NAME

atf-c, **ATF_CHECK**, **ATF_CHECK_MSG**, **ATF_CHECK_EQ**, **ATF_CHECK_EQ_MSG**, **ATF_CHECK_MATCH**, **ATF_CHECK_MATCH_MSG**, **ATF_CHECK_STREQ**, **ATF_CHECK_STREQ_MSG**, **ATF_CHECK_INTEQ**, **ATF_CHECK_INTEQ_MSG**, **ATF_CHECK_ERRNO**, **ATF_REQUIRE**, **ATF_REQUIRE_MSG**, **ATF_REQUIRE_EQ**, **ATF_REQUIRE_EQ_MSG**, **ATF_REQUIRE_MATCH**, **ATF_REQUIRE_MATCH_MSG**, **ATF_REQUIRE_STREQ**, **ATF_REQUIRE_STREQ_MSG**, **ATF_REQUIRE_INTEQ**, **ATF_REQUIRE_INTEQ_MSG**, **ATF_REQUIRE_ERRNO**, **ATF_TC**, **ATF_TC_BODY**, **ATF_TC_BODY_NAME**, **ATF_TC_CLEANUP**, **ATF_TC_CLEANUP_NAME**, **ATF_TC_HEAD**, **ATF_TC_HEAD_NAME**, **ATF_TC_NAME**, **ATF_TC_WITH_CLEANUP**, **ATF_TC_WITHOUT_HEAD**, **ATF_TP_ADD_TC**, **ATF_TP_ADD_TCS**, **atf_tc_get_config_var**, **atf_tc_get_config_var_wd**, **atf_tc_get_config_var_as_bool**, **atf_tc_get_config_var_as_bool_wd**, **atf_tc_get_config_var_as_long**, **atf_tc_get_config_var_as_long_wd**, **atf_no_error**, **atf_tc_expect_death**, **atf_tc_expect_exit**, **atf_tc_expect_fail**, **atf_tc_expect_pass**, **atf_tc_expect_signal**, **atf_tc_expect_timeout**, **atf_tc_fail**, **atf_tc_fail_nonfatal**, **atf_tc_pass**, **atf_tc_skip**, **atf_utils_cat_file**, **atf_utils_compare_file**, **atf_utils_copy_file**, **atf_utils_create_file**, **atf_utils_file_exists**, **atf_utils_fork**, **atf_utils_free_charpp**, **atf_utils_grep_file**, **atf_utils_grep_string**, **atf_utils_readline**, **atf_utils_redirect**, **atf_utils_wait** - C API to write ATF-based test programs

SYNOPSIS

#include <atf-c.h>

ATF_CHECK(*expression*);

ATF_CHECK_MSG(*expression*, *fail_msg_fmt*, ...);

ATF_CHECK_EQ(*expected_expression*, *actual_expression*);

ATF_CHECK_EQ_MSG(*expected_expression*, *actual_expression*, *fail_msg_fmt*, ...);

ATF_CHECK_MATCH(*regexp*, *string*);

ATF_CHECK_MATCH_MSG(*regexp*, *string*, *fail_msg_fmt*, ...);

ATF_CHECK_STREQ(*expected_string*, *actual_string*);

ATF_CHECK_STREQ_MSG(*expected_string*, *actual_string*, *fail_msg_fmt*, ...);

ATF_CHECK_INTEQ(*expected_int*, *actual_int*);

ATF_CHECK_INTEQ_MSG(*expected_int*, *actual_int*, *fail_msg_fmt*, ...);

ATF_CHECK_ERRNO(*expected_errno*, *bool_expression*);

ATF_REQUIRE(*expression*);

ATF_REQUIRE_MSG(*expression*, *fail_msg_fmt*, ...);

ATF_REQUIRE_EQ(*expected_expression*, *actual_expression*);

ATF_REQUIRE_EQ_MSG(*expected_expression*, *actual_expression*, *fail_msg_fmt*, ...);

ATF_REQUIRE_MATCH(*regexp*, *string*);

ATF_REQUIRE_MATCH_MSG(*regexp*, *string*, *fail_msg_fmt*, ...);

ATF_REQUIRE_STREQ(*expected_string*, *actual_string*);

ATF_REQUIRE_STREQ_MSG(*expected_string*, *actual_string*, *fail_msg_fmt*, ...);

ATF_REQUIRE_INTEQ(*expected_int*, *actual_int*);

ATF_REQUIRE_INTEQ_MSG(*expected_int*, *actual_int*, *fail_msg_fmt*, ...);

ATF_REQUIRE_ERRNO(*expected_errno*, *bool_expression*);

ATF_TC(*name*);

ATF_TC_BODY(*name*, *tc*);

ATF_TC_BODY_NAME(*name*);

ATF_TC_CLEANUP(*name*, *tc*);

ATF_TC_CLEANUP_NAME(*name*);

ATF_TC_HEAD(*name*, *tc*);

ATF_TC_HEAD_NAME(*name*);

ATF_TC_NAME(*name*);

ATF_TC_WITH_CLEANUP(*name*);

ATF_TC_WITHOUT_HEAD(*name*);

ATF_TP_ADD_TC(*tp_name*, *tc_name*);

ATF_TP_ADD_TCS(*tp_name*);

atf_tc_get_config_var(*tc*, *varname*);

atf_tc_get_config_var_wd(*tc*, *variable_name*, *default_value*);

atf_tc_get_config_var_as_bool(*tc*, *variable_name*);

atf_tc_get_config_var_as_bool_wd(*tc*, *variable_name*, *default_value*);

atf_tc_get_config_var_as_long(*tc*, *variable_name*);

atf_tc_get_config_var_as_long_wd(*tc*, *variable_name*, *default_value*);

atf_no_error();

atf_tc_expect_death(*reason*, ...);

atf_tc_expect_exit(*exitcode*, *reason*, ...);

atf_tc_expect_fail(*reason*, ...);

atf_tc_expect_pass();

atf_tc_expect_signal(*signo*, *reason*, ...);

atf_tc_expect_timeout(*reason*, ...);

atf_tc_fail(*reason*);

atf_tc_fail_nonfatal(*reason*);

atf_tc_pass();

atf_tc_skip(*reason*);

void
atf_utils_cat_file(*const char *file*, *const char *prefix*);

bool
atf_utils_compare_file(*const char *file*, *const char *contents*);

void
atf_utils_copy_file(*const char *source*, *const char *destination*);

void
atf_utils_create_file(*const char *file*, *const char *contents*, ...);

void
atf_utils_file_exists(*const char *file*);

pid_t
atf_utils_fork(*void*);

void
atf_utils_free_charpp(*char **argv*);

bool
atf_utils_grep_file(*const char *regexp*, *const char *file*, ...);

bool
atf_utils_grep_string(*const char *regexp*, *const char *str*, ...);

*char **
atf_utils_readline(*int fd*);

void
atf_utils_redirect(*const int fd*, *const char *file*);

void
atf_utils_wait(*const pid_t pid*, *const int expected_exit_status*, *const char *expected_stdout*, *const char *expected_stderr*);

DESCRIPTION

ATF provides a C programming interface to implement test programs. C-based test programs follow this template:

```
... C-specific includes go here ...

#include <atf-c.h>

ATF_TC(tc1);
ATF_TC_HEAD(tc1, tc)
{
   ... first test case's header ...
}
ATF_TC_BODY(tc1, tc)
{
   ... first test case's body ...
}

ATF_TC_WITH_CLEANUP(tc2);
ATF_TC_HEAD(tc2, tc)
{
   ... second test case's header ...
}
ATF_TC_BODY(tc2, tc)
{
   ... second test case's body ...
}
ATF_TC_CLEANUP(tc2, tc)
{
   ... second test case's cleanup ...
}

ATF_TC_WITHOUT_HEAD(tc3);
ATF_TC_BODY(tc3, tc)
{
   ... third test case's body ...
}

... additional test cases ...
```

```
ATF_TP_ADD_TCS(tp)
{
   ATF_TP_ADD_TC(tcs, tc1);
   ATF_TP_ADD_TC(tcs, tc2);
   ATF_TP_ADD_TC(tcs, tc3);
   ... add additional test cases ...

   return atf_no_error();
}
```

Definition of test cases

Test cases have an identifier and are composed of three different parts: the header, the body and an optional cleanup routine, all of which are described in atf-test-case(4). To define test cases, one can use the **ATF_TC**(), **ATF_TC_WITH_CLEANUP**() or the **ATF_TC_WITHOUT_HEAD**() macros, which take a single parameter specifying the test case's name. **ATF_TC**(), requires to define a head and a body for the test case, **ATF_TC_WITH_CLEANUP**() requires to define a head, a body and a cleanup for the test case and **ATF_TC_WITHOUT_HEAD**() requires only a body for the test case. It is important to note that these *do not* set the test case up for execution when the program is run. In order to do so, a later registration is needed with the **ATF_TP_ADD_TC**() macro detailed in *Program initialization*.

Later on, one must define the three parts of the body by means of three functions. Their headers are given by the **ATF_TC_HEAD**(), **ATF_TC_BODY**() and **ATF_TC_CLEANUP**() macros, all of which take the test case name provided to the **ATF_TC**() **ATF_TC_WITH_CLEANUP**(), or **ATF_TC_WITHOUT_HEAD**() macros and the name of the variable that will hold a pointer to the test case data. Following each of these, a block of code is expected, surrounded by the opening and closing brackets.

Program initialization

The library provides a way to easily define the test program's **main**() function. You should never define one on your own, but rely on the library to do it for you. This is done by using the **ATF_TP_ADD_TCS**() macro, which is passed the name of the object that will hold the test cases, i.e., the test program instance. This name can be whatever you want as long as it is a valid variable identifier.

After the macro, you are supposed to provide the body of a function, which should only use the **ATF_TP_ADD_TC**() macro to register the test cases the test program will execute and return a success error code. The first parameter of this macro matches the name you provided in the former call. The success status can be returned using the **atf_no_error**() function.

Header definitions

The test case's header can define the meta-data by using the **atf_tc_set_md_var**() method, which takes three parameters: the first one points to the test case data, the second one specifies the meta-data variable to be set and the third one specifies its value. Both of them are strings.

Configuration variables

The test case has read-only access to the current configuration variables by means of the *bool* **atf_tc_has_config_var**(), *const char* * **atf_tc_get_config_var**(), *const char* * **atf_tc_get_config_var_wd**(), *bool* **atf_tc_get_config_var_as_bool**(), *bool* **atf_tc_get_config_var_as_bool_wd**(), *long* **atf_tc_get_config_var_as_long**(), and the *long* **atf_tc_get_config_var_as_long_wd**() functions, which can be called in any of the three parts of a test case.

The '_wd' variants take a default value for the variable which is returned if the variable is not defined. The other functions without the '_wd' suffix *require* the variable to be defined.

Access to the source directory

It is possible to get the path to the test case's source directory from any of its three components by querying the 'srcdir' configuration variable.

Requiring programs

Aside from the *require.progs* meta-data variable available in the header only, one can also check for additional programs in the test case's body by using the **atf_tc_require_prog**() function, which takes the base name or full path of a single binary. Relative paths are forbidden. If it is not found, the test case will be automatically skipped.

Test case finalization

The test case finalizes either when the body reaches its end, at which point the test is assumed to have *passed*, unless any non-fatal errors were raised using **atf_tc_fail_nonfatal**(), or at any explicit call to **atf_tc_pass**(), **atf_tc_fail**() or **atf_tc_skip**(). These three functions terminate the execution of the test case immediately. The cleanup routine will be processed afterwards in a completely automated way, regardless of the test case's termination reason.

atf_tc_pass() does not take any parameters. **atf_tc_fail**(), **atf_tc_fail_nonfatal**() and **atf_tc_skip**() take a format string and a variable list of parameters, which describe, in a user-friendly manner, why the test case failed or was skipped, respectively. It is very important to provide a clear error message in both cases so that the user can quickly know why the test did not pass.

Expectations

Everything explained in the previous section changes when the test case expectations are redefined by the programmer.

Each test case has an internal state called ‘expect’ that describes what the test case expectations are at any point in time. The value of this property can change during execution by any of:

atf_tc_expect_death(*reason*, ...)
Expects the test case to exit prematurely regardless of the nature of the exit.

atf_tc_expect_exit(*exitcode*, *reason*, ...)
Expects the test case to exit cleanly. If *exitcode* is not ‘-1’, the runtime engine will validate that the exit code of the test case matches the one provided in this call. Otherwise, the exact value will be ignored.

atf_tc_expect_fail(*reason*, ...)
Any failure (be it fatal or non-fatal) raised in this mode is recorded. However, such failures do not report the test case as failed; instead, the test case finalizes cleanly and is reported as ‘expected failure’; this report includes the provided *reason* as part of it. If no error is raised while running in this mode, then the test case is reported as ‘failed’.

This mode is useful to reproduce actual known bugs in tests. Whenever the developer fixes the bug later on, the test case will start reporting a failure, signaling the developer that the test case must be adjusted to the new conditions. In this situation, it is useful, for example, to set *reason* as the bug number for tracking purposes.

atf_tc_expect_pass()
This is the normal mode of execution. In this mode, any failure is reported as such to the user and the test case is marked as ‘failed’.

atf_tc_expect_signal(*signo*, *reason*, ...)
Expects the test case to terminate due to the reception of a signal. If *signo* is not ‘-1’, the runtime engine will validate that the signal that terminated the test case matches the one provided in this call. Otherwise, the exact value will be ignored.

atf_tc_expect_timeout(*reason*, ...)
Expects the test case to execute for longer than its timeout.

Helper macros for common checks

The library provides several macros that are very handy in multiple situations. These basically check some condition after executing a given statement or processing a given expression and, if the condition is not met, they report the test case as failed.

The ‘REQUIRE’ variant of the macros immediately abort the test case as soon as an error condition is

detected by calling the **atf_tc_fail**() function. Use this variant whenever it makes no sense to continue the execution of a test case when the checked condition is not met. The ‘CHECK’ variant, on the other hand, reports a failure as soon as it is encountered using the **atf_tc_fail_nonfatal**() function, but the execution of the test case continues as if nothing had happened. Use this variant whenever the checked condition is important as a result of the test case, but there are other conditions that can be subsequently checked on the same run without aborting.

Additionally, the ‘MSG’ variants take an extra set of parameters to explicitly specify the failure message. This failure message is formatted according to the printf(3) formatters.

ATF_CHECK(), **ATF_CHECK_MSG**(), **ATF_REQUIRE**() and **ATF_REQUIRE_MSG**() take an expression and fail if the expression evaluates to false.

ATF_CHECK_EQ(), **ATF_CHECK_EQ_MSG**(), **ATF_REQUIRE_EQ**() and **ATF_REQUIRE_EQ_MSG**() take two expressions and fail if the two evaluated values are not equal. The common style is to put the expected value in the first parameter and the observed value in the second parameter.

ATF_CHECK_MATCH(), **ATF_CHECK_MATCH_MSG**(), **ATF_REQUIRE_MATCH**() and **ATF_REQUIRE_MATCH_MSG**() take a regular expression and a string and fail if the regular expression does not match the given string. Note that the regular expression is not anchored, so it will match anywhere in the string.

ATF_CHECK_STREQ(), **ATF_CHECK_STREQ_MSG**(), **ATF_REQUIRE_STREQ**() and **ATF_REQUIRE_STREQ_MSG**() take two strings and fail if the two are not equal character by character. The common style is to put the expected string in the first parameter and the observed string in the second parameter.

ATF_CHECK_INTEQ(), **ATF_CHECK_INTEQ_MSG**(), **ATF_REQUIRE_INTEQ**() and **ATF_REQUIRE_INTQ_MSG**() take two integers and fail if the two are not equal. The common style is to put the expected integer in the first parameter and the observed integer in the second parameter.

ATF_CHECK_ERRNO() and **ATF_REQUIRE_ERRNO**() take, first, the error code that the check is expecting to find in the *errno* variable and, second, a boolean expression that, if evaluates to true, means that a call failed and *errno* has to be checked against the first value.

Utility functions

The following functions are provided as part of the **atf-c** API to simplify the creation of a variety of tests. In particular, these are useful to write tests for command-line interfaces.

void **atf_utils_cat_file**(*const char *file, const char *prefix*)

Prints the contents of *file* to the standard output, prefixing every line with the string in *prefix*.

bool **atf_utils_compare_file**(*const char *file, const char *contents*)

Returns true if the given *file* matches exactly the expected inlined *contents*.

void **atf_utils_copy_file**(*const char *source, const char *destination*)

Copies the file *source* to *destination*. The permissions of the file are preserved during the code.

void **atf_utils_create_file**(*const char *file, const char *contents, ...*)

Creates *file* with the text given in *contents*, which is a formatting string that uses the rest of the variable arguments.

void **atf_utils_file_exists**(*const char *file*)

Checks if *file* exists.

pid_t **atf_utils_fork**(*void*)

Forks a process and redirects the standard output and standard error of the child to files for later validation with **atf_utils_wait**(). Fails the test case if the fork fails, so this does not return an error.

void **atf_utils_free_charpp**(*char **argv*)

Frees a dynamically-allocated array of dynamically-allocated strings.

bool **atf_utils_grep_file**(*const char *regexp, const char *file, ...*)

Searches for the *regexp*, which is a formatting string representing the regular expression, in the *file*. The variable arguments are used to construct the regular expression.

bool **atf_utils_grep_string**(*const char *regexp, const char *str, ...*)

Searches for the *regexp*, which is a formatting string representing the regular expression, in the literal string *str*. The variable arguments are used to construct the regular expression.

*char ** **atf_utils_readline**(*int fd*)

Reads a line from the file descriptor *fd*. The line, if any, is returned as a dynamically-allocated buffer that must be released with free(3). If there was nothing to read, returns ‘NULL’.

void **atf_utils_redirect**(*const int fd, const char *file*)

Redirects the given file descriptor *fd* to *file*. This function exits the process in case of an error and does not properly mark the test case as failed. As a result, it should only be used in subprocesses of the test case; specially those spawned by **atf_utils_fork**().

void **atf_utils_wait**(*const pid_t pid, const int expected_exit_status, const char *expected_stdout, const char *expected_stderr*)

Waits and validates the result of a subprocess spawned with **atf_utils_fork**(). The validation involves checking that the subprocess exited cleanly and returned the code specified in *expected_exit_status* and that its standard output and standard error match the strings given in *expected_stdout* and *expected_stderr*.

If any of the *expected_stdout* or *expected_stderr* strings are prefixed with ‘save:’, then they specify the name of the file into which to store the stdout or stderr of the subprocess, and no comparison is performed.

ENVIRONMENT

The following variables are recognized by **atf-c** but should not be overridden other than for testing purposes:

ATF_BUILD_CC	Path to the C compiler.
ATF_BUILD_CFLAGS	C compiler flags.
ATF_BUILD_CPP	Path to the C/C++ preprocessor.
ATF_BUILD_CPPFLAGS	C/C++ preprocessor flags.
ATF_BUILD_CXX	Path to the C++ compiler.
ATF_BUILD_CXXFLAGS	C++ compiler flags.

EXAMPLES

The following shows a complete test program with a single test case that validates the addition operator:

```
#include <atf-c.h>

ATF_TC(addition);
```

```
ATF_TC_HEAD(addition, tc)
{
   atf_tc_set_md_var(tc, "descr",
                "Sample tests for the addition operator");
}
ATF_TC_BODY(addition, tc)
{
   ATF_CHECK_EQ(0, 0 + 0);
   ATF_CHECK_EQ(1, 0 + 1);
   ATF_CHECK_EQ(1, 1 + 0);

   ATF_CHECK_EQ(2, 1 + 1);

   ATF_CHECK_EQ(300, 100 + 200);
}

ATF_TC(string_formatting);
ATF_TC_HEAD(string_formatting, tc)
{
   atf_tc_set_md_var(tc, "descr",
                "Sample tests for the snprintf");
}
ATF_TC_BODY(string_formatting, tc)
{
   char buf[1024];
   snprintf(buf, sizeof(buf), "a %s", "string");
   ATF_CHECK_STREQ_MSG("a string", buf, "%s is not working");
}

ATF_TC(open_failure);
ATF_TC_HEAD(open_failure, tc)
{
   atf_tc_set_md_var(tc, "descr",
                "Sample tests for the open function");
}
ATF_TC_BODY(open_failure, tc)
{
   ATF_CHECK_ERRNO(ENOENT, open("non-existent", O_RDONLY) == -1);
}
```

```
ATF_TC(known_bug);
ATF_TC_HEAD(known_bug, tc)
{
    atf_tc_set_md_var(tc, "descr",
                      "Reproduces a known bug");
}
ATF_TC_BODY(known_bug, tc)
{
    atf_tc_expect_fail("See bug number foo/bar");
    ATF_CHECK_EQ(3, 1 + 1);
    atf_tc_expect_pass();
    ATF_CHECK_EQ(3, 1 + 2);
}

ATF_TP_ADD_TCS(tp)
{
    ATF_TP_ADD_TC(tp, addition);
    ATF_TP_ADD_TC(tp, string_formatting);
    ATF_TP_ADD_TC(tp, open_failure);
    ATF_TP_ADD_TC(tp, known_bug);

    return atf_no_error();
}
```

SEE ALSO

atf-test-program(1), atf-test-case(4)