

IEEE Standard for Information Technology—Test Methods for Measuring Conformance to POSIX

Section 1: General

1.1 Scope

This standard is applicable to the development and use of conformance testing methodologies for POSIX standards. The generic test methods identified in this standard shall be used in conjunction with test methods identified for a specific standard.

This standard is intended for use by working groups developing test methods for POSIX standards, developers of POSIX test methods, and users of POSIX test methods.

The purpose of this standard is to define general rules for developing test assertions and related test methods for measuring conformance of an implementation to POSIX standards. Test methods may include POSIX Conformance Test Suites (PCTS), POSIX Conformance Test Procedures (PCTP), and audits of POSIX Conformance Documents (PCD).

Testing conformance of an implementation to a standard includes testing the claimed capabilities and behavior of the implementation with respect to the conformance requirements of the standard. Test methods are intended to provide a reasonable, practical assurance that the implementation conforms to the standard. Use of these test methods will not guarantee conformance of an implementation to the standard; that normally would require exhaustive testing (see 4.2.1), which is impractical for both technical and economic reasons.

21 **1.2 Normative References**

22 The following standards contain provisions which, through references in this text,
23 constitute provisions of this standard. At the time of publication, the editions
24 indicated were valid. All standards are subject to revision, and parties to agree-
25 ments based on this part of this standard are encouraged to investigate the possi-
26 bility of applying the most recent editions of the standards listed below. Members
27 of IEC and ISO maintain registers of currently valid International Standards.

28 {1} IEEE Std 729-1983, *IEEE Glossary of Software Engineering Terminology*
29 (ANSI).

30 {2} ISO/IEC 9945-1: 1990 (IEEE Std 1003.1-1990), *Information technology—*
31 *portable operating system interface (POSIX)—Part 1: System application program*
32 *interface (API) [C Language]*.

Section 2: Terminology and General Requirements

2.1 Conventions

This standard uses the following typographic conventions:

- (1) The *italic* font is used for:
 - The initial appearances of defined terms
 - Cross-references to defined terms within 2.2.1, 2.2.2, and 2.2.3
 - Parameters (option arguments and operands) that are generally substituted with real values by the application
 - C language data types and function names
 - Global external variable names
- (2) The **bold** font is used for:
 - Test result codes
 - Assertion type classification
- (3) The constant-width (Courier) font is used:
 - To illustrate examples of system input or output where exact usage is depicted
 - For references to utility names and C language headers
- (4) Symbolic *errno* names returned by functions are represented as [symbolic_name].
- (5) Symbolic constant limits are represented as {symbolic_constant_limit}.
- (6) Symbolic constant options are represented as {Option_name}.
- (7) Notes provided as parts of labeled tables and figures are integral parts of this standard (normative). Footnotes and Notes within the body of the text are for information only (informative).
- (8) Defined names that are usually in lowercase, particularly function names, are never used at the beginning of a sentence or anywhere else where regular English usage would require them to be capitalized.
- (9) Mathematical symbols such as $<$, $\leq$, etc. are only used in formulas, assertion classification assignments, and conditional clauses preceding conditional assertions.

- (10) In some cases tabular information is presented “inline”; in others it is presented in a separately labeled table. This arrangement was employed purely for ease of typesetting and there is no normative difference between these two cases.
- (11) The conventions listed above are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

A summary of typographical conventions is shown in Table 2-1.

Table 2-1 – Typographical Conventions

Reference	Example
C Language Header	<sys/stat.h>
Command Name	awk
Command Option	-c
Command Option With Argument	-w <i>width</i>
Data Types	<i>long</i>
Defined Terms	<i>file</i>
Environment Variables	PATH
Error Number	[EINTR]
File Name	filename
Function Argument Declaration	extern unsigned long int
Function Argument	<i>arg1</i>
Function Declaration	int fstat(int <i>fildev</i> , struct stat <i>*buf</i>);
Function Name	<i>func()</i>
Global External	<i>errno</i>
Implementation-dependent Limit	{MAX_INPUT}
Metavariable	<*>
Operand	<i>file_name</i>
Output	foo
Parameters	<directory pathname>
Section Reference	(see Section l)
Clause Reference	(see l.m)
Subclause Reference	(see l.n.m...)
Special Character	<newline>
Symbolic Constant Limit	{LINK_MAX}
Symbolic Constant Option	{_POSIX_JOB_CONTROL}
Table Reference	Table 6
Variable	<i>st_atime</i>

2.2 Definitions

2.2.1 Terminology

For the purposes of this standard, the following definitions apply:

2.2.1.1 may: an option for test methods.

In this standard, *need not* is used as the negative of *may*.

73 **2.2.1.2 shall:** A requirement for test methods.

74 **2.2.1.3 should:** A recommended practice for test methods.

75 **2.2.2 General Terms**

76 For the purposes of this standard, the following definitions apply:

77 **2.2.2.1 assertion:** A statement of functionality or behavior for a POSIX element
78 that is derived from the POSIX standard being tested and that is true for a con-
79 forming POSIX implementation.

80 **2.2.2.2 assertion number:** The numeric identifier assigned to an assertion.
81 The name of the element and the assertion number together uniquely identify an
82 assertion.

83 **2.2.2.3 assertion test:** The software or procedural methods that ascertain the
84 conformance of a POSIX implementation to an assertion.

85 **2.2.2.4 base assertion:** An assertion that is required to be tested for required
86 features and for implemented conditional features.

87 **2.2.2.5 conditional feature:** A feature or behavior referred to in a POSIX stan-
88 dard that need not be present on all conforming implementations.

89 **2.2.2.6 development system:** The computer system used to compile and
90 configure a PCTS.

91 **2.2.2.7 element:** A functional interface or a namespace allocation.
92 Examples of elements are C functions or utility programs. Examples of
93 namespace allocation include headers or error return value constants.

94 **2.2.2.8 extended assertion:** An assertion that is not required to be tested.

95 **2.2.2.9 POSIX Conformance Document (PCD):** The conformance document
96 required by a POSIX standard.

97 **2.2.2.10 POSIX Conformance Test Procedure (PCTP):** The nonsoftware pro-
98 cedures possibly used in conjunction with other test methods to measure
99 conformance.

100 **2.2.2.11 POSIX Conformance Test Suite (PCTS):** The collection of software
101 possibly used in conjunction with other test methods to measure conformance.

102 **2.2.2.12 required feature:** Either a single facility or behavior, or one of a pair
103 of alternative facilities or behaviors, required by a POSIX standard that is always
104 present on a conforming implementation.

105 **2.2.2.13 target system:** The combination of the computer system on which the
106 PCTS is executed and the parts of the development system that are used to gen-
107 erate the executable code of a PCTS.

108 **2.2.2.14 test method:** The software, procedures, or other means specified by a
109 POSIX standard to measure conformance.

110 Test methods may include a PCTS, PCTP, or an audit of a PCD.

111 **2.2.2.15 test result code:** A value that describes the result of an assertion test.

112 **2.2.3 Abbreviations**

113 For the purposes of this standard, the following abbreviations apply:

114 **2.2.3.1 IEEE:** The Institute of Electrical and Electronics Engineers.

115 **2.2.3.2 PCD:** POSIX Conformance Document.

116 **2.2.3.3 PCTP:** POSIX Conformance Test Procedures.

117 **2.2.3.4 PCTS:** POSIX Conformance Test Suite.

118 **2.2.3.5 POSIX:**¹⁾ Colloquial name for the collection of IEEE 1003 standards, draft
119 standards, and projects.

120 **2.2.3.6 POSIX.n:** Term used to refer to a specific IEEE P1003.n project or its pro-
121 ducts. For example, POSIX.1 represents the P1003.1 project and its associated
122 products, IEEE Std 1003.1-1988 and IEEE Std 1003.1-1990.

123 **2.2.3.7 POSIX.3:** This standard.

1) The name POSIX was suggested by Richard Stallman. It is expected to be pronounced *pahz-icks*, as in *positive*, not *poh-six*, or other variations. The pronunciation has been published in an attempt to promulgate a standardized way of referring to a standard operating system interface.

Section 3: Test Methods Conformance to This Standard

1 3.1 Conformance Criteria

2 Test methods that conform to POSIX.3 shall meet all of the following criteria:

- 3 — The test methods shall document the POSIX.n test method specifications to
4 which they conform.
- 5 — The test methods shall test base assertions for required features and imple-
6 mented conditional features of the POSIX standard for which conformance
7 is being measured. If an assertion lists a set of instances (often separated
8 by the word *or*), the assertion test shall test each specified instance.
- 9 — The PCTS shall consist of automated assertion tests to the extent possible.
- 10 — The documentation of the PCTS shall include all the information necessary
11 to install, configure, and execute the PCTS.
- 12 — The documentation of the test methods shall include, if needed, instruc-
13 tions for performing the PCTP and an audit of the PCD.
- 14 — The documentation of the test methods shall describe how to gather and
15 interpret the results.
- 16 — Additional criteria shall be specified in applicable POSIX.n standards.

Section 4: Testing Levels and Complexity Levels

4.1 Introduction

In principle, the objective of conformance testing is to establish whether the implementation being tested conforms to the specification in the relevant standard. Practical limitations make it impossible to be exhaustive, and economic considerations may restrict testing still further.

Due to these limitations, during the design and development of test methods the complexity level of an element will dictate the level of testing that satisfies conformance requirements.

Therefore, this standard distinguishes three major levels of testing, according to the extent to which they provide an indication of conformance, and three major levels of element complexity.

The three major levels of testing are:

- Exhaustive Testing
- Thorough Testing
- Identification Testing

The three major levels of element complexity are:

- Simple
- Intermediate
- Complex

4.2 Testing Levels

4.2.1 Exhaustive Testing

Exhaustive testing seeks to verify the behavior of every aspect of an element, including all permutations. For example, exhaustive testing of a given user command would require testing the command with no options, with each option, with each pair of options, and so on, up to every permutation of options.

The various command options and permutations rapidly approach numbers too large to reach execution completion in any realistic time frame. As an example, there are approximately 37 unique error conditions in POSIX.1. The occurrence of one error can (and often does) affect the proper detection of another error. An exhaustive test of the 37 errors would require not just one test per error, but one

30 test per possible permutation of errors. Thus, instead of 37 tests, billions of tests
31 would be needed (2 to the 37th power).

32 Exhaustive testing is normally infeasible.

33 4.2.2 Thorough Testing

34 Thorough testing is a useful alternative to exhaustive testing. Thorough testing
35 seeks to verify the behavior of every aspect of an element, but does not include all
36 permutations. For example, to perform thorough testing of a given command, the
37 command shall be tested with no options, then with each option individually.
38 Possible combinations of options may also be tested.

39 Given the above example concerning the 37 error conditions, thorough testing
40 would require 38 tests, a much more manageable number.

41 Thorough testing is a feasible solution. During testing of a multitude of indi-
42 vidual elements, certain combinations of subelements are coincidentally tested.

43 In this discussion, it is helpful to consider the number of assertions that can be
44 derived from the specification of each software element. Thorough testing aims to
45 verify each individual aspect in isolation and is thus much more feasible than
46 exhaustive testing. However, even thorough testing approaches infeasibility
47 when the sheer number of aspects of an element is very large. Therefore, a third
48 level of testing is defined.

49 4.2.3 Identification Testing

50 Identification testing seeks to verify some distinguishing characteristic of the ele-
51 ment in question. It consists of a cursory examination of the element, invoking it
52 with the minimal command syntax and verifying its minimal function.

53 For example, identification testing of a C compiler would distinguish it from a
54 compiler for another language on the system, but not necessarily from any other C
55 compiler on this or another system. It would not require a verification of all the
56 syntax or functions specified in the C compiler manual. Proper identification test-
57 ing of a C compiler would be to verify the minimal program constructs necessary
58 to establish its distinguishing characteristic as a C compiler.

59 4.3 Complexity Levels

60 4.3.1 Simple

61 Simple elements are those that are wholly defined in the description for that ele-
62 ment. Simple elements are those that have a few assertions to test and whose
63 functionality does not depend on other elements defined within the POSIX stan-
64 dard in which they are defined. Examples of simple elements include the cat
65 utility specified in POSIX.2 or the *close()* function specified in POSIX.1. Simple
elements should be thoroughly tested.

66 4.3.2 Intermediate

67 Intermediate elements are those that have a moderate number of assertions and
68 may depend upon the functionality of other elements defined within the POSIX
69 standard in which they are defined. Examples of intermediate elements are the
70 grep and sed utilities specified in POSIX.2, which support their own functional-
71 ity in addition to regular expressions. Thorough testing should be a goal for inter-
72 mediate elements, but it may not be feasible in some cases.

73 4.3.3 Complex

74 Complex elements are those that implement a language, depend upon the func-
75 tion of intermediate elements, or have effects on hardware devices. Typically,
76 complex elements need a large number of assertion tests to test them thoroughly.
77 Examples of complex elements are the sh and awk utilities as specified in
78 POSIX.2, which implement a language in addition to regular expressions. For
79 complex elements, thorough testing may be limited to specific areas of the
80 element.

81 4.4 Conclusion

82 It follows from the definitions of testing levels and of element complexity levels
83 that the functionality of each element must be analyzed and evaluated. Since the
84 number of possible combinations of options, events, and timing of events is unrea-
85 sonably large, testing normally cannot be exhaustive.

86 It follows from the informal nature of the definition of thorough testing that
87 thoroughness of testing will vary from one PCTS to another and, within a PCTS,
88 from one assertion to another. These are choices the PCTS implementor must
89 make. It is not within the scope of this standard to define the recommended test-
90 ing level so precisely as to preclude this degree of freedom.

Section 5: Classification of Assertions

5.1 Method of Classification

Each assertion falls into one of four categories represented by the two-by-two matrix as shown in Table 5-1.

Table 5-1 – Classification of Assertions

	Base Assertion	Extended Assertion
Required Feature	A	B
Conditional Feature	C	D

The rows of the assertion matrix correspond to whether the POSIX feature is required or conditional (see 2.2.2). The columns of the assertion matrix correspond to whether or not a test for the assertion is required. Those assertions not required to be tested are known as extended assertions. Whenever possible, the development of assertion tests for extended assertions is encouraged.

In some cases it may be appropriate to develop an assertion test that partially, but not fully, tests an extended assertion. For example, it is sometimes possible to detect failure to conform in certain instances, but not possible to detect all failures. Any assertion test that detects such failures shall produce a test result code of **FAIL** (see Section 7) for that assertion. Any assertion test that does not adequately test the assertion and that completes correctly on the system under test without detecting a failure in that system shall return a test result code of **UNTESTED**. The following is a list of reasons to classify assertions as extended:

Reason: 1 There is no known portable test method for this assertion.

Reason: 2 The corresponding statement in the POSIX standard to which conformance is being measured is not specific enough to write a portable test.

Reason: 3 There is no known reliable test method for this assertion.

Reason: 4 The assertion test requires setup procedures that involve an unreasonable amount of effort by the user of a test method.

Reason: 5 The assertion test would require an unreasonable amount of time or resources on most systems.

Reason: 6 Creating an assertion test would require an unreasonable amount of test development time.

33 Reason: 7 The assertion test could have an adverse effect on completion of a
34 test method.

35 Those assertions that are not extended are known as base assertions. Base asser-
36 tions for required features and implemented conditional features shall be tested.

37 Thus, each assertion falls into one of four categories:

- 38 (A) Base assertion of a required feature.
- 39 (B) Extended assertion of a required feature.
- 40 (C) Base assertion of a conditional feature.
- 41 (D) Extended assertion of a conditional feature.

Section 6: Writing Assertions

1 This section describes how test assertions in POSIX standards shall be written.

2 6.1 Assertion Determination

3 An assertion is written for each definitive statement in a POSIX standard that
4 pertains to an implementation. A corresponding assertion test is designed to
5 determine whether the statement is true or false for the implementation.

6 Since POSIX standards usually are written in prose style, it usually does not
7 suffice to copy a statement directly from the POSIX standard. Each assertion
8 shall be a stand-alone statement. Assertions shall be worded so that PASS
9 (see Section 7) is the result code for a conforming implementation.

10 A definitive statement is a statement in a standard that contains or implies a
11 *shall*, *should*, or *may*. This section, unless superseded by a POSIX.n stan-
12 dard, describes how statements in a POSIX standard using *shall*, *should*, and
13 *may* are interpreted to arrive at the assertions.

14 A statement in the POSIX standard that applies to the implementation and
15 either includes the word *shall* or is a declarative sentence that implies the
16 verb *shall* is interpreted as a requirement for the implementation. Such a
17 statement by itself, or in conjunction with other definitive statements,
18 corresponds to an assertion.

19 A statement that applies to a conditional feature for the implementation, that
20 uses the verbs *should* or *may*, or is a statement that implies the verbs *should*
21 or *may*, is interpreted as a requirement on the implementation, if the imple-
22 mentation supports the option. Such a statement by itself or in conjunction
23 with other definitive statements corresponds to an assertion.

24 No assertions shall be written for a statement that applies only to usage of an
25 implementation rather than to an implementation itself.

26 No assertion shall be written for a statement that uses the verbs *should* or
27 *may* when that statement is either a warning to programmers or an implemen-
28 tation recommendation.

29 **6.2 Assertion Constructs**

30 **6.2.1 Assertion Parts**

31 There are three parts to an assertion: the assertion number, the assertion
32 classification, and the actual text for the assertion.

33 **6.2.1.1 Assertion Number**

34 Each assertion is assigned an assertion number that is unique within the set of
35 assertions for a particular element. The assertions should appear in the same
36 order as the corresponding statement in the POSIX standard. They should be
37 numbered sequentially beginning with one (1) in each element.

38 New or modified assertions shall be numbered sequentially beginning with the
39 next available number. If changes to the assertion list result in unused assertion
40 numbers, those numbers are indicated at the end of the assertion list for each ele-
41 ment preceded by the phrase Unused Assertion Numbers:.

42 **6.2.1.2 Assertion Classification**

43 If an assertion corresponds to a statement that is based on a conditional feature,
44 it is classified as an assertion with type **C** or **D**, and it is written as an 'If' state-
45 ment. Otherwise, it is classified as an assertion with type **A** or **B**.

46 If an assertion corresponds to a statement that can be tested by a portable PCTS
47 in a reasonable amount of time and effort, it is classified as a base assertion (type
48 **A** or **C**). Otherwise, it is classified as an extended assertion (type **B** or **D**). The
49 reason why it is an extended assertion is provided as a number that corresponds
50 to at least one of the reasons listed in Section 5.

51 For most assertions, the assertion classification is simply **A**, **B**, **C**, or **D** (see the
52 matrix in Section 5). Sometimes the classification of base or extended is depen-
53 dent on whether or not a qualifier is true for the implementation under test. In
54 these cases, the assertion classification is written as either (**QUALIFIER?A:B**) or
55 (**QUALIFIER?C:D**). In the first case, if **QUALIFIER** is true, the required assen-
56 tion is classified as **A** (base); otherwise, the required assertion is classified as **B**
57 (extended). In the second case, if **QUALIFIER** is true, the assertion for the condi-
58 tional feature is classified as **C** (base); otherwise, the conditional assertion is
59 classified as **D** (extended).

60 **6.2.1.3 Assertion Text and Examples**

61 The actual text for each assertion follows one of the constructs described in this
62 subclause.

63 In the simplest case, the assertion is a simple statement.

64 Example:

65 01(A) The sky is blue.

66 If the assertion requires the environment to be in a particular state, a 'When'
67 clause is included, followed by the word 'then'.

68 Example:

69 02(A) When there are no clouds, then the sky is blue.

70 If the assertion is for a conditional feature, an 'If' clause containing the condi-
71 tional POSIX behavior is included, followed by a colon (':').

72 Example:

73 03(C) If the implementation provides C Standard Language-
74 Dependent system support:
75 The sky is blue.

76 In the case where an assertion for a conditional feature requires the environment
77 to be in a particular state, the above two constructs are combined.

78 Example:

79 04(C) If the implementation provides C Standard Language-
80 Dependent system support:
81 When there are no clouds, then the sky is blue.

82 If an assertion is for a required feature that has different behavior depending on
83 whether or not a particular condition is true on the implementation, an 'If ...
84 Otherwise ...' construct is used, and its classification is either A or B.

85 Example:

86 05(A) If the behavior associated with {_POSIX_JOB_CONTROL} is
87 supported:
88 The sky is blue.
89 Otherwise:
90 The sky is gray.

91 A qualifier is used when the classification of the assertion depends upon a feature
92 of the implementation. The following is an example of a conditional assertion
93 that has a qualifier to determine whether or not it is base or extended:

94 Example:

95 06({_POSIX_JOB_CONTROL}?C:D) If the implementation provides C Standard
96 Language-Dependent system support:
97 The sky is blue.

98 If a feature or behavior is similar across multiple functions defined in the same
99 section, the 'For' construct is allowed as a method of maintaining consistent asser-
100 tion numbers among functions.

101 Example:

102 07(C) For earth():
103 If the implementation provides C Standard Language-
104 Dependent system support:
105 When there are no clouds, then the sky is blue.
106 For moon():
107 If the implementation provides C Standard Language-
108 Dependent system support:
109 When there are no clouds, then the sky is black.

110 When an aspect of an implementation depends on two or more conditional
111 features, the assertions are written as multiple assertions that cover the applica-
112 ble combinations of these conditional features.

113 Example:

114 08(C) If the general terminal interface and the behavior associated with
115 {_POSIX_JOB_CONTROL} are supported:
116 The sky is blue.
117 09(C) If the general terminal interface and the behavior associated with
118 {_POSIX_JOB_CONTROL} are not supported:
119 The sky is gray.
120 10(C) If the general terminal interface is supported and the behavior
121 associated with {_POSIX_JOB_CONTROL} is not supported:
122 The sky is black.

123 If an assertion is based on a value that is too large to test or is indeterminate
124 according to a POSIX standard, a symbol beginning with the characters "PCTS"
125 and representing the testing limit for that value is used in the assertion. The
126 assertion describes the expected behavior of the feature based on the value of the
127 "PCTS_" symbol. This symbol is used by a PCTS to determine the expected
128 results for the assertion.

129 Example:

130 11(A) If {OPEN_MAX} is defined and {OPEN_MAX} < {PCTS_OPEN_MAX}:
131 The sky is blue.
132 Otherwise:
133 The sky is gray.