NAME

atf_add_test_case, **atf_check**, **atf_check_equal**, **atf_check_not_equal**, **atf_config_get**, **atf_config_has**, **atf_expect_death**, **atf_expect_exit**, **atf_expect_fail**, **atf_expect_pass**, **atf_expect_signal**, **atf_expect_timeout**, **atf_fail**, **atf_get**, **atf_get_srcdir**, **atf_init_test_cases**, **atf_pass**, **atf_require_prog**, **atf_set**, **atf_skip**, **atf_test_case** - POSIX shell API to write ATF-based test programs

SYNOPSIS

atf_add_test_case "name"
atf_check "command"
atf_check_equal "expected_expression" "actual_expression"
atf_check_not_equal "expected_expression" "actual_expression"
atf_config_get "var_name"
atf_config_has "var_name"
atf_expect_death "reason" "..."
atf_expect_exit "exitcode" "reason" "..."
atf_expect_fail "reason" "..."
atf_expect_pass ""
atf_expect_signal "signo" "reason" "..."
atf_expect_timeout "reason" "..."
atf_fail "reason"
atf_get "var_name"
atf_get_srcdir
atf_init_test_cases "name"
atf_pass
atf_require_prog "prog_name"
atf_set "var_name" "value"
atf_skip "reason"
atf_test_case "name" "cleanup"

DESCRIPTION

ATF provides a simple but powerful interface to easily write test programs in the POSIX shell language. These are extremely helpful given that they are trivial to write due to the language simplicity and the great deal of available external tools, so they are often ideal to test other applications at the user level.

Test programs written using this library must be run using the atf-sh(1) interpreter by putting the following on their very first line:

```
#! /usr/bin/env atf-sh
```

Shell-based test programs always follow this template:

```
atf_test_case tc1
tc1_head() {
    ... first test case's header ...
}
tc1_body() {
    ... first test case's body ...
}

atf_test_case tc2 cleanup
tc2_head() {
    ... second test case's header ...
}
tc2_body() {
    ... second test case's body ...
}
tc2_cleanup() {
    ... second test case's cleanup ...
}

... additional test cases ...

atf_init_test_cases() {
    atf_add_test_case tc1
    atf_add_test_case tc2
    ... add additional test cases ...
}
```

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_test_case** function, which takes a first parameter specifying the test case's name and instructs the library to set things up to accept it as a valid test case. The second parameter is optional and, if provided, must be 'cleanup'; providing this parameter allows defining a cleanup routine for the test case. It is important to note that this function *does not* set the test case up for execution when the program is run. In order to do so, a later registration is needed through the **atf_add_test_case** function detailed in *Program initialization*.

Later on, one must define the three parts of the body by providing two or three functions (remember that the cleanup routine is optional). These functions are named after the test case's identifier, and are **<id>_head**, **<id>_body** and **<id>_cleanup**. None of these take parameters when executed.

Program initialization

The test program must define an **atf_init_test_cases** function, which is in charge of registering the test cases that will be executed at run time by using the **atf_add_test_case** function, which takes the name of a test case as its single parameter. This main function should not do anything else, except maybe sourcing auxiliary source files that define extra variables and functions.

Configuration variables

The test case has read-only access to the current configuration variables through the **atf_config_has** and **atf_config_get** methods. The former takes a single parameter specifying a variable name and returns a boolean indicating whether the variable is defined or not. The latter can take one or two parameters. If it takes only one, it specifies the variable from which to get the value, and this variable must be defined. If it takes two, the second one specifies a default value to be returned if the variable is not available.

Access to the source directory

It is possible to get the path to the test case's source directory from anywhere in the test program by using the **atf_get_srcdir** function. It is interesting to note that this can be used inside **atf_init_test_cases** to silently include additional helper files from the source directory.

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_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*, or at any explicit call to **atf_pass**, **atf_fail** or **atf_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_pass does not take any parameters. **atf_fail** and **atf_skip** take a single string parameter that describes 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_expect_death "reason" "..."

Expects the test case to exit prematurely regardless of the nature of the exit.

atf_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_expect_fail "reason"

Any failure raised in this mode is recorded, but 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_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_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_expect_timeout "reason" "..."

Expects the test case to execute for longer than its timeout.

Helper functions for common checks

atf_check "[options]" "command" "[args]"

Executes a command, performs checks on its exit code and its output, and fails the test case if any of the checks is not successful. This function is particularly useful in integration tests that verify the correct functioning of a binary.

Internally, this function is just a wrapper over the atf-check(1) tool (whose manual page provides all details on the calling syntax). You should always use the **atf_check** function instead of the atf-check(1) tool in your scripts; the latter is not even in the path.

atf_check_equal "expected_expression" "actual_expression"
This function takes two expressions, evaluates them and, if their results differ, aborts the test case with an appropriate failure message. The common style is to put the expected value in the first parameter and the actual value in the second parameter.

atf_check_not_equal "expected_expression" "actual_expression"
This function takes two expressions, evaluates them and, if their results are equal, aborts the test case with an appropriate failure message. The common style is to put the expected value in the first parameter and the actual value in the second parameter.

EXAMPLES

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

```
atf_test_case addition
addition_head() {
    atf_set "descr" "Sample tests for the addition operator"
}
addition_body() {
    atf_check_equal 0 $((0 + 0))
    atf_check_equal 1 $((0 + 1))
    atf_check_equal 1 $((1 + 0))

    atf_check_equal 2 $((1 + 1))

    atf_check_equal 300 $((100 + 200))
}

atf_init_test_cases() {
    atf_add_test_case addition
}
```

This other example shows how to include a file with extra helper functions in the test program:

```
... definition of test cases ...

atf_init_test_cases() {
    . $(atf_get_srcdir)/helper_functions.sh

    atf_add_test_case foo1
    atf_add_test_case foo2
```

```
}
```

This example demonstrates the use of the very useful **atf_check** function:

```
# Check for silent output
atf_check -s exit:0 -o empty -e empty true

# Check for silent output and failure
atf_check -s exit:1 -o empty -e empty false

# Check for known stdout and silent stderr
echo foo >expout
atf_check -s exit:0 -o file:expout -e empty echo foo

# Generate a file for later inspection
atf_check -s exit:0 -o save:stdout -e empty ls
grep foo ls || atf_fail "foo file not found in listing"

# Or just do the match along the way
atf_check -s exit:0 -o match:"^foo$" -e empty ls
```

SEE ALSO

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