY_STR()*
- *CHECK_EQUAL_WAVES()*

1.5.5 Example5

The two test suites show how to use test hook overrides.

Here is shown how user 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 the test cases. This is done by declaring the `TEST_CASE_BEGIN_OVERRIDE` or `TEST_CASE_END_OVERRIDE` function `'static'`. Functions with this sepcific naming and the `_OVERRIDE` suffix are automatically found and registered as hooks.

Be aware that a `'static'` defined hook 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 to the static override function:

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

The second procedure file *example5-extensionhooks-otherSuite.ipf* is in `ProcGlobal` context so the test hook extensions are also global.

Listing 13: example5-extensionhooks.ipf

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

#include "unit-testing"

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
```

(continues on next page)

(continued from previous page)

```
End

static Function CheckSquareRoot()

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

Listing 14: example5-extensionhooks-otherSuite.ipf

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

#include "unit-testing"

Function TEST_BEGIN_OVERRIDE(name)
        string name

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

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
```

Listing 15: command

```
RunTest("example5-extensionhooks.ipf;example5-extensionhooks-otherSuite.ipf")
```

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

Note: Also take a look at the *Test Hooks* section.

The definition for the *Assertions* in this test suite:

- *CHECK_EQUAL_VAR()*
- *CHECK_CLOSE_VAR()*

1.5.6 Example6

This test suite shows the automatic execution of test runs from the command line. On Windows, call the "autorun-test-xxx.bat" from the helper folder.

The autorun batch script executes test runs for all pxp experiment files in the current folder. After the run, a log file is created in the folder. The log file includes the history of the Igor Pro Experiment. See also the section on *Automate Test Runs*.

Listing 16: example6-automatic-invocation.ipf

```
#pragma rtGlobals=3
#pragma TextEncoding="UTF-8"
#pragma ModuleName=Example6

#include "unit-testing"

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
```

Listing 17: example6-runner.ipf

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

#include "unit-testing"

Function run()

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

Note: The definition for the *assertion* in this test suite:

- *CHECK_EQUAL_VAR()*

1.5.7 Example7

This test suite is showing how unhandled aborts in the test cases are displayed.

The Test environment catches such conditions and treats them accordingly. This works with both `Abort` and `AbortOnValue`.

Listing 18: example7-uncaught-aborts.ipf

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

#include "unit-testing"

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
```

Listing 19: command

```
RunTest("example7-uncaught-aborts.ipf")
```

Note: Relevant definitions for the *Assertions* in this test suite:

- *PASS()*

1.5.8 Example8

This test suite shows the behaviour of the unit testing environment if 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 (RTE) is of course treated as *FAIL()*.

In this example, the highlighted lines both generate such a RTE due to a missing data folder reference.

There might be situations where the user wants to catch a runtime error (RTE) himself. In line 12 `TestWaveOpSelfCatch` shows how to catch the RTE before the test environment handles it. The test environ-

ment is controlled manually by *PASS()* and *FAIL()*. *PASS()* increases the assertion counter and *FAIL()* treats this assertion as fail when a RTE was caught.

Listing 20: example8-uncaught-runtime-errors

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

#include "unit-testing"

Function TestWaveOp()

                WAVE/Z/SDFR=$"I dont exist" wv;
End

Function TestWaveOpSelfCatch()

        try
                WAVE/Z/SDFR=$"I dont exist" wv; AbortOnRTE
                PASS()
        catch
                // Do not forget to clear the RTE
                variable err = getRTError(1)
                FAIL()
        endtry
End
```

Listing 21: command

```
RunTest("example8-uncaught-runtime-errors.ipf")
```

Note: Relevant definitions for the doc:*assertions* in this test suite:

- *PASS()*
- *FAIL()*

1.5.9 Example9

This examples shows how the whole framework can be run in an independent module.

Please note that when calling the test suite, the procedure window name does *not* need to include any independent module specification.

Listing 22: example9-IM.ipf

```
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3
#pragma IndependentModule=Example9

#include "unit-testing"

Function TestMe()
```

(continues on next page)

(continued from previous page)

```
        CHECK_EQUAL_VAR(1, 1)
End
```

Listing 23: command

```
Example9#RunTest("example9-IM.ipf")
```

Note: Definition for the *assertion* in this test suite:

- *CHECK_EQUAL_VAR()*

1.5.10 Example10

This example tests the functionality of a peak find library found on github. It demonstrates that by defining a unit test, we can rely on the functionality of an external library. Even though we can not see the code itself from this unit, we can test it and see if it fits our needs. Keep in mind that a program is only as good as the unit test the define it.

Listing 24: example10-peakfind.ipf

```
#pragma TextEncoding = "UTF-8"
#pragma rtGlobals=3                // Use modern global access method and strict wave␣
↪access.

#include "unit-testing"

// https://github.com/ukos-git/igor-common-utilities.git
#include "common-utilities"

Function testSinglePeakFit()

        // define a peak
        variable peak_position = 570
        variable peak_fwhm = 50

        // create the peak
        Make/O root:spectrum/WAVE=peak
        SetScale x, 0, 1000, "nm", peak
        peak = Gauss(x, peak_position, peak_fwhm) + gnoise(1e-3)

        // do the fit
        wave/Z/WAVE peakParam = Utilities#FitGauss(peak)

        // check that our input wave was good
        REQUIRE_WAVE(peak, NUMERIC_WAVE, minorType = FLOAT_WAVE)
        // check that the returned function is a valid wave
        REQUIRE_WAVE(peakParam, FREE_WAVE | WAVE_WAVE)
        // require at least one peak
        REQUIRE_EQUAL_VAR(1, DimSize(peakParam, 0) > 0)
        // warn if more than one peak was found
        WARN_EQUAL_VAR(1.0, DimSize(peakParam, 0))

        // convert to human readable result
        wave/Z peakInfo = Utilities#peakParamToResult(peakParam)
```

(continues on next page)

(continued from previous page)

```
        // again, check that the function returned a valid wave
        CHECK_WAVE(peakInfo, FREE_WAVE | NUMERIC_WAVE)
        // check the found peak against the peak definition
        REQUIRE_CLOSE_VAR(peakInfo[0][%position], peak_position, tol=peakInfo[0][
↪%position_err])
        REQUIRE_CLOSE_VAR(peakInfo[0][%fwhm], peak_fwhm, tol=peakInfo[0][%fwhm_err])
End
```

Listing 25: command

```
RunTest("example10-peakfind.ipf")
```

Note: Definition for the *Assertions* in this test suite:

- *CHECK_WAVE()*
- *CHECK_EQUAL_VAR()*
- *CHECK_CLOSE_VAR()*

1.6 Code Documentation

The code documentation is done using doxygen and converted for sphinx using breathe.

1.6.1 Assertions

group **Assertions**

Functions

variable **CHECK**(variable var)

Tests if var is true (1).

Parameters

- `var`: variable to test

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).

Variant for complex numbers.

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)

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)

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

- `str`: string to test

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)

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 *EqualWaveFlags*

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

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 *EqualWaveFlags*
- `tol`: (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 = 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)

variable **CHECK_NEQ_VAR**(variable var1, variable var2)
Tests two variables for inequality.

Parameters

- `var1`: first variable
- `var2`: second variable

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

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

variable **CHECK_NULL_STR**(string **str*)
Tests if str is null.

An empty string is never null.

Parameters

- `str`: string to test

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

variable **CHECK_SMALL_CMPLX**(variable/c var, variable tol = defaultValue)
Tests if a variable is small using the inequality $|var| < |tol|$.

Variant for complex numbers

Parameters

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

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

variable **CHECK_WAVE**(WaveOrNull *wv*, variable majorType, variable minorType = defaultValue)
Tests a wave for existence and its type.

See testWaveFlags

Parameters

- `wv`: wave reference
- `majorType`: major wave type
- `minorType`: (optional) minor wave type

variable **FAIL**()
Force the test case to fail.

variable **PASS**()
Increase the assertion counter only.

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**()
Tests if the current data folder is empty.

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

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)

1.6.2 Helper Functions

The following Helper Functions are available:

group **Helpers**

Functions

variable **DisableDebugOutput**()
Turns debug output off.

variable **EnableDebugOutput**()
Turns debug output on.

1.6.3 Logical Flags

The following flags are binary set. One or more of them can apply at the same time.

Equal Wave Flags

These flags are used in *CHECK_EQUAL_WAVES()*

group **EqualWaveFlags**

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

Test Wave Flags

The following flags are used in *CHECK_WAVE()*. Note that there is a minor and a major wave type.

General

group **TestWaveFlagsGeneral**

Variables

const variable NULL_WAVE = 0x00

MajorType

group **TestWaveFlagsMajor**

Variables

const variable DATAFOLDER_WAVE = 0x04

const variable FREE_WAVE = 0x20

const variable NORMAL_WAVE = 0x10

const variable NUMERIC_WAVE = 0x01

const variable TEXT_WAVE = 0x02

const variable WAVE_WAVE = 0x08

MinorType

group **TestWaveFlagsMinor**

Variables

const variable COMPLEX_WAVE = 0x01

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 UNSIGNED_WAVE = 0x40

INDEX

C

CHECK (C++ function), 25
CHECK_CLOSE_CMPLX (C++ function), 25
CHECK_CLOSE_VAR (C++ function), 26
CHECK_EMPTY_FOLDER (C++ function), 26
CHECK_EMPTY_STR (C++ function), 26
CHECK_EQUAL_STR (C++ function), 26
CHECK_EQUAL_TEXTWAVES (C++ function), 26
CHECK_EQUAL_VAR (C++ function), 27
CHECK_EQUAL_WAVES (C++ function), 27
CHECK_NEQ_STR (C++ function), 27
CHECK_NEQ_VAR (C++ function), 27
CHECK_NON_EMPTY_STR (C++ function), 27
CHECK_NON_NULL_STR (C++ function), 27
CHECK_NULL_STR (C++ function), 28
CHECK_PROPER_STR (C++ function), 28
CHECK_SMALL_CMPLX (C++ function), 28
CHECK_SMALL_VAR (C++ function), 28
CHECK_WAVE (C++ function), 28

D

DisableDebugOutput (C++ function), 30

E

EnableDebugOutput (C++ function), 30

F

FAIL (C++ function), 28

P

PASS (C++ function), 28

R

REQUIRE (C++ function), 28
REQUIRE_CLOSE_CMPLX (C++ function), 28
REQUIRE_CLOSE_VAR (C++ function), 28
REQUIRE_EMPTY_FOLDER (C++ function), 28
REQUIRE_EMPTY_STR (C++ function), 29
REQUIRE_EQUAL_STR (C++ function), 29
REQUIRE_EQUAL_TEXTWAVES (C++ function), 29
REQUIRE_EQUAL_VAR (C++ function), 29
REQUIRE_EQUAL_WAVES (C++ function), 29
REQUIRE_NEQ_STR (C++ function), 29
REQUIRE_NEQ_VAR (C++ function), 29
REQUIRE_NON_EMPTY_STR (C++ function), 29
REQUIRE_NON_NULL_STR (C++ function), 29
REQUIRE_NULL_STR (C++ function), 29
REQUIRE_PROPER_STR (C++ function), 29
REQUIRE_SMALL_CMPLX (C++ function), 29
REQUIRE_SMALL_VAR (C++ function), 29
REQUIRE_WAVE (C++ function), 29
RunTest (C++ function), 9

T

TEST_BEGIN_OVERRIDE (C++ function), 11
TEST_CASE_BEGIN_OVERRIDE (C++ function), 12
TEST_CASE_END_OVERRIDE (C++ function), 12
TEST_END_OVERRIDE (C++ function), 11
TEST_SUITE_BEGIN_OVERRIDE (C++ function), 12
TEST_SUITE_END_OVERRIDE (C++ function), 12

W

WARN (C++ function), 29
WARN_CLOSE_CMPLX (C++ function), 29
WARN_CLOSE_VAR (C++ function), 29
WARN_EMPTY_FOLDER (C++ function), 29
WARN_EMPTY_STR (C++ function), 29
WARN_EQUAL_STR (C++ function), 29
WARN_EQUAL_TEXTWAVES (C++ function), 29
WARN_EQUAL_VAR (C++ function), 29
WARN_EQUAL_WAVES (C++ function), 29
WARN_NEQ_STR (C++ function), 29
WARN_NEQ_VAR (C++ function), 29
WARN_NON_EMPTY_STR (C++ function), 29
WARN_NON_NULL_STR (C++ function), 29
WARN_NULL_STR (C++ function), 29
WARN_PROPER_STR (C++ function), 29
WARN_SMALL_CMPLX (C++ function), 29
WARN_SMALL_VAR (C++ function), 29
WARN_WAVE (C++ function), 29