NITS Integrated Test System

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Who | State | Version | Date | Comment |
| rmack | In progress | 0.1 | 12/31/2007 |  |
| rmack | In Review | 0.2 | 1/15/2008 | Improved examples and notes. |
| rmack | In Review | 1.1 | 11/9/2012 | New syntax/linkage. |
| girishjo | In Review | 1.2 | 9/30/2013 | Updates to New Syntax/linkage |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

[New & Improved Test System 1](#_Toc188153480)

[1 Introduction 2](#_Toc188153481)

[2 Project Setup 3](#_Toc188153482)

[3 Command Line Interface 3](#_Toc188153483)

[3.1 Options 3](#_Toc188153484)

[3.2 Tests 4](#_Toc188153485)

[3.3 Results 4](#_Toc188153486)

[4 Implementation 5](#_Toc188153487)

[5 Features 6](#_Toc188153488)

[5.1 Tracing 6](#_Toc188153489)

[5.2 Assertions 6](#_Toc188153490)

[5.3 Call Sites 6](#_Toc188153491)

[5.4 Setup and Cleanup 6](#_Toc188153492)

[5.5 Parameters 7](#_Toc188153493)

[5.6 Shared Memory 7](#_Toc188153494)

[5.7 Fault Simulation 8](#_Toc188153495)

[5.8 Test Setting Groups 8](#_Toc188153496)

[5.9 Test Variations 9](#_Toc188153497)

[6 Features Not Implemented 10](#_Toc188153498)

[6.1 XML Output 10](#_Toc188153499)

[6.2 Test Isolation 10](#_Toc188153500)

# Introduction

This document defines a C unit test framework and instructions for its use. This framework is a replacement for existing unit test infrastructure that makes debugging, maintenance, and code coverage significantly easier. The unit test framework encompasses several related components:

* **nitsdll.dll**: Contains the framework implementation. This binary contains shared memory spaces appropriate for test builds only. Product binaries must not link to nitsdll.dll on official builds.
* **nitsinj.dll**: Injects the test framework and any other mocked function tables into relevant processes during unit test runs. No one links to this binary directly.
* **nitsstub.lib**: Contains no-op implementations of the NITS API for product binaries using HOOK\_BUILD or DEBUG\_BUILD linkage.
* **nits.exe**: Contains a command-line interface for running the framework on a set of tests. Spawns child processes if necessary, runs test cases, and reports results.
* Unit test DLLs: Contains product unit test suites. All test cases are declared within the .cpp files of these binaries. Links to nitsdll.dll using TEST\_BUILD linkage.
* Product DLLs: Uses macros and either runtime instrumentation or conditional blocks of code to run under unit test passes. Should use HOOK\_BUILD or DEBUG\_BUILD linkage.

All of the NITS components were moved to admin/wmi/winomi/nits during the WinBlue release, and are in the process of being made portable. Most functionality is already working on Linux as part of the WinOMI project, using the same codebase.

# Linkage

There are five different possible ways to link to the NITS package, each with its own consequences:

* **<default>**: All test macros are no-ops and NitsCallSite is minimized. It is not possible to hook anything at runtime. This option is used in projects/binaries that do not understand NITS.
* **HOOK\_BUILD**: Test macros call NitsFT stubs in nitsstub.lib. NitsCallSite is functional but does not show file/line information. Use the –target option to hook selected binaries at runtime. This option is for production DLLs.
* **DEBUG\_BUILD**: Same as HOOK\_BUILD, but includes \_\_FILE\_\_/\_\_LINE\_\_/\_\_FUNCTION\_\_ information. This option is intended for production DLLs running under a debug/checked build.
* **TEST\_BUILD**: Test macros call the NITS implementation through DLL linkage to nitsdll.dll. This is for test DLLs.
* **LIB\_BUILD**: Test macros call the NITS implementation through static linkage to nitsbase.lib. This option is not recommended.

# Project Setup

This section describes the basic source tree configuration steps required to access the NITS API. Once these steps are complete, product and test binaries are ready to start using the framework. The project will need access to the following files from the framework: **nits.h**, **nitsdll.dll**, **nitsdll.lib**, **pal.lib,** **nitsstub.lib**, **nitsstub.obj** and **nits.exe**. The recommended, supported project configuration is as follows:

## Product Binaries

TARGETTYPE=DYNLINK

INCLUDES= $(INCLUDES); \

$(ADMIN\_INC\_PATH); \

TARGETLIBS= $(TARGETLIBS) \

$(ADMIN\_LIB\_PATH)\pal.lib \

$(ADMIN\_LIB\_PATH)\nitsstub.lib \

$(SDK\_LIB\_PATH)\kernel32.lib \

$(SDK\_LIB\_PATH)\advapi32.lib \

C\_DEFINES= $(C\_DEFINES) \

/DHOOK\_BUILD \

DLLDEF\_OBJECTS= \

$(ADMIN\_LIB\_PATH)\nitsstub.obj \

* The product code should be a DLL. Use of EXE modules should be only as a thin wrapper over DLLs.
* Define HOOK\_BUILD in product binaries as part of the sources files. (For checked builds, DEBUG\_BUILD provides more file/line information, but it also bloats the binary.)
* Include <nits/base/nits.h> in product and test source files.
* Link to nitsstub.lib and pal.lib from product binaries.
* Have nitsstub.obj in DLLDEF\_OBJECTS for the product binaries.(This is required so that the stubbed API table for the unittest framework can be dll exported and can be patched by the injector module at runtime to enable product to call into NITS APIs)
* For HOOK\_BUILD to work on Windows, advapi32.lib, kernel32.lib or equivalent mincore libraries must be linked as well. (This is required since nitsstub uses registry related APIs like RegOpenKeyEx, RegQueryValueEx, RegCloseKey and APIs like LoadLibraryEx to bootstrap the injector module.)

## Unit Test Binaries

TARGETTYPE=DYNLINK

INCLUDES= $(INCLUDES); \

$(ADMIN\_INC\_PATH); \

TARGETLIBS= $(TARGETLIBS) \

$(ADMIN\_LIB\_PATH)\nitsdll.lib \

$(SDK\_LIB\_PATH)\kernel32.lib \

C\_DEFINES= $(C\_DEFINES) \

/DTEST\_BUILD \

* Define TEST\_BUILD in unit test binaries as part of the sources files.
* Link to nitsdll.lib from unit test binaries.
* Link to kernel32.lib or its equivalent mincore library
* Include nitsdll.dll, nitsinj.dll, and nits.exe when distributing unit test binaries to test environments.

## Pitfalls/Notes

* Do not link test DLLs against product static libs. This mixes HOOK\_BUILD code with TEST\_BUILD code. If you really need to do this, contact ([rmack or girishjo)@microsoft.com](mailto:rmack@microsoft.com) for more specific guidelines.
* There are significant drawbacks to using HOOK\_BUILD against EXE modules (i.e. the trap table technique for calling internal product APIs does not work).

## Sample Project

For canonical up-to-date working examples, see the following parts of the monitoring platform project under admin/wmi/winomi/mp:

* Product linkage settings are in mp/makefile.cmn and mp/mpunits/sources.
* Unit test linkage settings are in mp/tests/makefile.cmn and mp/tests/common/sources.

There are more examples of various linkage types under admin\wmi\winomi\nits\linkageSample. There are many test examples under admin\wmi\winomi\nits\sample

# Deployment

Several steps are necessary in order to deploy and use NITS, which is not part of the OS image. Sample scripts are available under admin/wmi/winomi/mp/setup. General considerations:

* Deploy NITS by copying nits.exe, nitsdll.dll, and nitsinj.dll to %SystemRoot%\System32.
* Before deploying NITS, run nits –uninstall.
* After deploying NITS, but before running any tests, run nits –install.
* Don’t run NITS from %\_NTTREE% directly, since that won’t work if %\_NTTREE% points to a network share. Instead, add relevant folders under %\_NTTREE% to your PATH.
* Running code coverage analysis does not require any special measures; ad-hoc dev machine coverage data can be gathered by copying binaries from %\_NTTREE%\_\coverage to %\_NTTREE% before running the same deployment and unit test steps.

# How to Call Product APIs

NITS provides facilities to allow private APIs to be callable at will without individually DLL-exporting all of them. This makes it convenient to link test DLLs directly against product DLLs, which in turn allows simple and reliable code coverage measurements. The alternative is linking the test code to product static libs, which leads to product code existing in many binaries at once. While this is not technically broken, it is impossible to retrieve accurate coverage data in complex cases, and this has some tendency to encourage duplication of product code in multiple product binaries.

## Public APIs

Public interfaces must be DLL-exported somehow. This may be done individually for each API, or function tables may be used (as in mi.h) to export an entire collection of APIs from one binary using a single DLL export. The latter technique leads to smaller binary size but requires more care to avoid introducing breaking changes to the contents of the table.

## Private APIs

Non-public interfaces do not need to be DLL exported in the same way. The recommended approach is to place all of the externally callable private APIs from a given product DLL into a single NitsTrapTable(…), and then import that table into all the binaries that call any of the APIs. Macros can be used to hide this behavior from the calling binaries. For a complete and working example, see admin/wmi/winomi/mp/mpunits, in particular mpunits.traps.h and mpunits.traps.c.

# How to Mock Product APIs

NITS supports mocking private APIs that are already exposed through a function table using the NitsSetTrap API. The only additional change needed in the product is that all mockable calls in the product must invoke through the trap table, rather than directly calling the function. When NitsSetTrap is called, the function pointer in the table is replaced and all calls through the pointer are redirected to the mock. For a simplified proof of concept, see admin/wmi/winomi/mp/tests/common/testcommon.cpp, specifically the TestSampleTrap test.

# Command Line Interface

The usage pattern for NITS is the following: **nits [option|test]\***. The rest of this section describes the available options and test types, along with the possible results for each test.

## Options

* **-bpassert**: Issues a breakpoint any time an assertion fails when it should not. This is useful for debugging when it takes many steps to walk through the code to the point where the assertion fails.
* **-bpfault:number**: Issues a breakpoint on a specific fault injection iteration number. This option may be used multiple times.
* **-debug:”command”**: Automatically issues a debug attach command against all target processes in the user’s session, including service hosts and other processes not under the control of the unit test process.
* **-file:name**: Appends the contents of *name* to the command line options.
* **-filter:binary[;binary…]**: Prevents the injector from targeting processes containing none of the binaries in the filter list.
* **-install**: Immediately installs the injector to run in all relevant processes created after this point (including upon future boots).
* **-mode:Skip**: Explores the test tree and skips all body fixtures.
* **-mode:Enable**: (default) Runs test cases without fault injection.
* **-mode:IterativeFault**: Runs test cases with Nth site single faults. This option is exhaustive for deterministic tests, but very slow.
* **-mode:StackFault**: Optimized single fault injection; ignores duplicates based on call stack frames.
* **-pause**: Waits for user input after running. Useful for holding debugger settings in effect when doing manual testing.
* **-reset**: Ignores a test run in progress and forcibly resets the shared memory state. Use this to recover from shared memory corruption.
* **-target:binary[;binary…]**: Enables the injector on specific binaries in any process.
* **-trace:FailedTests**: Shows failed tests only. Least verbose result output.
* **-trace:WarnedTests**: Shows failed and warned tests only.
* **-trace:AllTests**: Shows all test cases.
* **-trace:Asserts**: (default) Shows failed asserts individually.
* **-trace:Warnings**: Shows failed asserts and warnings individually.
* **-trace:Iterations**: Shows fault injection iteration information and asserts/warnings.
* **-trace:Verbose**: Shows all of the above plus NitsTrace calls.
* **-uninstall**: Removes nitsinj.dll injection from all processes. Use this before updating NITS binaries.
* **-wtt:file**: Logs WTT results to a WTT log file (requires WTT to be installed).
* **-match:testFixtureNameSubstring**: Runs tests only if they contain fixture names matching the pattern (semicolon or comma-separated).
* **+name:value**: Sets test parameter 'name' to 'value'. Test module can retrieve the value of the parameter by using the API NitsTestGetParam by passing the parameter name.

## Tests

Tests may be run using any of the following syntax forms:

* **module.dll**: Specifying a file name runs all tests discovered during the loading of that library. NITS now accepts relative and absolute paths.
* **module.dll:test**: Specify the module name and a test name, separated by a bang, to run a particular test. This runs all test variations.
* **module.dll:+test**: Runs all tests in the module starting with the specified test.
* **module.dll:test/variation**: Specify the module name, the test name, and a set of variation choices to run one specific test variation. Each valid variation name is constructed from the set of choices made according to the hierarchy declared in the test code.
* **module.dll:\*testnameSubstring**: Runs all tests whose names contain the pattern.

The simplest way to discover a list of valid test names and variations for a particular module is to run the entire module with the **–mode:Skip** option.

NOTE: To run the NITS sample tests, use **nits –target:NitsSample.dll;NitsSampleProduct.dll NitsSample.dll**

## Results

Each test variation reports a result to the test framework. These results are printed and then summarized. The possible test results are as follows:

* **Faulted**: The test passed, and was then run successfully under automatic fault simulation. There were no unexpected successes or failures during the fault simulation loop.
* **Passed**: The test passed, and automatic fault simulation was not run.
* **Skipped**: The test was disabled by the command line or the test setup code, and the test body was not run. However, no errors were encountered during setup or cleanup.
* **Omitted**: The test declared itself to be irrelevant. These can be safely ignored.
* **Killed**: When using the (old) **isolate** option, a test case was terminated after running for too long without reporting a result.
* **Failed**: Assertions failed during the test body, or there were unexpected successes or failures during automatic fault simulation.
* **Error**: Assertions failed during test setup or cleanup, or no assertions were attempted during the test body.

# Implementation

This section describes how to create basic unit tests under the NITS framework. This section shows only the most basic examples, using as few features as possible. See the following section for additional examples for advanced features. To run a test DLL containing a simple, “hello world” unit test, see the following example:

#include <nits/base/nits.h>

NitsTest(Test1)

NitsTrace(PAL\_T("Test Body!"));

NitsAssert(1 == 1, PAL\_T("assert test!"));

NitsCompare(1, 1, PAL\_T("compare test!"));

NitsCompareString(PAL\_T("A"), PAL\_T("A"), PAL\_T("string test!"));

NitsCompareSubstring(PAL\_T("ABC"), PAL\_T("B"), PAL\_T("substring test!"));

NitsEndTest

The code above contains a simple test body with some basic assertions and traces. Note that the assertion descriptions must be Unicode, but some other arguments may be either ANSI or Unicode. Inserting this code into hello.dll and compiling it produces a binary which can be run using the following command:

nits hello.dll

[Passed] hello.dll!Test1

Summary:

Faulted: 0

Passed: 1

Skipped: 0

Killed: 0

Failed: 0

Error: 0

Successes: 1

Failures: 0

Total: 1

This listing shows that the test named **Test1** in **hello.dll** was run successfully. The test passed because the body ran one or more test assertions, all of which succeeded.

# Features (Under Construction)

This section contains a partial list of available NITS features, roughly in order of increasing complexity. Each feature can be used independently of the others unless otherwise specified.

## Tracing

The API NitsTrace is available for tracing messages using the current source location as the call site. These messages will appear in stderr only if tracing is enabled.

## Assertions

The APIs NitsAssert, NitsCompare, NitsCompareString, and NitsCompareSubstring validate results obtained during the test. These functions return true if successful, otherwise they are printed to stderr if assertions are enabled.

## Call Sites

NITS defines a **NitsCallSite** class which is available on all linkages, though on some linkages it becomes trivialized. This class contains a source file, line, function name, and call site ID to be used for fault injection and source location identification. The APIs above use the current source line (NitsHere() macro) as the NitsCallSite. However, this is not appropriate in common helper functions where it is actually the caller’s location that is relevant. The following example shows how to carry a CallSite through to a helper function:

void TestHelper(NitsCallSite cs)

{

NitsTraceEx(L"Helper function!", cs, NitsAutomatic); //Prints location of NitsHere(), below.

}

NitsTest(Test1)

TestHelper(NitsHere());

NitsEndTest

The macros NitsTraceEx, NitsAssertEx, NitsCompareEx, NitsCompareStringEx, and NitsCompareSubstringEx are available for this purpose, each taking the same argument list as their counterparts, plus a call site argument. NitsCallSite objects may be created easily using either the NitsHere() or NitsNamedCallSite macros. For manual fault simulation, use NitsNamedCallSite(id), where the site ID is defined by the application. Otherwise, NitsHere() is sufficient, which is anonymous.

## Fixtures

NITS tests are made from one or more Fixtures. Fixtures are reusable units of test code. They support composability i.e. they can be composed out of other fixtures. They also allow you to share data between each other by letting you associate a type(C struct) with them and providing an initialization value for that type. There are following types of fixtures

* Setup
* Split
* Test
* Cleanup
* ModuleSetup

Following subsections contain detailed information about the above types of fixtures and also how you can share data between the fixtures using an auto-created abstraction called NitsContext().

## Setup Fixtures

### Most Basic type of Setup Fixture

Syntax of most Basic Type of Setup Fixture involves writing test code between NitsSetup(<fixtureName>) and NitsEndSetup macros. Here is an example of most Basic type of Setup fixture

NitsSetup(SimpleSetup)

NitsTrace(PAL\_T("SimpleSetup being run"));

NitsEndSetup

* SimpleSetup is the name of the fixture.
* NitsSetup and NitsEndSetup form the start and end of the setup fixture.
* The part in between represents the body of the Setup fixture and it will be run during the test run.
* Note: NitsEndSetup and similar End macros are part of every fixture and they help NITS provide you functionality of function continuation style usage i.e. stacking up all the fixtures from which your test is composed to be on the same stack which lets you use local variables throughout the lifetime of the test

### Associating Data Type with a Setup Fixture

To associate data type with a Setup fixture, you need to define a C struct as the data type, and write test code between NitsSetup0(<fixtureName>, <name of the data type>) and NitsEndSetup macros. Here is an example which shows how to associate type with a Setup Fixture.

struct MyStruct

{

int x;

};

NitsSetup0(Fixture0, MyStruct)

NitsContext()->\_MyStruct->x = 0;

NitsEndSetup

* Fixture0 is the name of the Fixture
* NitsSetup0 and NitsEndSetup form the start and end of the setup fixture. The number 0 represents that it is composed out of 0 other fixtures. Further subsections will explain how NitsSetup<N> is used to define a Setup fixture composed out of N other fixtures.
* MyStruct defines a C Struct which is then associated with the Fixture0 as its data type.
* Once you associate a data type with a Setup fixture, NITS auto-creates an object of that type(MyStruct) and makes it available for use by the user inside the NitsContext() abstraction.
* The NitsContext() will have auto-created variable with name **\_<Data type name>** (in this example \_MyStruct) which represents pointer to that auto-created data object which can be used by the user for data sharing between fixtures.
* As seen in the example below the (NitsContext()->\_MyStruct->x) lets you access the int x from the auto-created object of type MyStruct which is pointed to by the variable \_MyStruct inside NitsContext()

### Setup Fixture Composition

Syntax for composing a Setup fixture with 1 other child fixture involves writing the test code between NitsSetup1(<fixtureName>, <name of fixture data Type>, <name of child fixture 1>, <initialization value for child fixture 1>) and NitsEndSetup macros. Here is an example of how to compose a fixture out of one other child fixture at the same time associating data types with both fixtures which let you pass data around in either direction.

struct FooStruct1

{

int i1;

};

struct FooStruct2

{

int i2;

};

NitsSetup0(FooSetup1, FooStruct1)

NitsAssert(NitsContext()->\_FooStruct1->i1 == 10, PAL\_T("wrong value"));

NitsContext()->\_FooStruct1->i1 = 15;

NitsTrace(PAL\_T("FooSetup1 being run"));

NitsEndSetup

struct FooStruct1 sFooStruct1 = {10};

NitsSetup1(FooSetup2, FooStruct2, FooSetup1, sFooStruct1)

NitsAssert(NitsContext()->\_FooSetup1->\_FooStruct1->i1 == 15, PAL\_T("wrong value"));

NitsContext()->\_FooSetup1->\_FooStruct1->i1 = 35;

NitsContext()->\_FooStruct2->i2 = 25;

NitsTrace(PAL\_T("FooSetup2 being run"));

NitsEndSetup

* Here FooSetup1 and FooSetup2 are names of the two setup fixtures.
* FooSetup1 uses the syntax NitsSetup0/NitsEndSetup since it is not composed out of any other fixture.
* FooSetup2 is composed out of child fixture FooSetup1 and hence uses the syntax NitsSetup1/NitsEndSetup
* The C struct FooStruct1 is the data type associated with FooSetup1
* The C struct FooStruct2 is the data type associated with FooSetup2
* When composing from other child fixture in NitsSetup1, the syntax used is (FooSetup2, FooStruct2, FooSetup1, sFooStruct1) which are
  + Name of the fixture
  + Data type associated with the fixture
  + And following is the pair containing name of the child fixture and initialization value for the fixture.
* The initialization value sFooStruct1 given to fixture FooSetup1 will be automatically populated into the data object pointed by NitsContext()->\_FooStruct when the fixture FooSetup1 runs. (Note that the initialization object is used as copy-by-value and it is copied into the data object pointed to by NitsContext() of the fixture which it is initializing). That is why in the above example, the FooSetup1 can assert that the value of NitsContext()->\_FooStruct1->i1 when it runs will be 10.
* Initialization value for a fixture has to be a compile-time defined global variable of the data type associated with the fixture. Alternatively to pass all-zero initialization value, you can use the NITS-auto-created global variable which has the name (<name of fixture>Defaults) i.e. in this example if FooSetup2 fixture wanted to pass all-zero initialization value for FooSetup1 then it could have used the auto-created object FooSetup1Defaults.
* When a fixture is composed out of other child fixtures, the auto-created NitsContext() abstraction inside the fixture contains appropriate pointers to access the data from all the child fixtures it is composed of. The name of the variable inside NitsContext() which points to context of the child fixture is of the form **\_<child fixture name>**
* In this example, NitsContext() inside FooSetup2 has following variables
  + \_FooStruct2 which is a pointer to the data object of type FooStruct2 auto-created for itself since the data type associated with FooSetup2 is FooStruct2
  + \_FooSetup1 which is a pointer to the NitsContext() inside the child fixture FooSetup1
  + \_FooSetup1 then contains the pointer \_FooStruct1 which is the pointer to the data object associated with the child fixture FooSetup1 since the data type associated is FooStruct1
* Thus as seen in the above example, there is data sharing in both directions between a fixture and all child fixtures it is composed out of. This is critical when you want to parameterize fixtures and reuse the same test code written in the Setup Fixtures with different data values.
  + FooSetup2 can pass initialization value to the FooSetup1 which it can see when it is run
  + FooSetup1 can modify the values inside NitsContext() associated with it
  + FooSetup2 which will be run after FooSetup1 can then read the values modified by FooSetup1 inside NitsContext()
* NitsSetup2, NitsSetup3, NitsSetup4, NitsSetup5 let you define Setup Fixtures composed out of 2, 3, 4, 5 other child fixtures and follow similar syntax NitsSetup<2/3/4/5>(fixtureName, dataTypeOfFixtureName, <child fixture1, initialization value of child fixture1>, <child fixture2, initialization value of child fixture2>, <child fixture3, initialization value of child fixture3>, <child fixture4, initialization value of child fixture4>, <child fixture5, initialization value of child fixture5>)
* Note that when a specific setup fixture runs, all child setup fixtures that it is composed out of will be run in left to right order of their definition before the specific fixture runs. All of them will be run on the same stack one above the other and hence if you define local variables, you can use their pointers to pass around inside NitsContext(); all local variables in child fixtures will still be available when the specific fixture runs. Look at the stack below to see how the stack may look like when running the above fixtures

0:000> kn

# Child-SP RetAddr Call Site

00 0000005c`3b05e198 000007fe`d108b2f1 NitsSample!FooSetup2

…

05 0000005c`3b05e920 000007fe`d108b2f1 NitsSample!FooSetup1+0x144

…

0f 0000005c`3b05f8d0 000007f6`2483a3a3 nits!main+0x10d

* If you do not want to associate data type with a specific Setup fixture, you can just use a NITS-defined empty data type NitsEmptyStruct as the data type and NITS-defined initialization value NitsEmptyValue to initialize such empty data type fixture.

### Declaring and defining Setup Fixtures for Reuse in multiple files

NitsDeclSetup<N>/NitsDefSetup<N> macros let you declare fixtures in .h file and define in .cpp files which helps you reuse same Setup fixture in multiple files. Here is an example.

Foo.h =>

struct MyStruct

{

int x;

};

NitsDeclSetup0(Fixture0, MyStruct);

Foo.cpp =>

NitsDefSetup0(Fixture0, MyStruct)

NitsContext()->\_MyStruct->x = 0;

NitsEndSetup

* In the above example, Fixture0 is declared with data type MyStruct in Foo.h
* Fixture0 is defined in Foo.cpp.
* Note that Fixture0 can be used in other .cpp files to construct other Setup/Split/Test fixtures.

## Split Fixtures

Split Fixtures let you split the execution of tests into two or more tests and that way help you execute the same test code in multiple configurations. Syntax for Split fixture which splits a test into two child fixtures involves writing test code between NitsSplit2(<fixtureName>, <data type of fixture>, <name of child fixture 1>, <name of child fixture 2>) and NitsEndSplit macros. Here is an example of the Split fixture composed out of two child fixtures which define two child configurations in which the test will run.

struct MyContext

{

int a;

};

NitsSetup0(MySetup1, MyContext)

NitsContext()->\_MyContext->a = 4;

NitsEndSetup

NitsSetup0(MySetup2, MyContext)

NitsContext()->\_MyContext->a = 8;

NitsEndSetup

NitsSplit2(MySplitSetup, MyContext, MySetup1, MySetup2)

NitsAssert((NitsContext()->\_MyContext->a == 4) || (NitsContext()->\_MyContext->a == 8), PAL\_T(“value is wrong”));

NitsEndSplit

* Here MySetup1 and MySetup2 are two setup fixtures which have data type MyContext.
* Each of them assign a value (4/8) to the variable ‘a’ inside the NitsContext data object.
* NitsSplit2/NitsEndSplit defines syntax for defining a split fixture; MySplitSetup is a split fixture which lets you split the test into two setup fixtures MySetup1 and MySetup2.
* NITS will automatically run the test which has MySplitSetup fixture into two configurations, once with MySetup1 and next with MySetup2.
* Note that the Split fixture must have same data type as the two or more child fixtures it is composed out of. In this example that data type is the C struct MyContext.
* Note that the split fixture syntax doesn’t allow you to pass initialization value to the child fixtures. Reason is that the initialization value in child fixtures will be same as the initialization value that the split fixture gets from whoever composes out of the split fixture.
* NITS makes sure that the NitsContext()->\_MyContext inside the split fixture (MySplitSetup) will automatically point to the correct child fixture’s context (MySetup1 or MySetup2’s context) when the test is run. This allows you to assert that the value of NitsContext()->\_MyContext->a when running MySplitSetup will be either 4 or 8 since the child fixtures MySetup1/MySetup2 set it up that way by updating it to 4/8 respectively.
* NitsSplit3/NitsSplit4/NitsSplit5 let you define split fixtures which will split into 3/4/5 other child fixtures and follow the syntax NitsSplit(3/4/5)(<fixtureName>, <data type of fixture>, <name of child fixture 1>, <name of child fixture 2>, <name of child fixture 3>, <name of child fixture 4>, <name of child fixture 5>).
* Similar to Setup fixtures, you can also separate declaration and definition of split fixtures by using macros NitsDeclSplit<N>/NitsDefSplit<N>.
* Some common usage scenarios of Split Fixtures so far are as follows.
  + You have a client server test which you need to run with server started in different configurations(e.g. hosted in a windows service or in test process itself or in IIS)
  + You have some test which you need to run over multiple Auth modes or different character sets, or over different protocols like HTTP vs HTTPs and so on.

## Test Fixtures

Test is just one other kind of fixture in NITS. Implementation section already defined syntax NitsTest/NitsEndTest for the most basic test in NITS. This section will describe syntax for more advanced tests which are composed out of 1 or more other Setup/Split fixtures. The syntax for defining a test fixture which is composed of one other child fixtures involves writing test code between NitsTest1(<testFixtureName>, <child fixture 1>, <initializer for child fixture1>) and NitsEndTest macros. Here is an example:-

struct FooStruct1

{

int i1;

};

NitsSetup0(FooSetup1, FooStruct1)

NitsAssert(NitsContext()->\_FooStruct1->i1 == 20, PAL\_T("wrong value"));

NitsContext()->\_FooStruct1->i1 = 35;

NitsEndSetup

struct FooStruct1 sFooStruct1 = {20};

NitsTest1(FooTest, FooSetup1, sFooStruct1)

NitsAssert(NitsContext()->\_FooSetup1->\_FooStruct1->i1 == 35, PAL\_T("wrong value"));

NitsEndTest

* In the above example, FooSetup1 is a setup fixture with data type FooStruct1
* NitsTest1/NitsEndTest describe the syntax for defining a test composed out of one child fixture
* FooTest is the name of the test and it is composed of FooSetup1 and passes sFooStruct1 as initialization value to FooSetup1
* When the test is run, FooSetup1 runs first and that is followed by the FooTest on the same stack as mentioned earlier.
* NitsContext() inside the test body also follows a similar format as the Setup fixtures i.e. it has auto-created variables of the format **\_<Child fixture name>** which point to NitsContext() data objects inside child fixtures**.**
* Note that there is no data type associated with a Test fixture; its NitsContext() just lets you access the NitsContext() data objects inside child fixtures.
* In above example, when FooSetup1 runs, the value of variable i1 inside NitsContext()->\_FooStruct1 data object is 20 since the test passes initializer sFooStruct1 which sets it to 20.
* The FooSetup1 then updates the i1 value to 35
* So when the test body runs, the value of i1 is 35 which shows the data sharing ability between Test and its child fixtures in both directions.
* NitsTest2/NitsTest3/NitsTest4/NitsTest5 macros let you define tests composed out of 2/3/4/5 Setup/Split fixtures and follow the syntax NitsTest(2/3/4/5)(<testfixtureName>, <child fixture 1>, <initializer for child fixture 1>, <child fixture 2>, <initializer for child fixture 2>, <child fixture 3>, <initializer for child fixture 3>, <child fixture 4>, <initializer for child fixture 4>, <child fixture 5>, <initializer for child fixture 5>). The child fixtures are run in the left to right direction in which they are defined.
* NitsTestWithSetup lets you define a simple test fixture composed of a child Setup fixture defined with NitsSetup. NitsTestWithInitializableSetup lets you define a test composed of one setup fixture taking initialization data. For all the Nits\* macro definitions, please look at admin/wmi/winomi/nits/base/nits.h header file.
* Note that the interface lets you construct arbitrary DAGs of test fixtures. It is possible that you can have same Setup/Split fixture multiple times in the graph; but that is fine; NITS handles that by instantiating and running the fixture only once and that way you can be sure of not having multiple executions of the same fixture or having multiple copies of its NitsContext() object.

## Cleanup Fixtures

Cleanup fixture can be defined on any of the above type of fixtures i.e. Setup/Split/Test/ModuleSetup. The Cleanup fixture will be run at the end of the test body or at the end of test module depending on whether it is a Cleanup for Setup/Split/Test or ModuleSetup. Cleanup fixture has access to the same NitsContext() that the fixture for which it is cleaning up and hence you can use this to cleanup anything that you need to. The syntax for Cleanup fixture involves writing test code between NitsCleanup(<fixtureName for which cleanup is defined>) and NitsEndCleanup macros. Here is an example:-

NitsSetup(MySetup1)

NitsTrace(PAL\_T("MySetup1 being run"));

NitsEndSetup

NitsCleanup(MySetup1)

NitsTrace(PAL\_T("Cleanup for MySetup1 being run"));

NitsEndCleanup

# Here the Cleanup is defined on setup fixture MySetup1

* The syntax is NitsCleanup/NitsEndCleanup
* The name of the cleanup fixture is same as the fixture for which the Cleanup is defined
* In a test hierarchy, each fixture could have a Cleanup fixture defined; in that case the cleanup fixtures will be run in the reverse order in which the fixtures themselves executed e.g. Setup1=>Test1=>Cleanup(Test1)=>Cleanup(Setup1)

## ModuleSetup Fixtures

ModuleSetup fixtures let you define test code which will be run at the beginning and end of test module. This is useful in places where you want to run some piece of test code only once per test module at the beginning and end of it. Here is an example:-

NitsModuleSetup(MyModuleSetup1)

NitsTrace(PAL\_T("MyModuleSetup1 being run"));

NitsAssert(PAL\_TRUE, PAL\_T(""));

NitsEndModuleSetup

NitsCleanup(MyModuleSetup1)

NitsTrace(PAL\_T("Cleanup for MyModuleSetup1 being run"));

NitsEndCleanup

# MyModuleSetup1 defines a ModuleSetup fixture; it will be run at the beginning of the module in which it is located.

* As with all other fixtures, you can define Cleanup fixtures also on ModuleSetup fixtures. They will be run at the end of the module.
* You can have multiple ModuleSetup fixtures in a test module(dll/shared object library) and all of them will be run sequentially at the beginning of the module.
* In case of test isolation mode i.e. running tests in their own processes, the module setup is run per test process.
* Currently this feature has limited capability i.e. it doesn’t let you pass data around/compose ModuleSetup fixtures out of other fixtures; but there is future plan to add that capability.

# OLD MATERIAL – PLEASE IGNORE

## Setup and Cleanup

The simplest tests contain only a body, with no segregated setup and cleanup code. However, this is not practical for complex tests, especially when running under fault simulation. Also, many tests may share a common body, where only the setup and cleanup parts should be different. The TSETUP macro declares a test setup function, taking the body function as an argument. A test cleanup function may also be specified from inside the setup function, if desired.

NOTE: The test body function contains no test-specific arguments. The setup function should edit global variables and structures, and the body function should reference them. This also avoids test body functions with unreadable, impossible to maintain argument lists.

void Body(Switch &test)

{

TTRACE(L"trace test!");

TASSERT(1 == 1, L"assert test!");

}

void Cleanup(Switch &test)

{

TTRACE(L"Cleanup!");

TASSERT(StopSoapProcessor(), L"Could not stop service!");

}

TSETUP(Test1, Body)(Switch &test)

{

TTRACE(L"Setup!");

test.SetCleanup(Cleanup);

test.SetFaultMode(Automatic); //Overrides command line options.

TASSERT(StartSoapProcessor(), L"Could not start service!"); //Test body is not run if setup asserts fail.

}

During the test run, the test runs in three parts. First, the setup function runs. The setup function may configure test settings, such as changing the fault simulation mode. Then, the body function is run. If fault simulation is set to Automatic mode, the body may be run additional times with simulated faults. Finally, the cleanup function is run, if any. Setup and cleanup functions should use the assertion macros to detect errors – assertion failures during setup and cleanup cause the test to report **Error** instead of **Failed**.

## Parameters

Parameters specified on the command line may be retrieved using the Switch class. This class contains getter and setter methods for framework settings that may be retrieved or overridden. Test setup or body functions may look for custom parameters and change their behavior accordingly.

## Shared Memory

NITS is designed to record settings and test output in a cross-process shared buffer, allowing test macros and certain APIs to be used in any process that links to **nitsdll.dll**. Tests that interact with other product processes should generally link those processes to NITS on private builds. Then, instrumentation can be added to all such processes to use the tracing, assertion, and fault simulation APIs. Also, applications can register their own named shared memory sections using the **Initialize** API for easy access to any self-relative data across relevant processes. A helper class **Event** is provided for cross-process synchronization using named events, which is also used during manual fault simulation.

## Fault Simulation

The **Switch** class contains APIs for configuring fault simulation. The test body or test setup may configure manual fault simulation as it pleases, in addition to choosing whether or not to run the automatic fault simulation loop. Manual fault simulation may be configured for a specific call site ID using the methods **FaultOff**, **FaultError**, and **FaultWait**. Use the functions **ShouldFault**, **GetFaultError**, and **GetFaultWin32** to instrument product code at runtime. Defining macros to do this properly on private and public builds is recommended.

NOTE: The setup and cleanup portions of the test are excluded from automatic fault sim. Using setup and cleanup functions is preferred, since any unnecessary fault points in the body slow down the test.

Tests do not need any additional code to run automatic fault simulation. However, the test should have asserts in Manual mode for any failure conditions NOT expected under fault simulation. See TSMASSERT, TSMCOMPARE, and TSMSUBSTRING for manual mode assertions. Failing such an assertion during a test body fails the test whether or not fault simulation is running. Also, there may be certain call sites that do not work properly under fault simulation because the fault is consumed somewhere before it propagates back to the test body. This may be worked around either by failing test asserts on purpose in the product code paths that are reachable by the fault, or by adding the call site to the exclusion list using **FaultExclude**.

## Test Setting Groups

Certain test settings used by the body may be common to many tests. Instead of just defining helper functions to configure this, use the TGROUP macro to define additional test setup functions. This is particularly important when defining multiple test variations as in the next section.

Then, declare the new group as a child of every test that needs to run it using **Switch::Child** during the main test setup function or another group’s setup function. Groups may be declared in arbitrary hierarchies. Child groups must be declared by a setup function in the same order every time the setup function is run. Setup functions are run exactly once for each node in the hierarchy, in declaration order. Each setup function may declare its own cleanup function; cleanup functions are run in the opposite order after the test is complete. Declaring the same group as a child in multiple places in the same hierarchy is supported; the setup and cleanup functions run only once.

…

TGROUP(IISHosting)(Switch &test)

{

TTRACE(L"IISHosting setup!");

}

void ServiceEpilog(Switch &test)

{

TASSERT(StopSoapProcessor(), L"Could not stop service!");

}

TGROUP(InProcService)(Switch &test)

{

TTRACE(L"InProcService setup!");

TASSERT(StartSoapProcessor(), L"Could not start service!");

test.SetCleanup(ServiceEpilog); //Each setup helper may have its own cleanup function.

}

TSETUP(GroupTest, Body)(Switch &test)

{

test.Child(InProcService); //Reuse InProcService as if it were a helper function.

}

The following are some advanced notes and pointers:

* A TGROUP should not modify its own state in a way that causes the list of children to be different.
* Child setup functions run at the location where they are declared, in the order they are declared. If a TGROUP depends on settings chosen by the parent function, declare the child after choosing those settings.
* Each TGROUP may declare its own cleanup function. Cleanup functions run in the opposite order of setup functions, after the test is complete. The parent cleanup function runs before cleanup functions for its children.
* Declaring the same TGROUP as a child in multiple places in the hierarchy is supported; the setup and cleanup functions run only once. The infrastructure does not suffer from the classic multiple inheritance problem.

## Test Variations

Certain tests should be run under several different combinations of settings. This is supported using the TCHOICE macro. Whereas TGROUP nodes run all of their child setup functions at the same time, TCHOICE nodes run only one at a time. If a choice declares three children, then the test body will be run three times, each using a different child. The test framework automatically handles multiple choice points in the hierarchy, running a separate variation for each possible combination of setup functions.

…

TGROUP(IISHosting)(Switch &test)

{

TTRACE(L"IISHosting setup!");

}

void ServiceEpilog(Switch &test)

{

TASSERT(StopSoapProcessor(), L"Could not stop service!");

}

TGROUP(InProcService)(Switch &test)

{

TTRACE(L"InProcService setup!");

TASSERT(StartSoapProcessor(), L"Could not start service!");

test.SetCleanup(ServiceEpilog); //Each setup helper may have its own cleanup function.

}

//This construct declares test variations. It runs multiple children, one at a time, for each variation.

TCHOICE(HostingChoices)(Switch &test)

{

TTRACE(L"HostingChoices setup!");

test.Child(IISHosting);

test.Child(InProcService);

}

TSETUP(ChoiceTest, Body)(Switch &test)

{

test.Child(HostingChoices);

}

# Features Not Implemented

This section summarizes features and command-line options that are not currently supported but may be implemented in the future.

## XML Output

NITS should provide XML output in a form consumable by scripts or report generators.