NAME

CUnit - A unit testing framework for C

SYNOPSIS

#include <CUnit/CUnit.h>	ASSERT definitions, test management.
#include <CUnit/Automated.h>	Automated interface with xml output.
#include <CUnit/Basic.h>	Basic interface with console output.
#include <CUnit/Console.h>	Interactive console interface.
#include <CUnit/CUCurses.h>	Interactive curses interface.

DESCRIPTION

CUnit is a system for writing, administering, and running unit tests in C. It uses a simple framework for building test structures, and provides a rich set of assertions for testing common data types. CUnit is built as a static library which is linked with the user’s testing code.

STRUCTURE & GENERAL USAGE

CUnit is a combination of a platform-independent framework with various user interfaces. The core framework provides basic support for managing a test registry, suites, and test cases. The user interfaces facilitate interaction with the framework to run tests and view results.

The basic hierarchichal organization of CUnit is depicted here:

```
                    Test Registry
                          |
               -----------------------------
               |                           |
            Suite '1'      . . . .     Suite 'N'
               |                           |
         ---------------             ---------------
         |             |             |             |
      Test '11' ... Test '1M'     Test 'N1' ... Test 'NM'
```

Individual test cases are packaged into suites, which are registered with the active test registry. Suites can have setup and teardown functions which are automatically called before and after running the suite’s tests. All suites/tests in the registry may be run using a single function call, or selected suites or tests can be run.

The typical usage of CUnit is:

1. Write functions for tests (and suite init/cleanup if necessary).
2. Initialize the test registry using **CU_initialize_registry()**
3. Add test suites to the registry using **CU_add_suite()**
4. Add test cases to the suites using **CU_add_test()**
5. Run tests using the desired interface, e.g.
   CU_console_run_tests() to use the interactive console.
6. Cleanup the test registry using **CU_cleanup_registry()**

All public names in CUnit are prefixed with 'CU_'. This helps minimize clashes with names in user code. Note that earlier versions CUnit used different names without this prefix. The older API names are deprecated but still supported. To use the older names, user code must now be compiled with **USE_DEPRECATED_CUNIT_NAMES** defined.

WRITING TEST FUNCTIONS

A "test" is a C function having the signature: **void test_func(void).** There are no restrictions on the content of a test function, except that it should not modify the CUnit framework (e.g. add suites or tests, modify the test registry, or initiate a test run). A test function may call other functions (which also may not modify the framework). Registering a test will cause it's function to be run when the test is run.

CUnit provides a set of assertions for testing logical conditions. The success or failure of these assertions is tracked by the framework, and can be viewed when a test run is complete. Each assertion tests a single logical condition, and fails if the condition evaluates to CU_FALSE. Upon failure, the test continues unless the user chooses the 'xxx_FATAL' version of an assertion. In that case, the test function returns immediately.

CUnit provides a set of assertions for testing logical conditions. The success or failure of these assertions is tracked by the framework, and can be viewed when a test run is complete.

Each assertion tests a single logical condition, and fails if the condition evaluates to CU_FALSE. Upon failure, the test function continues unless the user chooses the 'xxx_FATAL' version of an assertion. In that case, the test function is aborted and returns immediately. **FATAL versions of assertions should be used with caution!** There is no opportunity for the test function to clean up after itself once a FATAL assertion fails. The normal suite cleanup function is not affected, however.

There are also special "assertions" for registering a pass or fail with the framework without performing a logical test. These are useful for testing flow of control or other conditions not requiring a logical test.

Other functions called by a registered test function may use the CUnit assertions freely. These assertions will be counted for the calling function. They may also use FATAL versions of assertions - failure will abort the original test function and its entire call chain.

The assertions defined by CUnit are:

#include <CUnit/CUnit.h>

CU_ASSERT(int expression)
CU_ASSERT_FATAL(int expression)
CU_TEST(int expression)
CU_TEST_FATAL(int expression)

Assert that expression is CU_TRUE (non-zero).

CU_ASSERT_TRUE(value)
CU_ASSERT_TRUE_FATAL(value)

Assert that value is CU_TRUE (non-zero).

CU_ASSERT_FALSE(value)
CU_ASSERT_FALSE_FATAL(value)
Assert that value is CU_FALSE (zero).

CU_ASSERT_EQUAL(actual, expected)
CU_ASSERT_EQUAL_FATAL(actual, expected)
Assert that actual == expected.

CU_ASSERT_NOT_EQUAL(actual, expected)
CU_ASSERT_NOT_EQUAL_FATAL(actual, expected)
Assert that actual != expected.

CU_ASSERT_PTR_EQUAL(actual, expected)
CU_ASSERT_PTR_EQUAL_FATAL(actual, expected)
Assert that pointers actual == expected.

CU_ASSERT_PTR_NOT_EQUAL(actual, expected)
CU_ASSERT_PTR_NOT_EQUAL_FATAL(actual, expected)
Assert that pointers actual != expected.

CU_ASSERT_PTR_NULL(value)
CU_ASSERT_PTR_NULL_FATAL(value)
Assert that pointer value == NULL.

CU_ASSERT_PTR_NOT_NULL(value)
CU_ASSERT_PTR_NOT_NULL_FATAL(value)
Assert that pointer value != NULL.

CU_ASSERT_STRING_EQUAL(actual, expected)
CU_ASSERT_STRING_EQUAL_FATAL(actual, expected)
Assert that strings actual and expected are equivalent.

CU_ASSERT_STRING_NOT_EQUAL(actual, expected)
CU_ASSERT_STRING_NOT_EQUAL_FATAL(actual, expected)
Assert that strings actual and expected differ.

CU_ASSERT_NSTRING_EQUAL(actual, expected, count)

CU_ASSERT_NSTRING_EQUAL_FATAL(actual, expected, count)
Assert that 1st count chars of actual and expected are the same.

CU_ASSERT_NSTRING_NOT_EQUAL(actual, expected, count)
CU_ASSERT_NSTRING_NOT_EQUAL_FATAL(actual, expected, count)
Assert that 1st count chars of actual and expected differ.

CU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity)
CU_ASSERT_DOUBLE_EQUAL_FATAL(actual, expected, granularity)
Assert that |actual - expected| <= |granularity|.
Math library must be linked in for this assertion.

CU_ASSERT_DOUBLE_NOT_EQUAL(actual, expected, granularity)
CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL(actual, expected, granularity)
Assert that |actual - expected| > |granularity|.
Math library must be linked in for this assertion.

CU_PASS(message)
Register a success without performing a logical test.

CU_FAIL(message)
CU_FAIL_FATAL(message)
Register a failure without performing a logical test.

THE TEST REGISTRY

The test registry is the repository for suites and associated tests. The user normally only needs to initialize the registry before use and clean up afterwards. However, other functions are provided to manipulate the registry when necessary.

The main functions needed by clients are:

#include <CUnit/TestDB.h> (included automatically by <CUnit/CUnit.h>)

CU_ErrorCode CU_initialize_registry(void)
Initializes the framework. This function should be called before any other CUnit functions. Failure to do so will likely result in a crash. An error status code is returned:

CUE_SUCCESS	if initialization is successful.
CUE_NOMEMORY	if memory allocation failed.

CU_BOOL CU_registry_initialized(void)

Checks whether the framework has been initialized. This may be useful if the registry setup is distributed over multiple files that need to make sure the registry is ready for test registration.

void CU_cleanup_registry(void)

Cleans up and releases memory used by the framework. No CUnit functions (other than **CU_initialize_registry()**) should be called after this function. Failure to call **CU_cleanup_registry()** will result in memory leaks. Note also that this function will destroy all suites (and associated tests) in the registry.

Other registry functions are primarily for internal and testing purposes. However, general users may find use for them and should be aware of them. These include:

CU_pTestRegistry CU_get_registry(void)

Retrieve a pointer to the active test registry. The registry is a variable of data type CU_Testregistry (declared in <CUnit/TestDB.h>). Note that the returned pointer will be invalidated by a call to **CU_cleanup_registry()** or **CU_initialize_registry()**

CU_pTestRegistry CU_set_registry(CU_pTestRegistry pTestRegistry)

Replace the active registry with the specified one. A pointer to the previous registry is returned. **It is the caller's responsibility to destroy the old registry.** This can be accomplished using **CU_destroy_existing_registry()** on the returned pointer. Alternatively, the old registry can be set as the active one. A subsequent call to **CU_cleanup_registry()** will then destroy it automatically. Care should be taken not to explicitly destroy a registry that is set as the active one. This will result in multiple frees of the same memory and a likely crash.

CU_pTestRegistry CU_create_new_registry(void)

Create a new registry and return a pointer to it. The new registry will not contain any suites or tests. It is the caller's responsibility to destroy the new registry by one of the mechanisms described previously.

void CU_destroy_existing_registry(CU_pTestRegistry* ppRegistry)

Destroy the specified test registry, including any registered suites. This function should not be called for a registry which is set as the active test registry. This will result in a multiple free of the same memory when **CU_cleanup_registry()** is called. ppRegistry may not be NULL, but the pointer it points to may be. Note that *ppRegistry will be NULL on return.

MANAGING TESTS AND SUITES

In order for a test to be run by CUnit, it must be added to a test collection (suite) which is registered with the test registry.

Adding Suites to the Registry

The first step in setting up a test system is creating and registering one or more test collections (suites). Each suite has a name which may be used to reference the suite. Therefore, it is recommended (but not required) that each registered suite have a unique name. The current implementation does not support the creation of suites independent of the test registry. Suites are simultaneously created and added to the active registry as follows.

#include <CUnit/TestDB.h> (included automatically by <CUnit/CUnit.h>)

CU_pSuite CU_add_suite(const char* strName, CU_InitializeFunc pInit,
CU_CleanupFunc pClean)" This creates and registers a new suite having the specified name, initialization function, and cleanup function. A pointer to the new suite is returned for use in adding tests to the suite. This pointer will be NULL if a fatal error occurs. In addition, the framework error status is set as follows:

CUE_SUCCESS	The suite was successfully created and registered.
CUE_NOREGISTRY	Error: Test Registry is not initialized.
CUE_NO_SUITENAME	Error: Suite name is not specified or NULL.
CUE_DUP_SUITE	Warning: The registry already has a suite with this name.
CUE_NOMEMORY	Error: Memory allocation failed.

The initialization and cleanup functions are optional. Both are C functions having the signature **int func_name(void).** These functions can perform setup and teardown operations needed to support the suite's tests. They are called before and after the suite's tests are run, even if only 1 of the suite's tests is run. They take no arguments, and should return NULL if they complete successfully (non-NULL otherwise). If either function is not required for a particular suite, pass NULL to **CU_add_suite().**

Adding Tests to Suites

Tests are created and added to suites. Each test has a name which may be used to reference the test later. Therefore, it is recommended (but not required) that the name be unique among all tests added to a single suite. The current implementation does not support the creation of tests independent of registered suites. Tests are simultaneously created and added to a suite as follows.

#include <CUnit/TestDB.h> (included automatically by <CUnit/CUnit.h>)

CU_pTest CU_add_test(CU_pSuite pSuite, const char* strName, CU_TestFunc
pTestFunc)" This creates a new test having the specified name and test function, and adds it to the indicated suite. The suite should have been previously created using **CU_add_suite().** A pointer to the new test is returned, which will be NULL if a fatal error occurred. In addition, the framework error status is set as follows:

CUE_SUCCESS	The test was successfully created and added.
CUE_NOREGISTRY	Error: Test Registry is not initialized.

CUE_NOSUITE	Error: Specified suite is NULL or invalid.
CUE_NO_TESTNAME	Error: Test name is not specified or NULL.
CUE_NOTEST	Error: Test function is not specified or NULL.
CUE_DUP_TEST	Warning: The suite already has a test with this name.
CUE_NOMEMORY	Error: Memory allocation failed.

Activation of Suites and Tests

A suite or test must be active to be executed during a test run (all suites and tests are active by default upon creation). The active state of a suite or test is available as pSuite->fActive and pTest->fActive, respectively. The flag will be CU_TRUE when the entity is active, CU_FALSE otherwise. Use the following functions to selectively deactivate suites and tests to choose subsets of tests to run dynamically. Note that it is a framework error to deactivate a test or suite and then specifically request that it be run.

#include <CUnit/TestDB.h> (included automatically by <CUnit/CUnit.h>)

CU_ErrorCode CU_set_suite_active(CU_pSuite pSuite, CU_BOOL fNewActive)

CU_ErrorCode CU_set_test_active(CU_pTest pTest, CU_BOOL fNewActive)

Pass CU_TRUE to these functions to activate a suite/test, CU_FALSE to deactivate it. These functions return CUE_NOSUITE and CUE_NOTEST, respectively, if the specified suite or test is NULL.

Modifying Other Attributes of Suites and Tests

Normally the attributes of suites and tests are set at creation time. In some cases, a client may wish to manipulate these to modify the test structure dynamically. The following functions are provided for this purpose, and should be used instead of directly setting the value of the data structure members. All return CUE_SUCCESS on success, and the indicated error code on failure.

CU_ErrorCode CU_set_suite_name(CU_pSuite pSuite, const char *strNewName)

CU_ErrorCode CU_set_test_name(CU_pTest pTest, const char *strNewName)

These functions change the name of registered suites and tests. The current names are available as the **pSuite->pName</I>** and **pTest->pName** data structure members. If the suite or test is NULL, then CUE_NOSUITE or CUE_NOTEST is returned, respectively. If strNewName is NULL, then CUE_NO_SUITENAME or CUE_NO_TESTNAME is returned, respectively.

CU_ErrorCode CU_set_suite_initfunc(CU_pSuite pSuite, CU_InitializeFunc pNewInit)

CU_ErrorCode CU_set_suite_cleanupfunc(CU_pSuite pSuite, CU_CleanupFunc pNewClean)

These functions change the initialization and cleanup functions for a registered suite. The current functions are available as the **pSuite->pInitializeFunc** and **pSuite->pCleanupFunc** data structure members. If the suite is NULL then CUE_NOSUITE is returned.

CU_ErrorCode CU_set_test_func(CU_pTest pTest, CU_TestFunc pNewFunc)

This function changes the test function for a registered test. The current test function is available as the **pTest->pTestFunc</I>** data structure member. If either pTest or pNewFunc is NULL, then CUE_NOTEST is returned.

Lookup of Individual Suites and Tests

In most cases, clients will have references to registered suites and tests as pointers returned from **CU_add_suite()** and **CU_add_test().** Occassionally, a client may need to be able to retrieve a reference to a suite or test. The following functions are provided to assist clients with this when the client has some information about the entity (name or order of registration). In cases where nothing is known about the suite or test, the client will need to iterate the internal data structures to enumerate the suites and tests. This is not directly supported in the client API.

CU_pSuite CU_get_suite(const char* strName)

CU_pSuite CU_get_suite_at_pos(unsigned int pos)

unsigned int CU_get_suite_pos(CU_pSuite pSuite)

unsigned int CU_get_suite_pos_by_name(const char* strName)

</P> These functions facilitate lookup of suites registered in the active test registry. The first 2 functions allow lookup of the suite by name or position and return NULL if the suite cannot be found. The position is a 1-based index in the range [1 .. **CU_get_registry()** ->uiNumberOfSuites]. This may be helpful when suites having duplicate names are registered, in which case lookup by name can only retrieve the 1st suite having that name. The second 2 functions help the client identify the position of a registered suite. These return 0 if the suite cannot be found. In addition, all these functions set the CUnit error state to CUE_NOREGISTRY> if the registry is not initialized. As appropriate, CUE_NO_SUITENAME is set if strName is NULL, and CUE_NOSUITE is set if pSuite is NULL.

CU_pTest CU_get_test(CU_pSuite pSuite, const char *strName)

CU_pTest CU_get_test_at_pos<(CU_pSuite pSuite, unsigned int pos)

unsigned int CU_get_test_pos<(CU_pSuite pSuite, CU_pTest pTest)

unsigned int CU_get_test_pos_by_name(CU_pSuite pSuite, const char *strName)

These functions facilitate lookup of tests registered in suites. The first 2 functions allow lookup of the test by name or position and return NULL if the test cannot found. The position is a 1-based index in the range [1 .. pSuite->uiNumberOfSuites]. This may be helpful when tests having duplicate names are registered, in which case lookup by name can only retrieve the 1st test having that name. The second 2 functions help the client identify the position of a test in a suite. These return 0 if the test cannot be found. In addition, all these functions set the CUnit error state to CUE_NOREGISTRY if the registry is not initialized, and to CUE_NOSUITE if pSuite is NULL. As appropriate, CUE_NO_TESTNAME is set if strName is NULL, and CUE_NOTEST is set if pTest is NULL.

RUNNING TESTS

CUnit supports running all tests in all registered suites, but individual tests or suites can also be run. During each run, the framework keeps track of the number of suites, tests, and assertions run, passed, and failed. Note that the previous results are cleared each time a test run is initiated (even if it fails).

While CUnit provides primitive functions for running suites and tests, most users will want to use one of the user interfaces. These interfaces handle the details of interaction with the framework and provide output of test details and results for the user. For more about the primitive functions, see **<CUnit/testRun.h>.**

Test Results

The interfaces present results of test runs, but client code may sometimes need to access the results directly. These results include various run counts, as well as a linked list of failure records holding the failure details. Test results must be retrieved before attempting to run other tests, which resets the result information. Functions for accessing the test results are:

#include <CUnit/TestRun.h> (included automatically by <CUnit/CUnit.h>)

unsigned int CU_get_number_of_suites_run(void)’

Retrieve the number of suites run. Suite having initialization functions which fail are not run. To get the total number of registered suites, use **CU_get_registry()->uiNumberOfSuites.**

unsigned int CU_get_number_of_suites_failed(void)

Retrieve the number of suites which had initialization or cleanup functions which failed (returned non-NULL).

unsigned int CU_get_number_of_tests_run(void)

Retrieve the number of tests run. Tests in suites having initialization functions which fail are not run. To get the total number of registered tests , use **CU_get_registry()->uiNumberOfTests.**

unsigned int CU_get_number_of_tests_failed(void)

Retrieve the number of tests which contained at least 1 failed assertion.

unsigned int CU_get_number_of_asserts(void)

Retrieve the number of CUnit assertions made during the test run.

unsigned int CU_get_number_of_successes(void)

Retrieve the number of assertions which passed.

unsigned int CU_get_number_of_failures(void)

Retrieve the number of assertions which failed.

const CU_pRunSummary CU_get_run_summary(void)

Retrieve a **CU_RunSummary** containing all the run count information. This data structure is declared in **<CUnit/TestRun.h>** and includes the (self-explanatory) *unsigned int* fields nSuitesRun, nSuitesFailed, nTestsRun, nTestsFailed, nAsserts, and nAssertsFailed.

const CU_pFailureRecord CU_get_failure_list(void)

Retrieve the head of the linked list of failure records for the last run. Each assertion failure or suite init/cleanup function failure is registered in a new **CU_FailureRecord** in the linked list. This data structure is declared in **<CUnit/TestRun.h>** and includes the following fields:

unsigned int	**uiLineNumber**
char*	**strFileName**
char*	**strCondition**
CU_pTest	**pTest**
CU_pSuite	**pSuite**

Automated Interface

The automated interface is non-interactive. The current implementation only supports running all registered suites. Results are output to an xml file to be viewed by appropriate external tools. Registered tests can also be listed to an xml file for viewing. The following public functions are available:

#include <CUnit/Automated.h>

void CU_automated_run_tests(void)

Run all tests in all registered (and active) suites. Results are output to a file named *ROOT-Results.xml.* The filename ’ROOT’ is set using **CU_set_output_filename(),** or else the default ’CUnitAutomated’ is used. This means that the same filename is used each run (and the results file overwritten) if the user does not explicitly set the ’ROOT’ for each run.

CU_ErrorCode CU_list_tests_to_file(void)

Lists the registered suites and associated tests to file. The listing file is named *ROOT-Listing.xml.* The filename ’ROOT’ is set using **CU_set_output_filename(),** or else the default ’CUnitAutomated’ is used. This means that the same filename is used each run (and the listing file overwritten) if the user does not explicitly set the ’ROOT’ for each run.

void CU_set_output_filename(const char* szFilenameRoot)

Set the filename root to use for automated results and listing files.

Basic Interface (non-interactive)

The basic interface is also non-interactive, with results output to stdout. This interface supports running individual suites or tests, and allows client code to control the type of output displayed during each run. This interface provides the most flexibility to clients desiring simplified access to the CUnit API. The following public functions are provided:

#include <CUnit/Basic.h>

CU_ErrorCode CU_basic_run_tests(void)

Run all tests in all registered suites. Only the active suites are run, and it is not considered an error if inactive suites are encountered and skipped. Returns the 1st error code occurring during the test run. The type of output is controlled by the current run mode, which can be set using **CU_basic_set_mode().**

CU_ErrorCode CU_basic_run_suite(CU_pSuite pSuite)

Run all tests in single specified suite. Returns the 1st error code occurring during the test run. **CU_basic_run_suite()** itself generates CUE_NOSUITE if pSuite is NULL, and CUE_SUITE_INACTIVE if the requested suite is not active. The type of output is controlled by the current run mode.

CU_ErrorCode CU_basic_run_test(CU_pSuite pSuite, CU_pTest pTest)

Run a single test in a specified suite. Returns the 1st error code occurring during the test run. **BU_basic_run_test()** itself generates CUE_NOSUITE of pSuite is NULL; CUE_NOTEST if pTest is NULL; CUE_SUITE_INACTIVE if pSuite is not active for execution, CUE_TEST_NOT_IN_SUITE if pTest is not a registered member of pSuite, and CUE_TEST_INACTIVE if pTest is not active for execution. The type of output is controlled by the current run mode.

void CU_basic_set_mode(CU_BasicRunMode mode)

Set the basic run mode, which controls the output during the run. Choices are:

CU_BRM_NORMAL	Failures and run summary are printed.
CU_BRM_SILENT	No output is printed except error messages.
CU_BRM_VERBOSE	Maximum output of run details.

CU_BasicRunMode CU_basic_get_mode(void)

Retrieve the current basic run mode code.

void CU_basic_show_failures(CU_pFailureRecord pFailure)

Prints a summary of all failures to stdout. Does not depend on the run mode.

Interactive Console Interface

The console interface is interactive. All the client needs to do is initiate the console session, and the user controls the test run interactively. This include selection & running of suites and tests, and viewing test results.

#include <CUnit/Console.h>

void CU_console_run_tests(void)

Initiate an interactive test run in the console.

Interactive Curses Interface

The curses interface is interactive. All the client needs to do is initiate the curses session, and the user controls the test run interactively. This include selection & running of suites and tests, and viewing test results. Use of this interface requires linking the ncurses library into the application.

#include <CUnit/CUCurses.h>

void CU_curses_run_tests(void)
Initiate an interactive test run in curses.

ERROR HANDLING

CUnit Error Status Codes

Many CUnit functions set a framework error code when an exception occurs. The error codes are an *enum* named **CU_ErrorCode** declared in header file **<CUnit/CUError.h>** (included automatically by **<CUnit/CUnit.h>**). The following functions are provided for retrieving the framework error status:

#include <CUnit/CUError.h> (included automatically by <CUnit/CUnit.h>)

CU_ErrorCode CU_get_error(void)
Returns the framework error status code.

const char* CU_get_error_msg(void)
Returns a message for the current error code.

Error Actions

By default, CUnit continues running tests when a framework error occurs. In this context, failed assertions are not considered "framework errors". All other error conditions including suite initialization or cleanup failures, inactive suites or tests which are run explicitly, etc. are included. This 'error action' can be changed by the user if desired. The following functions are provided:

#include <CUnit/CUError.h> (included automatically by <CUnit/CUnit.h>)

void CU_set_error_action(CU_ErrorAction action)
Set the framework error action.

CU_ErrorAction CU_get_error_action(void)
Retrieve the current error action.

The error actions are defined in **enum CU_ErrorAction** in header file **<CUnit/CUError.h>** (included automatically by **<CUnit/CUnit.h>**) as follows:

CUEA_IGNORE	Continue test runs on framework errors (default).
CUEA_FAIL	Stop test runs on a framework error.
CUEA_ABORT	Exit the application on a framework error.

AUTHORS

Anil Kumar <anilsaharan AT users DOT sourceforge DOT net>
Jerry St.Clair <jds2 AT users DOT sourceforge DOT net>

WEBSITE

http://cunit.sourceforge.net