INTERNATIONAL STANDARD

ISO/IEC 13210

IEEE Std 2003

Second edition
1999-12-15

Information technology — Requirements and Guidelines for Test Methods Specifications and Test Method Implementations for Measuring Conformance to POSIX Standards

Technologies de l'information — Exigences et lignes directrices pour les spécifications de méthodes d'essai et les mises en œuvre de méthode d'essai pour mesurer la conformité aux normes POSIX

Reference number
ISO/IEC 13210:1999(E)
IEEE
Std 2003, 1997 edition

Abstract: This International Standard defines the requirements and guidelines for test method specifications and test method implementations for measuring conformance to POSIX standards. Test specification standard developers for other Application Programming Interface (API) standards are encouraged to use this standard. This document is aimed primarily at developers and users of test method specifications and implementations.

Keywords: assertion, assertion test, implementation under test, option, conformance document, conformance test procedure, conformance test software, test method implementation, test method specification, test result code

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN 1-55937-895-6

Contents

FIGURES

TABLES

International Standard ISO/IEC 13210: 1999 (E)

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC 13210: 1999 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC22, *Programming languages, their environments and system software interfaces*.

This second edition cancels and replaces the first edition (ISO/IEC 13210:1994), which has been technically revised.

Annexes A and B of this edition of ISO/IEC 13210 are for information only.

Participants

IEEE Std 2003-1997 was prepared by the 2003 Working Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. At the time this standard was approved, the membership of the 2003 Working Group was as follows:

Information technology—Requirements and Guidelines for Test Methods Specifications and Test Method Implementations for Measuring Conformance to POSIX® Standards

Section 1: General

1.1 Scope

This International Standard is applicable to the development and use of conformance test method specifications for POSIX standards and may be applicable to other application programming interface specifications. This International Standard is intended for developers and users of test method specifications and test method implementations.

The users of this standard include

— Assertion Writers: to format assertions

— Assertion Test Writers: to write assertion tests

— Test Suite or System Procurers: to interpret the results of test suites

The purpose of this standard is to define requirements and guidelines for developing assertions and related test methods for measuring conformance of an implementation under test (IUT) to POSIX standards. Test method implementations may include Conformance Test Software (CTS), POSIX Conformance Test Procedures (CTP), and audits of Conformance Documents (CD).

Testing conformance of an implementation to a standard includes testing the 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 7.2.1), which is impractical for both technical and economic reasons.

1.2 References

1.2.1 Normative References

The following standards contain provisions that, through references in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below.

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

1.2.2 Informative References

Several of the terms defined in 2.2.2, General Terms, have corresponding counterparts that are used in the international community for conformity assessment purposes. The references cited here use terminology and concepts that correlate to some of the terminology used in this standard.

{2} ISO/IEC 9646-1:1994, *Information technology—Open Systems Interconnection—Conformance testing methodology and framework—Part 1, General concepts.*

{3} ISO/IEC *Guide* 25:1990, *General Requirements for the Competence and Calibration of Testing Laboratories.*

1.3 Conformance Criteria

Test method specifications and implementations claiming to conform to this standard shall conform to all the requirements contained in this standard. Materials identified in this standard as *guidelines* or as *methodology* are suggestions or options, as are all examples, notes, and footnotes. All statements containing the term *shall* are requirements, as are statements where *shall* is obviously implied.

1) IEEE documents can be obtained from the The Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, PO Box 1331, Piscataway, New Jersey 08855-1331, USA.

1.3.1 Test Method Specification Conformance Criteria

A test method specification conforming to this standard shall include all the following criteria:

— It shall use the required assertion definitions, types, syntax, and constructs specified in this standard as applicable (see Section 3).

— It shall use the test result codes specified in this standard for test results defined by this standard (see Section 4).

— It shall define any test method specification structures and test result codes it uses in addition to those defined by this standard.

— It shall specify a list of conforming test result codes that indicate conformance to the specifications being tested (see 4.3.1).

A conforming test method specification may specify other constructs not specified in this standard.

1.3.2 Test Method Implementation Conformance Criteria

Test method implementations that conform to this standard shall include all the following criteria:

— A test method implementation conforming to this standard shall document that it conforms to this standard and shall document any other test method specifications to which it claims to conform.

— The test method implementation shall test each instance for an assertion that specifies a set of instances often separated by the word *or*.

— The CTS shall be automated to the maximum extent possible.

— The documentation of the CTS shall include the following:

  — All information necessary to install, configure, and execute the CTS

  — The edition of the standard being tested

  — An overview of the CTS and the optional base standard features tested by the CTS

  — Deviances from the recommendations defined within this standard

  — Known limitations of the CTS

  — Release-specific changes

  — Configuration-specific parameters

  — Development system hardware requirements such as memory, disk space, terminal, and printer needs

  — Development system software requirements

  — Hardware and software requirements necessary to execute the CTS

  — A description of any CTS trouble-clearing procedures

— The documentation of the test method implementations shall include, if needed, instructions for performing the CTP and an audit of the CD.

— The documentation of the test method implementations shall describe how to gather and interpret the results.

— Additional criteria may be specified in applicable standards.

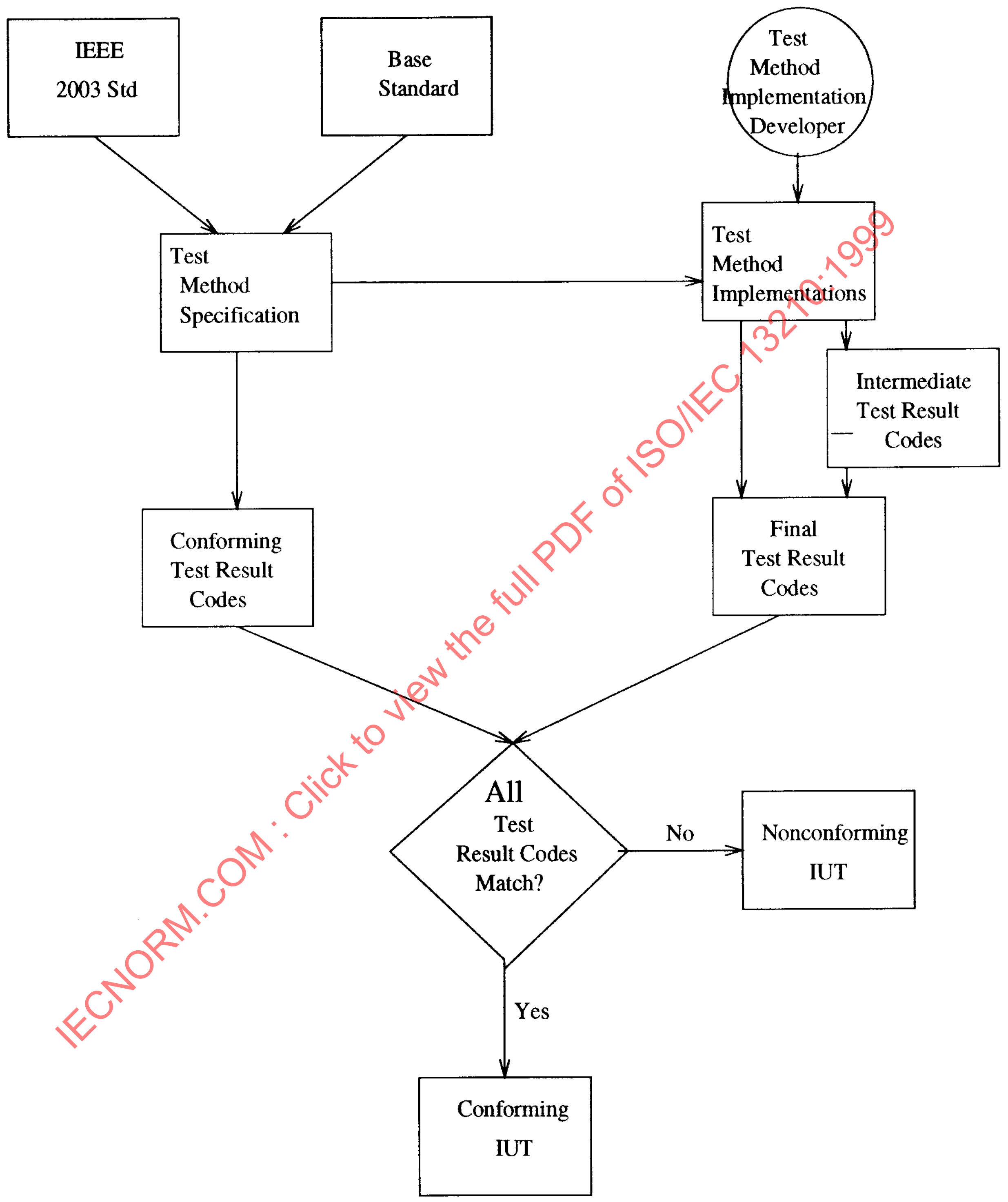
When no test method specification exists, test method implementations shall derive the test method specifications from the base standards and the test method implementation shall then meet all the criteria listed above.

1.4 IUT Conformance Assessment

Figure 1-1 depicts the basic model for conformance assessment for an IUT where a test method standard is derived from an existing base standard by using this standard. The test method standard shall identify the test result codes that are required for conformance and provides the assertions from which test method implementations are derived. When a test method implementation is executed, it provides intermediate test result codes that must be resolved to final test result codes. When the final test result codes match the conformance test result codes, the implementation is judged to be conforming.

Note: This process does not include every potential step that may be required by a certifying body, such as a conformance document audit. It addresses only the relationship between the conforming test result codes identified in the test method standard and the final test result codes derived from the test method implementation.

Figure 1-1 — Single Base Standard


IEEE 2003 Std
Base Standard
Test Method Implementation Developer
Test Method Specification
Test Method Implementations
Intermediate Test Result Codes
Conforming Test Result Codes
Final Test Result Codes
All Test Result Codes Match?
No
Nonconforming IUT
Yes
Conforming IUT

(Blank page)

Section 2: Definitions and General Requirements

2.1 Conventions

This International Standard uses the following typographical conventions:

(1) The *italic* font is used for

— The initial appearances of defined terms

— Cross-references to defined terms in 2.2.1, 2.2.2, and 2.2.3

— Parameters (option arguments and operands) that generally are substituted with real values by the application

— C language data types and function names

— Global external variable names

— The element's trailing base standard subclause reference

(2) The **bold** font is used for

— Test result codes

(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, element groupings, and C language headers

— Keywords used in assertions

(4) Symbolic test assertion parameters that are required are represented as <Test_Assertion_Parameter>.

(5) Headers are represented as `<header>`.

(6) The notation '()*' means that the production is repeated zero or more times.

(7) Symbolic test assertion parameters that are optional are represented as [Test_Assertion_Parameter].

(8) Symbolic *errno* constants returned by functions as error numbers are represented as [Symbolic_name].

(9) Symbolic constant limits are represented as {Symbolic_constant_limit}.

(10) Symbolic constant options are represented as {Option_name}.

(11) 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).

(12) 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.

(13) Mathematical symbols such as <, ≤, etc., are used only in formulas, assertion classification assignments, and conditional clauses preceding conditional assertions.

(14) 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.

(15) The double quote ("name") is used when a name is specified.

(16) The single quote ('value') is used when a specific value is specified.

(17) 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.

(18) All assertion examples are written in 9-point text.

A summary of typographical conventions is shown in Table 2.1.

Table 2-1 – Typographical Conventions

Reference	Example
C Language header	`<sys/stat.h>`
Data types	*long*
Defined terms	*file*
Environment variables	**PATH**
Error number	[EINTR]
File name	`filename`
Function argument declaration	unsigned long int
Function argument	*arg1*
Function declaration	long int
Function name	*funct*()
Global external	*errno*
IEEE Std 1003.n-19xx reference	*POSIX.n*
Implementation-dependent limit	{MAX_INPUT}
Metacharacter	* (any character string)
Parameters	<*path*>
Reference to other standards	[ISO Guide 25]
Section x.y reference	See x.y or x.y
Special character	`<newline>`
Symbolic constant limit	{LINK_MAX}
Symbolic constant option	{_POSIX_JOB_CONTROL}
Symbolic constant value	_POSIX_JOB_CONTROL
Symbol defined in header	_POSIX_JOB_CONTROL
Table reference	Table 2-1
Utility name	`tar`
Variable	*st_atime*

2.2 Definitions

2.2.1 Terminology

For the purpose of this International Standard, the following definitions apply.

2.2.1.1 may: An option for test method specifications or implementations.

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

2.2.1.2 must: The same as *shall*.

2.2.1.3 shall: A requirement for test method specifications or implementations.

2.2.1.4 should: A recommended practice for test method specifications or implementations.

2.2.2 General Terms

2.2.2.1 assertion: The specification for testing a *conformance requirement* in an IUT in the forms defined in this standard. It defines what to test and is **TRUE** for a *conforming implementation*. Assertions are the basic entities for test method specifications and test method standards.

2.2.2.2 assertion identifier: The identifier assigned to an *assertion*. It consists of characters from the *portable identifier character set* and follows the name space requirements specified in 3.3.1 and other conventions that may be specified by *test methods standards* conforming to this standard.

The name of the *element* and the *assertion identifier* together shall uniquely identify an *assertion* within a *test method specification*. See 3.3.1 for examples of assertion identifiers.

2.2.2.3 assertion test: The software or procedural methods that generate the *test result* codes used for assessment of *conformance* to an *assertion*.

2.2.2.4 base standard: The standard for which a *test method specification* is written and/or a *test method implementation* is developed.

2.2.2.5 conformance: [ISO/IEC Guide 25] Fulfillment by a product, process, or service of all relevant specified *conformance requirements*. [JTC-1]

2.2.2.6 conformance document (CD): The *conformance document* required by the standard that meets the requirements specified in that standard for such a document.

2.2.2.7 Conformance Documentation Audit: The process of reviewing a *Conformance Document* to ascertain that it meets the requirements of a *base standard* as specified by *documentation assertions*.

2.2.2.8 conformance log: [ISO/IEC 9646-1] A human-readable record of information, produced as a result of a testing session, that is sufficient to verify the assignment of *test results* (including *test verdicts*). [JTC-1, generalized]

2.2.2.9 conformance requirement: A requirement stated in a *base standard* that identifies a specific requirement in a finite, measurable, and unambiguous manner. A *conformance requirement* by itself or in conjunction with other conformance requirements corresponds to an *assertion*.

NOTE: [ISO/IEC Guide 25]
Behavior and/or capabilities imposed upon an implementation by the *base standard* for the implementation to conform to that *base standard*. [JTC-1, modified]

2.2.2.10 Conformance Test Procedure (CTP): Manual procedures used in conjunction with other test methods to measure *conformance*.

2.2.2.11 Conformance Test Software (CTS):
Test software used to ascertain *conformance* to standards.

2.2.2.12 conformance testing: [ISO/IEC 9646-1] Testing the extent to which an *implementation under test* is a *conforming implementation*. [JTC-1]

2.2.2.13 conforming implementation: [ISO/IEC 9646-1] An implementation that satisfies all relevant specified *conformance requirements*. [JTC-1, generalized]

2.2.2.14 conforming test result codes: The complete list of *test result codes* associated with each *assertion* that a CTS can report for a *conforming implementation*.

2.2.2.15 CTS build system: The hardware and software used to compile and configure a CTS.

2.2.2.16 CTS execution system: The hardware and system software on which the CTS is executed.

2.2.2.17 documentation assertion: An *assertion* generated by a requirement in the *base standard* being tested that a specific feature or behavior be documented.

2.2.2.18 element: A functional interface or a namespace allocation.

Examples of *elements* are functions and utility programs. Examples of namespace allocation include headers and error return value constants.

2.2.2.19 final test result code: A *test result code* obtained from an *assertion test* that requires no further processing.

2.2.2.20 formal test specification: A specification of the *assertion test* using a formal method specified by the *test method specification*. The *test method specification* shall specify whether the *formal test specification* is normative or informative.

2.2.2.21 implementation: That which implements the requirements of a *base standard*, or a profile.

Test method specifications shall define specifically what an implementation is within the meaning of that specification.

Implementation, as used here, is not to be confused with implementation-defined.

2.2.2.22 implementation under test (IUT): That which implements the standard(s) being tested. An IUT may consist of hardware and software located on different systems.

Test method specifications shall define specifically what an implementation is composed of within the meaning of that specification.

2.2.2.23 intermediate test result code: A *test result code*, obtained from an *assertion test*, that requires further processing to determine the *final test result code* (see 4.2.2).

2.2.2.24 logical expression: An expression of boolean terms that may be combined using logical *and* (&&) and *or* (||) operators that evaluate to either **TRUE** or **FALSE**. The resulting value is used to determine which branch of an 'If ... Else' condition to take.

2.2.2.25 option: Any behavior or feature defined in the *base standard* that need not be present in all *conforming implementations*.

2.2.2.26 portable identifier character set: The set of characters from which portable identifiers are constructed. This set shall consist only of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ ( ) -

2.2.2.27 system under test: The computer system hardware and software on which the *implementation under test* operates.

2.2.2.28 test case: [ISO/IEC 9646-1] A specification of the actions required to achieve a specific test purpose or combination of test purposes. In OSI, test cases can be generic, abstract, or executables. [JTC-1]

2.2.2.29 test method implementation: The software, procedures, or other means used to measure *conformance*. For PASC, *test method implementations* may include a CTS, a CTP, or an audit of a CD.

2.2.2.30 test method specification: A document that expresses the required functionality and behavior of a *base standard* as *assertions* and provides the complete set of conforming *test result codes*.

2.2.2.31 test method standard: A *test method specification* that has been adopted as a standard.

2.2.2.32 test purpose: [ISO/IEC 9646-1] A prose description of a narrowly defined objective of testing, focusing on a single *conformance requirement*, as specified in the appropriate product specification (i.e., verifying the support of a specific value or a specific parameter). [JTC-1]

2.2.2.33 test report: [ISO/IEC Guide 25] A document that presents *test results* and other information relevant to the execution of the *test methods* against an IUT. [JTC-1, modified]

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

This is equivalent to *test verdict* in ISO/IEC Guide 25.

2.2.2.35 test software: [ISO/IEC Guide 25] Software used in order to carry out or assist in carrying out the testing required. [JTC-1]

2.2.2.36 test support: Those facilities not specified by the standard(s) being tested, or specified but not required, that need to be provided by the SUT in order to perform an *assertion test*.

2.2.2.37 testing constant: A constant that is not specified in the standard being tested but is required by an assertion test to test an *assertion*.

2.2.2.38 verdict criteria: [ISO/IEC Guide 25] See *conforming test result codes*.

2.2.3 Abbreviations

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

2.2.3.1 IEEE: Institute of Electrical and Electronics Engineers

2.2.3.2 IUT: implementation under test

2.2.3.3 OSE: Open System Environment

2.2.3.4 PASC: Portable Applications Standards Committee

2.2.3.5 CD: Conformance Document

2.2.3.6 CTP: Conformance Test Procedures

2.2.3.7 CTS: Conformance Test Software

2.2.3.8 POSIX: Portable Operating System Interface[2)]

2.2.3.9 POSIX.n: IEEE Std 1003.n-xxxx[3)]

2.2.3.10 SUT: system under test

2) Used as a name for the collection of IEEE 1003 standards, draft standards, and projects.

3) Term used to refer to a specific IEEE P1003.n project or its products. For example, POSIX.1 represents the P1003.1 project and its associated products, ISO/IEC 9945-1: 1990, IEEE Std 1003.3-1991, and IEEE Std 2003.1-1992 {4}.

(Blank page)

Section 3: Assertion Definitions, Types, Syntax, and Constructs

3.1 Introduction

This section defines required assertion types (see 3.3) and the syntax that shall be used in writing assertions. Section 8 describes the creation of assertions from a base standard.

An assertion is a statement of functionality or behavior for one or more elements, sufficient in detail that test method implementors can code assertion tests, that is derived from one or more requirements of a base standard. Assertions are stated so that a test result code of **PASS** indicates conformance to the standard.

Test method specifications may choose to use other assertion definitions, types, syntax, and constructs. If a test method specification chooses to deviate from what is specified in this standard, it shall document in detail the assertion definitions, types, syntax, and constructs used. All assertions shall have at least two additional components associated with them:

— An assertion identifier as defined in this standard

— Conforming Test Result Codes (see 4.3.1)

An assertion may require multiple preconditions. Preconditions include all the items listed in Figure 3-1 leading up to <Test_Text>. All, or none, of the preconditions described may or may not be applicable to a given assertion.

The names defined in this section contained within the < > characters are symbolic names. They are used to characterize specific details that shall be provided in writing assertions. The details required by these symbolic names shall be provided when applicable. They shall not be used when not applicable.

A test method specification shall provide complete coverage of the base standard. The test method specification is written for the developer of the test method implementation and, when properly employed, provides an authoritative means to indicate conformance to a base standard without ambiguity. The assertions generated shall be complete, understandable, and correct.

The test methods shall be organized according to the same sections, clauses, and subclauses in the corresponding POSIX standard.

Assertions shall be written only for conformance requirements.

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

No assertion shall be written for a statement that uses definitive terminology in a statement that is meant to be only a warning to programmers or an implementation recommendation.

3.2 Generic Assertion Structure

<Generic_Assertion> refers to the generic assertion structure represented in Figure 3.2. The only required text for a <Generic_Assertion> is the <Test_Text>, except for General Assertions and General Documentation Assertions that require both the 'For' construct and the <Test_Text>.

Figure 3-1 – Generic Assertion Structure

```
For <Element_1>, ..., <Element_n>:
If <Applicable_Standard> then
  If <Option> then
    If <Test_Support> then
          (Setup:   <Setup_Requirements>)*
          Test:     <Test_Text>
          (TR:      <Testing_Requirements>)*
          (Note:    <Notes>)*
     Else <No_Test_Support>
   Else <No_Option>
Else <No_Applicable_Standard>
```

This structure provides the basis for each of the assertion types defined in this section. The specification for each line of the generic assertion structure shall be identified when applicable. <Test_Text> is always applicable to a generic assertion.

Test method specifications are allowed flexibility in the representation of specific terms identified in this structure; however, such representation shall be editorially consistent. For example, else may be represented as else, ELSE, ELSE:, Else, etc., as long as the representation is consistent.

If, in traversing an assertion from top to bottom, a precondition is not supported by the SUT or IUT, then the outcome for the assertion is mandated by its corresponding 'Else' specification.

3.2.1 For

If a feature or behavior is similar across multiple elements, the `'For'` construct is allowed as a method of specifying identical requirements to multiple elements in an accurate and concise manner. This structure normally is used in writing general assertions.

`'For'` may be used as a looping construct similar to that found in programming languages. It may list a set of functions or constants that are to be substituted for parameters referenced in the body of the assertion.

3.2.2 If then Else

The construct 'If <precondition>, then ... Else <outcome>' is used to specify a requirement needed to test an assertion. This is represented in the assertion structure by the 'If <precondition>' clause when the requirement is not met by the 'Else <outcome>' clause.

When the precondition is supported by the SUT, the 'If' clause evaluates as **TRUE**. When the precondition is not supported by the SUT, the 'If' clause evaluates as **FALSE** and the corresponding 'Else' clause determines the outcome to be reported.

This construct provides a straightforward approach to assigning an outcome to an assertion when a required precondition is not provided.

3.2.3 Applicable Standard

<Applicable_Standard> represents the base standards applicable for testing this assertion. <Applicable_Standard> may simply be the single base standard from which the assertion is derived or multiple base standards to which the assertion applies. This parameter may be represented in a logical expression.

3.2.4 Option

<Option> represents any behavior or feature defined in the base standard that need not be present in all conforming implementation. This parameter may be represented as a logical expression.

3.2.5 Test Support

<Test_Support> represents those facilities not specified, or specified but not required, by the standard(s) being tested that are needed by an SUT to perform an assertion test. Test method specifications may need the support of nonrelated optional features in the system under test in order to test an implementation thoroughly.

3.2.6 Setup Requirements

<Setup_Requirements> are the steps that a test program or tester must perform to set up the proper environment for performing the test of an assertion. <Setup_Requirements> may have one or more steps. <Setup_Requirements> are instructions to create the appropriate test environment. If a setup requirement step does not succeed, a conforming test method shall report which step failed and, to the largest extent possible, the reason for failure. If any setup step does not succeed, the test result code reported for the assertion shall be **UNRESOLVED**.

3.2.7 Test Text

<Test_Text> specifies the test to be performed. The text usually takes the form

<action> <result>

When an assertion requires multiple tests, a list or table may be provided to define the minimum number of items to be tested.

3.2.8 Testing Requirements

<Testing_Requirements> specifies the minimal testing that is required for an assertion. It typically is specified when an assertion allows more than one way to test it. For example, an assertion having to do with some property of a file could be tested for a single type of file, or several types, or all types for which the property is valid. A testing requirement should be used to specify the required minimum set of file types to test in such a circumstance.

<Testing_Requirements> may depend on the context of the assertion. Thus it is possible for a single assertion to have multiple testing requirements each of which is applicable to a specific assertion context.

When an assertion requires many tests in order to test it thoroughly, a list or table containing the test parameters may be provided in place of repetitive text.

3.2.9 Notes

<Notes> represent additional information that should be known to those using the test method specification.

3.3 Assertion Types and Constructs

Depending on the text of the base standard, each assertion structure shall have one of five possible forms to correspond to the five assertion types. These assertion types are

— Basic

— General

— Reference

— Documentation

— General Documentation

Each of these assertion structures is based on the generic assertion structure shown in Figure 3-1.

3.3.1 Assertion Identifiers

Each assertion has an assertion identifier that is indicated in the assertion structure by any of the symbols <Assertion_Identifier>, <D_Assertion_Identifier>, <GA_Assertion_Identifier>, <GD_Assertion_Identifier>, <R_Assertion_Identifier>. The guideline for writing assertion identifiers, indicated by the symbol <Assertion_Identifier>, recommended by this standard is as follows: If other conventions are used, they shall be specified by test methods specifications conforming to this standard:

[(<specification_list>)]<Assertion_Type><portable_identifier_chars>

Where <specification_list> is a comma-separated list of standards and specifications to which the rest of the assertion identifier applies; it is optionally present and is enclosed in parentheses if it is present. The <Assertion_Type> indicates the type of the assertion and is one of the types specified below or another type specified by the test methods specification in which it appears. The <portable_identifier_chars> are the set of characters from the portable identifier character set that give the assertion its identification or name. Note that there are no spaces allowed in an <Assertion_Identifier> and that the only places where commas may appear are in the <specification_list>.

The following characters are reserved for <Assertion_Type> and shall be used only for the type of assertion indicated; the indicated symbol as used in this standard indicates the specified type of <Assertion_Identifier>. Test method specifications shall define the assertion identifier conventions employed. See Figure 3-2.

Figure 3-2 — Assertion Types

<Assertion_Type>	Type of Assertion	Assertion Identifier Symbol
D_	Documentation Assertion	<D_Assertion_Identifier>
GA_	General Assertion	<GA_Assertion_Identifier>
GD_	General Documentation Assertion	<GD_Assertion_Identifier>
R_	Reference Assertion	<R_Assertion_Identifier>
<NULL>	Basic Assertion	<Assertion_Identifier>

Examples of assertion identifiers that meet the recommended guidelines are

23
GA_17
GA_stdC_proto_decl
D_5
17.35
R_erofs
(2003.1-1990)GA_10 - Means GA_10 in IEEE Std 2003.1-1990 standard.
(2003.1-1990, 2003.4-1995)15 - Means assertion 15 in both standards stated.

The following subclauses specify the layout for each assertion type. Section 4 provides examples for writing assertions for each assertion type by using the actual text from ISO/IEC 9945-1: 1990 to derive assertions and provide assertion layouts.

3.3.2 Basic Assertion

The symbolic <Basic_Assertion> represents one or more standard requirements with respect to a single element. A <Basic_Assertion> shall be written to conform to the assertion structure of Figure 3-3.

Figure 3-3 – Basic Assertion Structure

```
<Assertion_Identifier> <Generic_Assertion>
```

3.3.2.1 Assertion Identifiers for Basic Assertions

The <Assertion_Identifier> is unique to each <BASIC_ASSERTION> within an element. These identifiers should be numeric and may contain a decimal point and a fractional part. The assertion associated with such an identifier should be placed in numeric sequence relative to other numbered assertions. Assertions with non-numeric assertion identifiers may be placed in any sequence, including interspersed between assertions with numeric assertion identifiers. The use of a numeric assertion identifier shall not imply a particular sequence for testing assertions.

3.3.3 General Assertions

General assertions are statements of behavior or functionality derived from a base standard that apply to more than one element and expand into one or more assertions specific to each element.

General assertions shall be uniquely identified within a test method specification. The test method specification shall document in an index the location of each general assertion.

The symbolic <General_Assertion> represents the standard's requirements that affect multiple elements in a similar manner. Each general assertion results in one or more assertions for each of the elements listed or implied. General assertions shall be written to conform to the assertion structure of Figure 3-4. The 'For' construct shown in Figure 3-1, Generic Assertion Structure, is required for each General Assertion.

Figure 3-4 – General Assertion Structure

```
<GA_Assertion_Identifier>
    <Generic_Assertion>
```

For each element listed or implied by a general assertion, a basic assertion will be generated for that element using the structure shown in Figure 3-3 and a reference to its corresponding <General_Assertion>. These basic assertions shall be written to conform to the assertion structure of Figure 3-5.

Figure 3-5 – Assertion Derived from a General Assertion

```
<Assertion_Identifier>  <Generic_Assertion>
See: <GA_Assertion_Identifier >
```

3.3.3.1 Assertion Identifiers for General Assertions

Assertion identifiers for <General_Assertion>s are unique within a base standard. It is recommended that <GA_Assertion_Identifier> be given meaningful names such as the following:

GA_stdC_proto - General assertion for C Standard {3}[4] prototype declaration

GA_commC_result - General assertion for common-usage C result declaration

3.3.4 Reference Assertions

The symbolic <Reference_Assertion> represents a means to avoid performing the same assertion test multiple times for the same element. A reference assertion is used when a base standard repeats a requirement in the same element. The test method writer states the basic assertion once in the element, and the other places where the text is replicated in the element are stated as reference assertions. Reference assertions are used only when an assertion already exists within an element and shall be referenced only to assertions within that element. Reference assertions may refer to more than one existing assertion. Reference assertions shall contain text derived from the standard being covered.

Reference assertions shall be written to conform to the assertion structure of Figure 3-6.

Figure 3-6 – Reference Assertion Structure

```
<R_Assertion_Identifier>
  (Setup:   <Setup_Requirements>)*
  Test:     <Test_Text>
  See:      <Assertion_Identifier>
```

NOTE: The only optional text for <Reference_Assertion> is the 'Setup:' line.

Reference assertions cannot be written as general assertions, and vice versa. Reference assertions refer to assertions that correspond to text in base standards that exist for the sections referenced. General assertions, when expanded, do not have the text in the base standard.

4) The numbers in curly brackets correspond to those of the references in 1.2.

3.3.4.1 Assertion Identifiers for Reference Assertions

Assertion identifiers for <R_Assertion_Identifier> are unique within each element. These identifiers should be numeric following the prefix and may contain a decimal point and a fractional part. The assertion associated with such an identifier should be placed in numeric sequence relative to other numbered assertions. Assertions with nonnumeric assertion identifiers may be placed in any sequence, including interspersed between assertions with numeric assertion identifiers. The use of a numeric assertion identifier shall not imply a particular sequence for testing assertions.

3.3.5 Documentation Assertions

Documentation Assertions are required in test method specifications when the base standard explicitly requires that certain features be documented.

Standards may require that a vendor's conformance document specify the documentation assertions such that the document is organized similarly to the standard.

Information may appear under a higher-level heading or in a section, clause, or subclause whose content in the standard is intended to cover multiple sections, clauses, or subclauses throughout the standard.

Each documentation assertion shall state the section, clause, or subclause where the documentation shall reside. When assertions are dependent on the Conformance Document for an announcement mechanism about the support or nonsupport of options, the assertions should use the symbol {CD_*}, where "*" is uniquely expanded. The {CD_*} symbols should reside in a table that specifies the option associated with each symbol. This type of documentation provides a rational approach for handling the myriad of options in writing assertions.

Documentation assertions are used to identify Conformance Document requirements. Documentation assertions shall be written to conform to the assertion structure format shown in Figure 3-7.

Figure 3-7 – Documentation Assertion Structure

<D_Assertion_Identifier> <Generic_Assertion>

POSIX standards usually require that certain features be documented in a conformance document. When required, this document is referred to as a POSIX Conformance Document (PCD). In writing a PCD, all documentation requirements of the base standard shall be met. If a test method specification exists, the PCD shall contain all the documentation required by the documentation assertions. Information provided in a PCD shall be directly traceable to a requirement for documentation in the base standard.

3.3.5.1 Assertion Identifiers for Documentation Assertions

Assertion identifiers for <Documentation_Assertion>s are unique within each element. These identifiers should be numeric following the prefix and may contain a decimal point and a fractional part. The assertion associated with such an identifier should be placed in numeric sequence relative to other numbered assertions. Assertions with nonnumeric assertion identifiers may be placed in any sequence, including interspersed between assertions with numeric assertion identifiers. The use of a numeric assertion identifier shall not imply a particular sequence for testing assertions.

3.3.6 General Documentation Assertions

General documentation assertions affect multiple elements in a similar manner. Each general documentation assertion results in one or more documentation assertions for each of the applicable elements.

Each assertion that is derived from a general documentation assertion contains a reference to the general documentation assertion from which it was derived. General documentation assertions shall be written to conform to the assertion structure of Figure 3-8. The 'For' construct shown in Figure 3-1, Generic Assertion Structure, is required for each General Documentation Assertion.

Figure 3-8 – General Documentation Assertion Structure

```
  <GD_Assertion_Identifier>

     <Generic_Assertion>
```

For each element listed or implied by a general documentation assertion, a documentation assertion shall be generated for that element using the structure as shown in Figure 3-9. The 'See:' provides a reference to its corresponding general documentation assertion.

Figure 3-9 – Documentation Assertion from a General Documentation Assertion

```
  <Assertion_Identifier>  <Generic_Assertion>
     See: <GD_Assertion_Identifier>
```

3.3.6.1 Assertion Identifiers for General Documentation Assertions

Assertion identifiers for <GD_Assertion_Identifier>s are unique within each test method specification.

It is recommended that <General_Documentation_Assertion> assertion identifiers be given meaningful names such as the following:

GD_stdC_proto - General documentation assertion for C Standard {3} prototype declaration

GD_commC_result - General documentation assertion for common-usage C result declaration

3.3.7 Unused Assertion Identifiers

When a test method specification developer wants to keep track of assertion identifiers that were previously used but are unused in the current draft or specification, the assertion identifiers may be listed at the end of each element or may be specified in sequence with the word **Unused** in place of an assertion, as shown in Figure 3-10.

Figure 3-10 – Examples of Unused Assertion Identifiers

```
<Assertion_Identifier> Unused
  Or:
Unused <Assertion_Identifier>: (,<Assertion_Identifier>)*
```

3.4 Macros

3.4.1 Introduction

Since test methods require precise specifications and since assertions contain many repetitive phrases, the use of a macro mechanism may be employed to group the common text of base standards in producing assertions. The specifics of any macro mechanism used shall be specified in the test method specification.

3.4.2 Macro Naming Convention and Usage

The prefix "M_" shall be reserved for naming macros. Macro names shall be made up of characters from the portable identifier character set except for the parentheses and dash, '(', ')', '-' and shall begin with "M_". Macros should be given meaningful names descriptive of their purpose. In addition, a macro that is used only within an element should contain the element name in its name to help give the reader information about its scope. Macros that apply across a base standard should not have an element name.

NOTE: For example, a macro could be defined and used in the following way:

M_GA_stdC_proto_decl	name of the macro that contains the text of the <General_Assertion> for C Standard {3} prototype declarations
M_fork_extra_inherit	name of the macro that deals with extra items inherited in a *fork*() call

A macro definition might look like the following:

```
M_GA_stdC_proto_decl(hdr,proto) =
    If PCTS_C_standard Then
        Setup: The header <hdr> is included.
        Test:  The function prototype proto is declared.

    Else NO_OPTION
    Note: GA_stdC_proto_declaration
```

Such a definition would yield the following definition for a <General_Assertion> covering Standard C prototype declarations:

```
GA_stdC_proto_decl  M_GA_stdC_proto_declaration(header,synopsis_prototype)
```

The way such a <General_Assertion> would appear in an element is as follows (note that it has 1 as its assertion identification):

```
1      M_GA_stdC_proto_decl(semaphore.h,
              "int sem_init(sem_t *sem, int pshared, unsigned int value);")
```

The above definition of assertion '1' was done completely with a macro call using parameters. The meaningful name tells the reader immediately that it is a macro (M_) for the General Assertion (GA_) covering the Standard C prototype declaration.

3.5 Summary

The exact structure for assertions shall vary according to the requirements of the base standard. Many of the preconditions and other symbolic names described here may be applicable to a given assertion. However, all assertions shall have at least three components:

— An assertion identifier i.e., <Assertion_Identifier>

— An assertion test, i.e., <Test_Text>

— Conforming test result codes (see 4.3.1)

(Blank page)

Section 4: Test Result Codes

4.1 Introduction

This section contains the requirements for test result codes used by test method implementations and test method specifications.

4.2 Test Method Implementations

There shall be only two types of test result codes for test method implementations: final and intermediate. Test method implementations may use additional intermediate test result codes to provide the user with more information on why a specific test result code was issued. Test method implementations shall not give any other meaning to the test result codes specified in this standard and are required to use these test result codes when applicable. Test method implementations shall not define any meaning for final test result codes other than what is defined in this standard.

4.2.1 Intermediate Test Result Codes

An intermediate test result code is one that requires further processing to determine the final result code. The intermediate test result codes are as follows:

— **UNRESOLVED** - At least one of the following conditions is true:

[a] The proper setup state was never achieved.

[b] The assertion test requires manual inspection in order to determine its result.

[c] The test program software for an assertion was unexpectedly interrupted.

[d] The assertion test could not be executed because a previous assertion test on which it depended failed.

[e] The test program containing the assertion test was not initiated.

[f] Translation or execution of the test program produced unexpected errors or warnings.

[g] The assertion test did not resolve to a final test result code for any other reason.

All occurrences of **UNRESOLVED** test result codes shall resolve to one of the allowable final test result codes before a statement of conformance (see Section 6) is made.

If an unexpected event occurs while one is determining the result of an assertion test that could invalidate the test result, the test method shall identify it to the extent possible and report such an occurrence.

If the assertion test cannot be completed after such an occurrence, the test result code shall be **UNRESOLVED**. If the test method implementation is a CTS, it should strive to continue to execute test programs but shall report any assertion test that could not be executed as **UNRESOLVED**. The CTS should attempt to recover from the unexpected event in a manner that maximizes the execution of assertion tests while ensuring the validity and correctness of subsequent test results.

— **INCOMPLETE** - The test of the assertion was unable to prove **PASS** but encountered no **FAIL**s.

4.2.2 Final Test Result Codes

A final test result code is one that requires no further processing to determine the result of testing an assertion. When a test result code is other than **PASS** or **UNTESTED**, the test method shall provide, to the extent possible, sufficient information to determine the cause of the result. The final test result codes are

— **PASS** - The IUT meets the requirements as specified by an assertion.

— **FAIL** - The IUT does not meet the requirements as specified by an assertion.

— **NO_APPLICABLE_STANDARD** - The Standard required to test the assertion is not supported by the IUT.

— **NO_OPTION** - The base standard option required to test the assertion is not supported by the IUT.

— **NOT_APPLICABLE** - The assertion does not apply to this profile.

— **NO_TEST_SUPPORT** - The hardware or software needed to support the testing of the assertion is not available on the IUT.

— **UNTESTED** - There is no assertion test for this assertion.

The **UNTESTED** test result code is allowed when a test assertion cannot be portably tested. The ability to determine whether an assertion is portably testable depends on a number of factors:

- The ability of the test method writers to specify the assertions so that they are testable
- The ability of the test method writers to concoct portable test scenarios for test assertions
- The base standards required by the test method standard

The assignment of an **UNTESTED** test result code to an assertion should be used only after all attempts to write the assertion as a testable assertion have failed. In most cases, nonportable assertions can be tested for specific implementations. Test suite writers should provide, to the extent possible, tests for nonportable assertions.

The complete set of reasons why an assertion is allowed the test result code of **UNTESTED** is as follows:

(1) There is no known portable test method for this assertion.

A portable test cannot be written for most conforming systems because the actual software may change for different target systems that the CTS supports. This is the case when some software may have to be customized for the system under test and is likely to be different for each system under test.

The testing organization providing the CTS must provide a procedure to be followed when an implementation cannot portably test an assertion.

(2) The corresponding statement in the base standard to which conformance is being measured is not specific enough to write a portable test.

The POSIX standard was not clear in the specification of *features* from a software testing viewpoint. An example of this is in POSIX.1, where it is stated that the *stat.h* data items shall have meaningful values. A CTS would have great difficulty determining the test software to verify that a *stat* structure contains meaningful values. It may be that supplements developed for the POSIX standards can address these statements in the applicable POSIX standard.

(3) There is no known reliable test method for this assertion.

The developers of this standard were unable to determine a software or procedure test to determine whether the assertion did what was stated in the applicable base standard. An example of this case can be seen in trying to determine elapsed clock time for a particular function in a standard.

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

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

(6) Creating an assertion test would require an unreasonable amount of test development time.

(7) The assertion test could have an adverse effect on the completion of a test method.

The reason, the specific number from one of those listed above, for allowing a test result code of **UNTESTED** is provided whenever the **UNTESTED** test result code is allowed (see Figure 4-1).

Additional final test result codes shall be used only in situations for which none of the above final test result codes apply. Any additional final test result codes used by a test method shall be documented in the test method documentation.

Test methods shall not attach any other meaning to the test result codes described above.

4.3 Test Method Specifications

4.3.1 Conforming Test Result Codes

Conforming test result codes, (see Figure 4-1), are those which are acceptable for an assertion to demonstrate conformance to a base standard. Each test method specification shall state for each assertion its set of conforming test result codes.

Examples of these codes associated with their assertions are provided (see 9.1.1).

Figure 4-1 – Entity versus Allowable Test Result Code

Entity	Test Result Code
Applicable_Standard	<No_Applicable_Standard>
Option	<No_Option>
Test_Support	<No_Test_Support>
Test	PASS, UNTESTED

When a set of assertions have mutually exclusive preconditions, the final test result code of **PASS** for an IUT may occur at most once for the assertions in this set. We recommend that mutually exclusive assertions be flagged as a reminder that only one final test result code may be **PASS**.

A guideline for flagging mutual exclusive assertions is a notation of
PASS[<Assertion_Identifier>, ... <Assertion_Identifier>]
See Figure 9-1 and 9.1.2 assertions 01-02 and 13-15 for actual examples.

Section 5: Test Report

5.1 Test Report

The results of the execution of the test method implementation against an IUT may be summarized in a Test Report. The Test Report shall contain the following information:

— The name and edition of the standard to which conformance is being measured

— The names and version numbers of the test method implementations used

— The test method specifications to which the test method implementations conform

— The name, model, and configuration of the computer systems tested and the name, version, and release level of the implementation stated in terms of the identification scheme of the implementor

— The name and version of the audited CD (if a CD is required by the standard)

— The date the IUT was tested

In addition, the following information shall be available:

— The test result for each assertion test

— A description of any modifications made to the test methods

— Information on how to reproduce the test results

An IUT conforms to the standard or profile when each final test result code obtained from the test method implementation matches a conforming test result code from the corresponding assertion of the associated test method standard.

5.2 CD Audit

When documentation assertions require *details*, the details shall be provided. System documentation may be referenced in place of providing the details, but such references shall be appropriate.

Some standards require that the structure of the CD from the vendor be similar to the associated base standard. Allowed and required CD information may appear under a higher-level heading or in a section, clause, or subclause whose content in the associated base standard is intended to cover multiple sections, clauses, or subclauses.

If the base standard does not specifically require the structure of the CD to match the standard, other formats are allowed; however, it is strongly recommended that the structure of the standard be followed as closely as possible.

Section 6: Profiles

6.1 Definition

A Profile makes explicit the relationships within a set of base standards when they are used together and also may specify particular details for each base standard used. A Profile may refer to other International Standardized Profiles (ISP) in order to make use of the functions and interfaces already defined by them and thus limit its own direct reference to base standards.

Thus a Profile

- — Specifies the base standards that apply and may restrict the implementation of specific behaviors and features to maximize application portability
- — Shall not specify any requirements that would contradict or cause nonconformance to a base standard to which it refers
- — May require a larger minimum-maximum value or a smaller maximum minimum value for variables specified in a referenced base standard
- — May require certain interactions between base standards (which shall usually be some form of requirement that they work together)
- — May contain conformance requirements that are more specific and limited in scope than those of the the base standards to which it refers

6.2 Conformance to a Profile

Profiles incorporate base standards by reference as part of their specification. In addition, a profile may specify features or behaviors to be required to be implemented in a base standard and less restrictive limits and sizes for items specified in a base standard. A profile also may specify requirements to fill in gaps between the base standards it incorporates, such as interoperability between implementations of base standards.

Thus, for an implementation to conform to a profile, it shall conform to all the base standards incorporated in a profile as well as all the requirements specified in the profile itself. The requirements for conformance to a profile are defined in the profile itself. Test methods, possibly specified separately from the profile itself, shall be used to measure conformance to a profile.

6.2.1 Profile Test Methods

Profile test methods are the combinations of test methods for each base standard incorporated in the profile and test methods for those requirements specified in the profile itself.

6.2.2 Base Standard Test Methods

If there is no standard or recognized way to measure conformance to a particular base standard, the profile test method shall specify that the base standard does not apply to the measurement of conformance to the profile. It should be noted that there are some recognized ways to measure conformance to standards that do not have test methods specifications; for example, some National Bodies have certification programs to validate conformance of implementations of the C Standard {3}, but there are no test methods specifications for the C Standard {3}.

Profile test methods shall specify the standard or recognized way of measuring conformance to a base standard for those base standards which have one. If the test methods for a base standard follow the requirements of this standard for specifying test methods, the profile test methods shall also specify a modified conformance matrix for the base standard, with the only changes being those needed to further restrict the set of conforming assertion result codes to meet the requirements of the profile. When test methods are applied to measure conformance to a base standard (for example, using test software or test procedures), the application shall be done to be consistent with the requirements of the profile. For example, test software could be configured to use the limits specified in the profile rather than those in the base standard in cases where there is a difference. Or test software could be configured to use the options specified in a base standard that are required in the profile.

6.2.3 Profile-Specific Test Methods

Test method specifications or implementations that contain requirements unique to a profile are called profile-specific. Such requirements may include the requirements of options in a base standard, less restrictive limits and sizes for items defined in a base standard, and interoperability requirements between implementations of base standards.

Profile-specific assertions shall satisfy the requirements for this standard.

6.3 Conformance Assessment

Two models, of many possible, for assessing conformance to profiles are presented here. These are derived from the Single Base Standard model presented in 1.4 and should serve as a guideline for certifying bodies conducting conformance assessment for profiles.

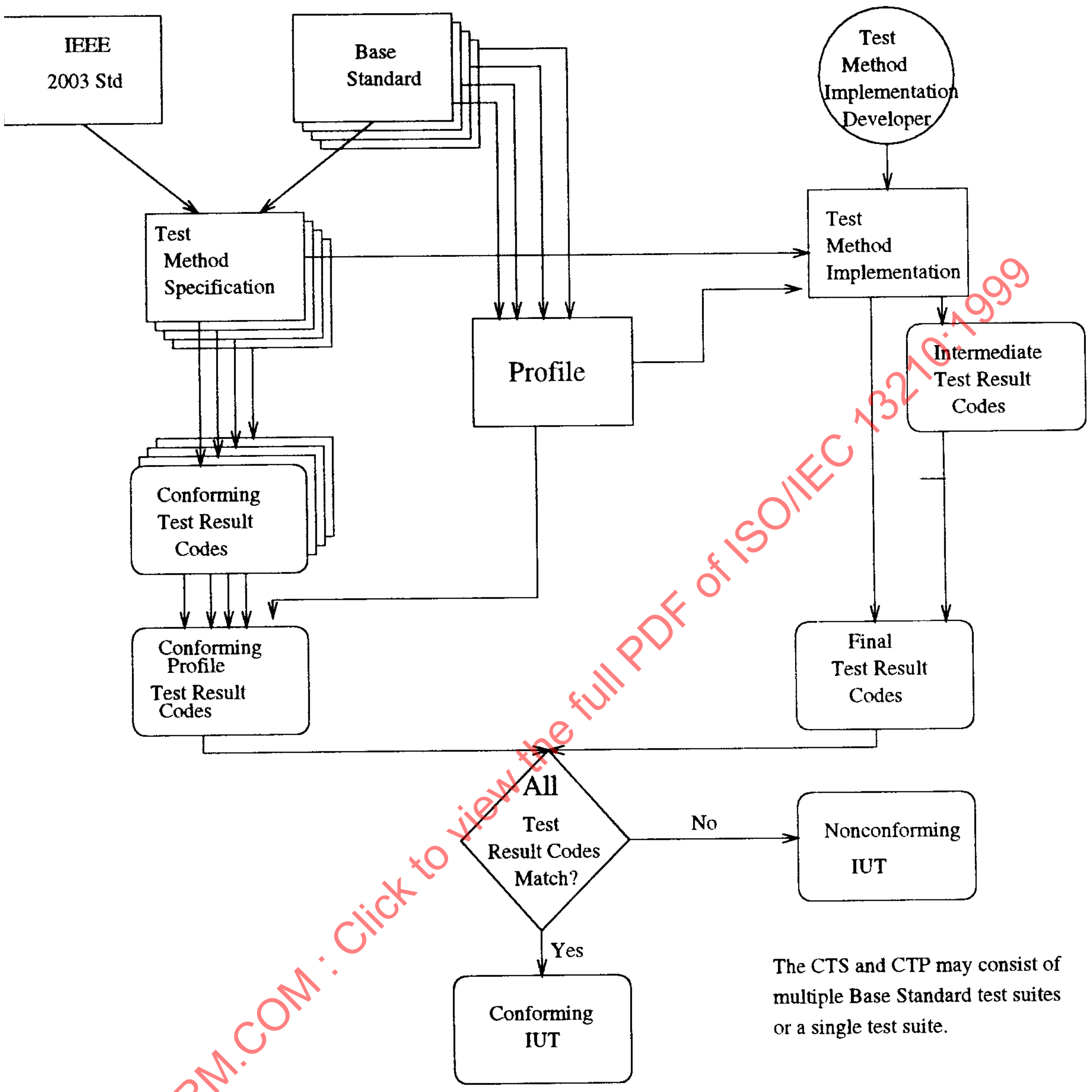
Both models assume the existence of Test Method Standards. The first uses multiple Base Test Methods, while the second uses a Profile Test Method Standard. For either, the concepts remain essentially the same.

It is important to note that the primary purpose of these models is to illustrate the relationship between the test result codes contained in the Test Method Standards and those obtained from the CTS, and CTP, in making conformance assessments.

6.3.1 Using Multiple Base Test Method Standards

The model in Figure 6-1 depicts a Profile composed of several Base Standards with their associated Test Method Standards. A Profile test method implementation consisting of several Base Standard test method implementations and consistent with the existing Test Method Standards is used to conduct conformance testing. The Test Method Standards identify sets of Conforming Test Result Codes, which are then modified as needed per the requirements of the Profile. This results in the identification of a single set of Profile Conforming Test Result Codes. When the Profile test method implementation is executed, any Intermediate Test Result Codes generated are resolved to Final Test Result Codes. If the Profile Conforming Test Result Codes match the Final Test Result Codes, the implementation is judged to be conforming to the Profile.

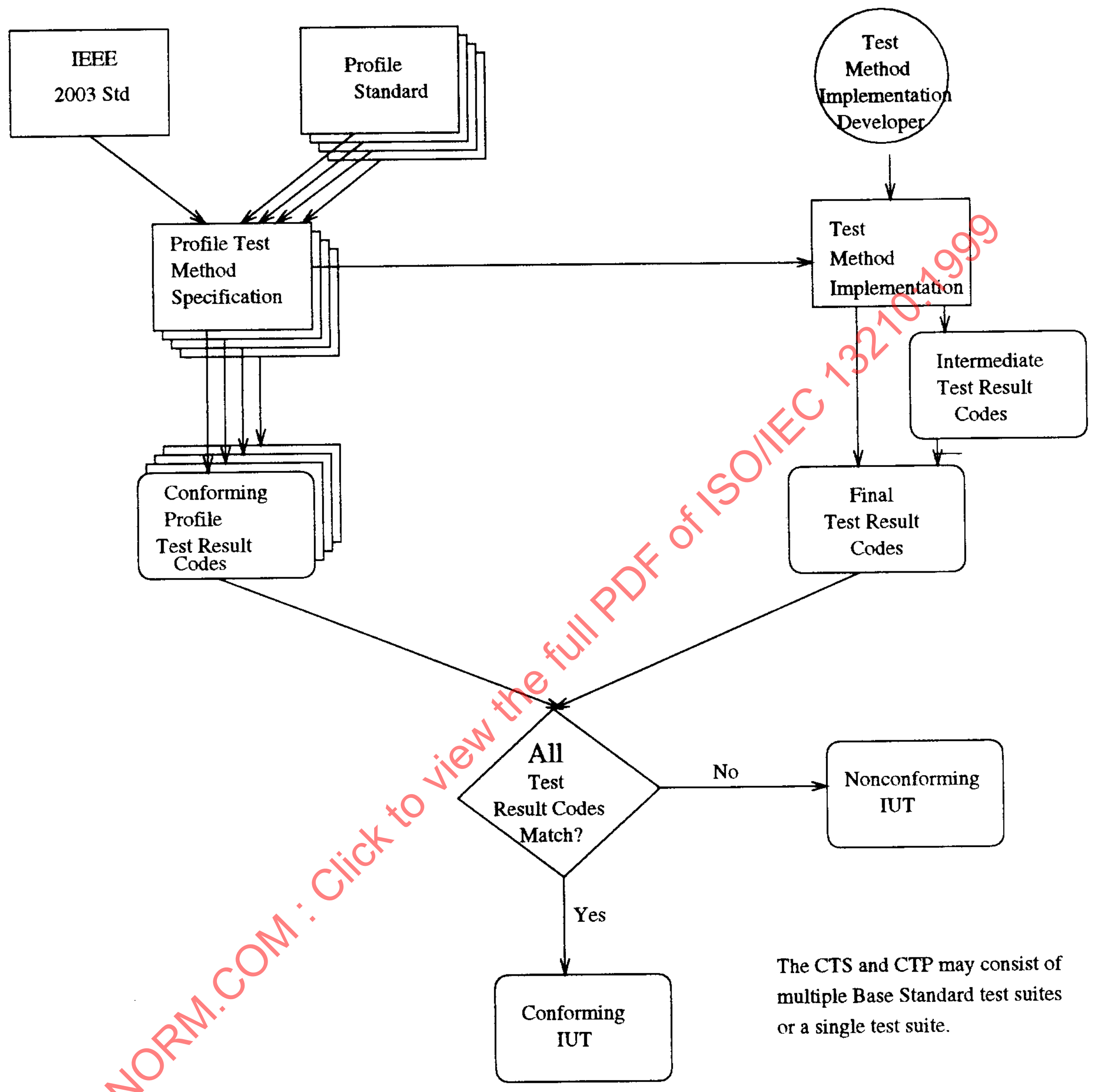
Figure 6-1 — Profile Standard from Multiple Base Standards


IEEE
2003 Std
Base Standard
Test Method Implementation Developer
Test Method Specification
Test Method Implementation
Profile
Intermediate Test Result Codes
Conforming Test Result Codes
Conforming Profile Test Result Codes
Final Test Result Codes
All Test Result Codes Match?
No
Nonconforming IUT
Yes
Conforming IUT
The CTS and CTP may consist of multiple Base Standard test suites or a single test suite.

6.3.2 Using Profile Test Method Standards

The model in Figure 6-2 is similar to the model in Figure 6-1, although in this model multiple Profile Standards (each derived from an associated Base Standard) are depicted with their associated Profile Test Method Standards (each derived from an associated Base Test Method Standard). Profile Conforming Test Result Codes are identified by each Profile Test Method Standard. The Profile CTS in this case is really one or more Base Standard CTSs configured per the requirements of the Profile Test Method Standard (i.e., some *options* may be required). If the Profile Conforming Test Result Codes match the Final Test Result Codes, the implementation is judged to be conforming to the Profile.

Figure 6-2 — Profile Standard from Multiple Profiles


IEEE
2003 Std
Profile Standard
Test Method Implementation Developer
Profile Test Method Specification
Test Method Implementation
Intermediate Test Result Codes
Conforming Profile Test Result Codes
Final Test Result Codes
All Test Result Codes Match?
No
Nonconforming IUT
Yes
Conforming IUT
The CTS and CTP may consist of multiple Base Standard test suites or a single test suite.

(Blank page)

Section 7: Guidelines for Testing and Complexity Levels

This section contains informative material only but is included here as a useful preamble to the material in the next section.

7.1 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 further.

As a result of these limitations, during the design and development of test methods the complexity level of an element shall 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

7.2 Testing Levels

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

Exhaustive testing is normally infeasible.

7.2.2 Thorough Testing

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

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

Thorough testing is a feasible solution. During the testing of a multitude of individual elements, certain combinations of subelements are coincidentally tested.

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

7.2.3 Identification Testing

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

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

7.3 Complexity Levels

7.3.1 Simple

Simple elements are those which are wholly defined in the description for that element. Simple elements are those which have a few assertions to test and whose functionality does not depend on other elements defined within the POSIX standard in which they are defined. Examples of simple elements include the `cat` utility specified in POSIX.2 and the *close*() function specified in POSIX.1. Simple elements should be tested thoroughly.

7.3.2 Intermediate

Intermediate elements are those which have a moderate number of assertions and may depend on the functionality of other elements defined within the POSIX standard in which they are defined. Examples of intermediate elements are the `grep` and `sed` utilities specified in POSIX.2, which support their own functionality in addition to regular expressions. Thorough testing should be a goal for intermediate elements, but it may be infeasible in some cases.

7.3.3 Complex

Complex elements are those which implement a language, depend on the function of intermediate elements, or have effects on hardware devices. Typically, complex elements need a large number of assertion tests to test them thoroughly. Examples of complex elements are the `sh` and `awk` utilities specified in POSIX.2, which implement a language in addition to regular expressions. For complex elements, thorough testing may be limited to specific areas of the element.

7.4 Conclusion

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

It follows from the informal nature of the definition of thorough testing that thoroughness of testing will vary from one CTS to another and, within a CTS, from one assertion to another. These are the options available to the developers of test method specifications and test method implementations. It is not within the scope of this standard to define the recommended testing level so precisely as to preclude this degree of freedom.

(Blank page)

Section 8: Guidelines for Writing Assertions

8.1 Introduction

This section contains informative material only. It presents guidelines for writing assertions for test method specifications. Users of this standard may use other methods to satisfy the requirements of this standard.
There are three fundamental steps in writing test assertions:

— Identify a requirement and its respective assertion types.

— Identify all preconditions for each requirement.

— Write the assertion text.

8.1.1 Identifying Conformance Requirements

Conformance requirements are only those required for conformance. They must be *definitive* in the sense that they must specify or define specific requirements for conforming implementations. Conformance requirements are not vague, and they are not issues that differentiate the quality or performance characteristics of one implementation from those of another.

There is a distinct difference between testing how well something is done and testing whether it was done correctly. The former is a *quality of implementation* issue, and while it is important, it is outside the scope of conformance testing. It is left to system developers to determine whether their systems will perform with a given implementation. Conformance testing is concerned only with the correctness of an implementation's required functionality or behavior. The following steps may be useful in identifying conformance requirements:

— Check the definitions and terminology sections in the standard to identify the terms used to define conformance requirements. Pay particular attention to the definitions that exist for *shall*, *may*, *should*, *can*, *implementation-defined*, *undefined*, and *unspecified*. Among these terms, *shall* is used in POSIX and other International and National Body standards to denote a requirement. Less clear is the use of *may*, *should*, and *can*.

Further, examine closely the material contained in the conformance section of the standard. That section usually addresses overall conformance requirements for the standard.

— Highlight the occurrence of those terms everywhere they are used as well as statements that do not use definitive terminology but mean the same thing.

— Examine the statements containing those terms as candidates for definitive requirements—decide which ones are and which ones are not. Keep in mind that conformance requirements are those required for conformance to the standard. Examine the section in the standard that addresses conformance requirements as well as the rationale for insight. Keep in mind that the rationale is for information only and may even be wrong.

Conformance requirements are found in the baseline requirements of the standard as well as in optional features and behavior. For this reason, careful attention must be given to the use of *may* to distinguish it from an option in an implementation that has *shall* requirements when the *may* option is supported.

As a simple example, a statement to the effect

> The implementation may do A or B.

does not in itself establish any requirement for the implementation. This statement allows either A or B to occur but may not require that either case ever be **TRUE**. The implementation could do one, both, neither, or even something else (C). However, a statement to the effect

> The implementation may do either A or B.
> If A occurs, then A1 shall ...

constitutes a conformance requirement for the implementation only if A occurs.

Another area of potential confusion has to do with the use of the terms *unspecified* and *undefined* At first glance they may seem to mean the same thing; however, they do not. In PASC standards, *unspecified* refers to behavior resulting from *correct program constructs or correct data*, while *undefined* refers to behavior from *erroneous program constructs or erroneous data*. The intent of *unspecified* is to allow standards writers to allow for indeterminate behavior for correct program constructs. Likewise, *undefined* is intended to allow certain error conditions to occur that are not required to generate diagnostic messages. In either case, neither term can result in a conformance requirement for a test assertion but may result in a documentation requirement (see 3.3.5, Documentation Assertions).

It is left to the writers of the standards to define these terms, and it is up to the assertion writers to understand their use. Likewise, statements that imply the use of definitive terminology must be examined for their applicability as a definitive requirement. A conformance requirement by itself or in conjunction with other conformance requirements corresponds to an assertion.

When standards make use of terminology other than *shall* in statements that could be conformance requirements, care must be taken by the assertion writer to determine whether the resulting statement is a requirement of the standard. The use of the word *may* is an example. Certain statements containing the word *may* do not provide sufficient clarity or constitute a conformance requirement. For example, the statement

> A call to *fopen*() may cause the *st_atime* field of the underlying file to be marked for update.

does not declare whether the behavior is to be consistent for all file types or a specific implementation or indicate that it must happen in every instance. As written, the statement would not constitute a conformance requirement of the implementation. While it can be argued that an assertion could be written for the above statement, such an assertion could never evaluate as **FALSE** and therefore provides no information on conformance to the standard. Thus, an assertion that can be evaluated only as **TRUE** or not evaluated at all is meaningless with regard to conformance requirements. (Such an assertion may reflect on the quality of an implementation, but quality of implementation is not a standards conformance issue.) If the above statement has an associated documentation requirement, however, an assertion may be written to validate the Conformance Document.

However, if *may* is used to describe an optional feature allowed by the standard and there are statements that use *shall* in describing behavior characteristics of the optional feature (the implementation may do this ... if it does, when this happens, the implementation shall ...). While it is arguable that including such a construct in a standard is poor practice, it does happen, and the *shalls* used within the *may* constitute conformance requirements.

The definition of *may* as an allowed term for a conformance requirement can be very difficult to deal with in writing assertions and a test suite. Each *may* could indicate allowed branches in the flow of control of a function the definition of which is at the discretion of the system implementor. From the perspective of both the assertion and the assertion test, a reliable method that exercises a *may* feature is difficult to create. The assertion test may be successful in attempting to use the feature and report the results, but it cannot generate a test result code of **FAIL** if the feature acts in an nondeterministic way (i.e., in a way not required by the standard). Further, an allowable side effect of a *may* feature could be catastrophic.

If *shall* is used in a statement applicable to an option, i.e., a feature identified by a *may*, the statement containing the *shall* is a conformance requirement when the optional behavior is supported, or used, by the implementation. The term *should* never identifies a conformance requirement in this standard but is regarded as a recommended practice.

8.1.2 Identifying Requirements and Assertion Types

8.1.2.1 Terms and Assertions

The conformance requirements of base standards often require that the following terms generate specific types of assertions. The base standard must be examined to determine whether this is the case:

— shall: basic assertion

— may: documentation assertion based on "if documented" and one or more basic assertions each containing one or more <Option> attributes

— unspecified: documentation assertion based on "if documented" and a combination of one or more portable or nonportable basic assertions

— undefined: documentation assertion based on "if documented" and no basic assertions

— implementation-defined: documentation assertion and a combination of one or more portable or nonportable basic assertions

— documented: documentation assertion

— The conformance requirements of base standards often require that the following terms generate no assertions. The base standard must be examined to determine if this is the case:

  — should

  — undefined

  — extensions

  — statements that apply only to the usage of an implementation rather than to an implementation itself, generating no assertions

  — warning to programmers

  — implementation recommendations

Other terms used in base standards relate directly to the terms above. Some of these terms are specified in Figure 8-1.

Figure 8-1 – Terms

Term/Phrase	Defined Term
can	may
may vary	may
is	shall
must	shall
will	shall
document	documented
implementation warning	should

8.1.2.2 Highlighting Guidelines

The following describes a procedure that has been used successfully to determine definitive requirements within a base standard.

8.1.2.2.1 Basic Assertion

Get a highlighter.

Go through the standard and highlight all the statements with keywords defined by the base standard to specify conforming behavior. In POSIX standards, these normally will be *shall*, *will*, *may*, *must*, and *or*. Also, highlight statements that imply the use of these words. For example, the statement

> A call to *fopen*() causes the *st_atime* field of the underlying file to be marked for update.

is an implied *shall*. Determine which of these highlighted statements apply only to applications and cross them out. Assertions are written only for implementation requirements.

8.1.2.2.2 Optional Assertions Text

Get a highlighter of a different color.

Go through the standard and highlight the terms and phrases associated with generating assertions based on <Option>.

8.1.2.2.3 Documentation Assertions Text

This step is required only if conformance to the base standard requires that certain features, such as those whose behavior is implementation-defined, or that support options, be documented. POSIX standards generally require such documentation in a conformance document, usually referred to as a PASC (or POSIX) Conformance Document. When a base standard has such a requirement, documentation assertions must be generated.

Get a pen or pencil.

Go through the standard and underline those terms and phrases associated with generating documentation assertions.

8.1.2.3 Establish Symbols

This approach provides a naming convention for base standard features that is needed to improve the readability and accuracy of test method specifications and to allow test method implementations to refer to base standard features with terms that are consistent with those used in the test method specification.

To produce assertions that are easily understood, a systematic approach should be used to ensure that the text of assertions is consistent. Determine all the preconditions specified by the base standard and the associated base standards. If it has not already been done by the base standard, assign a symbol to each of these preconditions. When a standard is unclear on the text of a precondition, the test method writer will have to come up with text to adequately define the symbol to associate with the precondition.

— When assertions are dependent on the PCD for an announcement mechanism about the support or nonsupport of an <Option>, the symbol to be used and its associated text must be determined. Symbols associated with these requirements should be named, starting with "PCD_".

— When assertions are dependent on base standard limits that need not be attainable for all conforming implementations, then a symbol and a testing limit should be established for these assertions to use. Symbols associated with these requirements should be named, starting with "CTS_".

— When assertions are dependent on <Test_Support>, then a symbol should be established. Symbols associated with these requirements should be named, starting with "TS_".

— All symbols associated with PCD_, CTS_, and TS_ designations should be collected in one or more tables that associate a symbol's name with its usage.

8.1.2.4 Generating Assertions

Assertions are required for each definitive statement in a base standard. A single assertion may include more than one definitive requirement. Since standards generally are written in prose style, it usually does not suffice to copy the text directly from the base standard as the assertion text. Each assertion shall exist as a stand-alone statement and be worded so that requirements are stated for conforming implementations.

Assertions that require support for implementation-defined or unspecified behaviors are nonportable. The ability to portably test the assertion should be a major factor in determining the text associated with an assertion. Base standard specifications that cannot be portably tested as a whole should be addressed, when feasible, by assertions that test the portable testable features as well as assertions that state the nonportable testable features.

8.1.3 Basic Assertions via Examples

Many examples are provided of the POSIX.1 text and its related assertions as specified by IEEE Std 2003.1-1992 {4} that have been altered to adhere to the assertion constructs of this standard. These examples appear in the following subclauses of this section.

8.1.4 Conclusion

Writing assertions is a learning process. Two working group members writing assertions for the same base standard text probably will not write identical assertions, though both sets of assertions may be correct.

8.2 Identifying Preconditions

When the ′If <precondition>′ construct evaluates to **TRUE**, the next sequential construct is processed. When the ′If <precondition>′ construct evaluates to **FALSE**, the next construct processed is the ′Else′ associated with the **FALSE** ′If.′

When a precondition for an assertion is not needed, neither the precondition syntax nor its corresponding ′Else′ is stated.

In traversing an assertion construct, if support for a precondition is not provided, the test result code is obtained from the text that represents the corresponding ′Else′ symbolic.

8.2.1 <Applicable_Standard>

The first precondition is the Applicable_Standard from which the assertion is derived. For example, if the assertion is for an element specified in POSIX.1-1990, the corresponding precondition would be

```
If POSIX.1-1990 then
  ...
Else <No_Applicable_Standard>
```

The <Applicable_Standard> for all assertions in a base standard shall be the same as that standard. If <Applicable_Standard> evaluates as **TRUE**, the assertion logic shall continue to the next precondition. If <Applicable_Standard> evaluates as **FALSE**, the assertion logic shall terminate and return <No_Applicable_Standard>.

The purpose of this precondition is to enable a developer of profile assertions to establish the base standard from which the element is defined.

For text specifications, see 3.2.2.

8.2.2 Option

An <Option> refers to an optional element or feature of an element defined in the Applicable_Standard. If the assertion is dependent on the existence of an optional element or feature defined in the Applicable_Standard, each optional feature shall be identified as a Option, creating a nested IF structure.

For example, in POSIX.1-1990, one option allowed is support for Standard C.

> POSIX.1, 2.7.3: Implementations claiming C Standard Language-Dependent Support shall declare function prototypes for all functions.

If Standard C is supported, all functions must be declared as function prototypes. An assertion to test whether this is **TRUE** for a given function would start as

```
If POSIX.1-1990 then
    If Standard C then
        ...
    Else <No_Option>
Else <No_Applicable_Standard>
```

Another example from POSIX.1, 1.3.4, has to do with whether a function is implemented as a macro:

> POSIX.1, 1.3.4 (3): Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments only once...

An assertion testing such behavior for the function write() would start out as

```
If POSIX.1-1990 then
    If write() is defined as a macro when the header <unistd.h> is
    included, then
        ...
    Else <No_Option>
Else <No_Applicable_Standard>
```

An <Option> shall evaluate as **TRUE** in order for the test assertion logic to proceed to the next step. If no <Option> is required, it shall be designated as **NONE** and evaluated as **TRUE**. If <Option> evaluates as **FALSE**, the assertion logic shall terminate with a return of <No_Option>.

For text specifications, see 3.2.4.

Example: ISO/IEC 9945-1:1990, 3.1.1.2, lines 13,33-34

Base Standard Text:

The fork function creates a new process. ... All other process characteristics defined by this part of ISO/IEC 9945 shall be the same in the parent and the child processes.

Test Method Assertion:

23

If the behavior associated with {_POSIX_JOB_CONTROL} is supported, then

Test: When a call to *fork*() completes successfully, then the child process is in the same session as its parent.

Else No_Option

Example: ISO/IEC 9945-1:1990, 3.1.1.4, lines 51-54

Base Standard Text:

For each of the following conditions, if the condition is detected, the *fork*() function shall return -1 and set *errno* to the corresponding value:

[ENOMEM] The process requires more space than the system is able to supply.

Test Method Assertion:

28

If the implementation supports the detection of [ENOMEM] for *fork*(), then

Test: When the process requires more space than the system is able to supply, then a call to *fork*() returns a value of (*pid_t*)-1, sets *errno* to [ENOMEM], and no process is created.

Else No_Option

29

If the implementation does not support the detection of [ENOMEM] for *fork*(), then

Test: When the process requires more space than the system is able to supply, then a call to *fork*() is successful (unless a different error condition is detected).
See GA26 in 2.4.

Else No_Option

8.2.3 Test Support

The nature of standards development is such that it is possible to allow for the existence of features in an implementation that are not explicitly defined. For example, POSIX.1 allows for the existence of a read-only file system but does not define the mechanism for creating one.

Situations such as this lead to the development of test assertions that cannot be transformed into assertion tests without the support of functionality that is neither defined nor required by the base standard. This needed test support functionality is known as <Test_Support>. A <Test_Support> precondition may be hardware or software or both, required to be supported by the implementation under test.

For text specifications, see 3.2.4.

Example: ISO/IEC 9945-1:1990, 3.1.1.2, lines 13,33-34

Base Standard Text:

The fork function creates a new process. ... All other process characteristics defined by this part of ISO/IEC 9945 shall be the same in the parent and the child processes.

Test Method Assertion:

20

If PCTS_GTI_DEVICE:

Test: When a call to *fork*() completes successfully, then the controlling terminal for the child process is the same as for the parent.

Else No_Test_Support

8.2.4 Setup

Setup addresses the state of the test environment required to perform the `'Test:'` construct of the assertion. Its text should include all the preconditions specified by the base standard and may include additional manual or automated procedures.

Some guidelines for 'Setup' text are as follows:

— If <Test_Text> begins a clause with the word *when* or *where*, this clause probably should be moved to <Setup_Requirements>.

— If <Test_Text> specifies constraints on files or features needed to test, these constraints probably should be moved to <Setup_Requirements>.

— If special needs must be addressed that are not stated in the base standard, such as opening a terminal file with CLOCAL clear, this need should be stated in <Setup_Requirements>.

Example: ISO/IEC 9945-1:1990, 6.4.2.2, lines 201-202

Base Standard Text:

If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write. ...

Test Method Assertion:

11

Setup: When the O_APPEND flag of the file status flags is set

Test: then a call to *write*() sets the file offset to the end of the file prior to each *write*().

TR: Test for files opened in O_WRONLY and O_RDWR modes.

8.3 Writing the <Test_Text>

<Test_Text> is a statement of behavior of functionality. It should be written in a manner that allows correct behavior or correct functionality for conforming implementations.

12

Test: Before a successful return from a call to write(), the file offset is incremented by the number of bytes actually written.

8.3.1 Using Tables

Example: ISO/IEC 9945-1:1990, 2.8.1, lines 959-963

Base Standard Text:

The following limits used in this part of ISO/IEC 9945 are defined in the C Standard {2}: {CHAR_BIT}, {CHAR_MAX}, {CHAR_MIN}, {INT_MAX}, {INT_MIN}, {LONG_MAX}, {LONG_MIN}, {MB_LEN_MAX}, {SCHAR_MAX}, {SCHAR_MIN}, {SHRT_MAX}, {SHRT_MIN}, {UCHAR_MAX}, {UINT_MAX}, {ULONG_MAX}, {USHRT_MAX}.

Test Method Assertion:

01

Setup: Include the header `<limits.h>` in the test module.

Test: The following symbols and corresponding values are defined and have values that meet the requirements of the C Standard {3}, as shown in Table 8-1.

Table 8-1 – C Language Limits

Symbol	Value	Description
CHAR_BIT	8	Minimum number of bits
CHAR_MAX		See note below
CHAR_MIN		See note below
INT_MAX	+32767	Minimum maximum
INT_MIN	-32767	Maximum minimum
LONG_MAX	+2147483647	Minimum maximum
LONG_MIN	-2147483647	Maximum minimum
MB_LEN_MAX	1	Minimum maximum
SCHAR_MAX	+127	Minimum maximum
SCHAR_MIN	-127	Maximum minimum
SHRT_MAX	+32767	Minimum maximum
SHRT_MIN	-32767	Maximum minimum
UCHAR_MAX	255	Minimum maximum
UINT_MAX	65535	Minimum maximum
ULONG_MAX	4294967295	Minimum maximum
USHRT_MAX	65535	Minimum maximum

8.3.2 Using Testing Requirements

Example: ISO/IEC 9945-1:1990, 3.3.1.4, lines 575-578

Base Standard Text:

If the signal-catching function executes a *return*, the behavior of the interrupted function shall be described individually for that function. Signals that are ignored shall not affect the behavior of any function; signals that are blocked shall not affect the behavior of any function until they are delivered.

Test Method Assertion:

60

Test: When any signal that is being blocked is generated while any function is being executed, then the signal does not have any effect on the behavior of the function.

TR: Test for the functions *fcntl*(), *open*(), *pause*(), *read*(), *sleep*(), *sigsuspend*(), *wait*(), *waitpid*(), and *write*(), each of which can be placed in a state where it would report being interrupted by signal if it were delivered. The action of SIGALRM on *sleep*() is unspecified.

8.3.3 Notes

When additional factors should be known to the CTS writer for generating a portable CTS, then a NOTE is added.

Example: ISO/IEC 9945-1:1990, 2.6, lines 717-727

Base Standard Text:

LOGNAME	The login name associated with the current process. The value shall be composed of characters from the portable filename set.
	Note: An application that requires, or an installation that actually uses, characters outside the portable filename character set would not strictly conform to this part of ISO/IEC/9945. However, it is reasonable to expect that such characters would be used in many countries (recognizing the reduced level of interchange implied by this), and applications or installations should permit such usage where possible. No error is defined by this part of ISO/IEC 9945 for violation of this condition.

Test Method Assertion:

02

If the environment variable **LOGNAME** was defined by the implementation and currently has the value defined by the implementation, then

Test:	The environment variable **LOGNAME** corresponds to the *login name* associated with the current process.
	Note: Testing that **LOGNAME** is composed of *characters from the portable filename character set* is not asserted since POSIX.1 implies in a note that this condition should be tolerated.
Else	No_Option

8.4 Other Assertion Types

8.4.1 General Assertions

General assertions are utilized when common functionality among elements is specified in the base standard in a single generalized statement. The general statement is broken out into specific assertions in the appropriate areas of the test methods standard. General assertions also allow a single statement to govern the language of successive assertions that result in a consistent terminology for the test method standard.

An orderly process can be used in developing general assertions:

(1) Determine commonality criteria between elements or classes of elements. This can be done by examining introductory material in the standard, and in each section of the standard, as well as the details of each element.

(2) Group common elements on the base of the initial commonality criteria established. This is likely to be an iterative process where the commonality criteria are adjusted as needed.

(3) Determine specific characteristics within each element.

(4) Create a generic phrase which describes those characteristics.

(5) Verify that the general assertion accurately represents each common element as applicable.

Once a general assertion is developed, it must be expanded to a specific assertion for each applicable element, which then refers to the general assertion from which it was derived. The general assertion is not tested per se. It is the derived assertion that is tested.

Example: ISO/IEC 9945-1:1990, 2.7.3, lines 895-898

Base Standard Text:

Implementations claiming C Standard {2} Language-Dependent Support shall declare the function prototypes for all functions.

Implementations claiming Common-Usage C Language-Dependent Support shall declare the result type for all functions not returning a "plain" *int*.

Test Method Assertion:

For all elements with result type not *void* and not *int*:
GA_STDC_func_prot_01

If ISO/IEC 9899:1990 Programming Language — C:

Setup: Include the header `<*.h>`.

Test: The function prototype *type1 funct*(*type2*, *type3*) is declared.[5)]

Else No_Applicable_Standard

If Programming Language Common-Usage C:

Setup: Include the header `<*.h>`.

Test: The function *funct*() is declared with the result type *type1*.

Else No_Applicable_Standard

For all elements with result type *void* except *assert*():

If ISO/IEC 9899:1990 Programming Language — C:

Setup: Include the header `<*.h>`.

Test: The function prototype *void funct*(*type2*, *type3*) is declared.

Else No_Applicable_Standard

For all elements with result type *int* except *setjmp*() and *sigsetjmp*():

If ISO/IEC 9899:1990 Programming Language — C:

Setup: Include the header `<*.h>`.

Test: The function prototype *int funct*(*type2*, *type3*) is declared.

Else No_Applicable_Standard

If Programming Language Common-Usage C:

Setup: Include the header `<*.h>`.

Test: The function *funct*() either is declared with the result type *int* or is not

5) The function prototypes are aligned with ISO/IEC 9945-1: 1990. When not specified in ISO/IEC 9945-1: 1990, they are aligned with ISO/IEC 9899:1990.

declared in the header.

Else No_Applicable_Standard

For *funct*() of *setjmp*() and *sigsetjmp*():

If *funct*() is not defined as a macro, then

Setup: Include the header `<setjmp.h>`.

Test: Function *funct*() is declared as an identifier with external linkage and result type *int*.

Else No_Option

8.4.2 Reference Assertions

A reference assertion is one that refers to another existing assertion within the element being tested. Reference assertions are used to avoid having to write the same assertion more than once within a single element.

Example: ISO/IEC 9945-1:1990, 6.4.2.2, lines 214-216, and 6.4.2.4, lines 277-279

Base Standard Text:

If *write*() is interrupted by a signal after it successfully writes some data, either it shall return -1 with *errno* set to [EINTR], or it shall return the number of bytes written.

[EINTR] The write operation was interrupted by a signal, and either no data was transferred or the implementation does not report partial transfers for this file.

Test Method Assertion:

R_01

Setup: Interrupt *write*() by a signal before any data is written.

Test: The call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

See: [Assertion(s) 39 in 6.4.2.4.]

39

Setup: Terminate *write*() by receipt of a signal before any data was transferred.

Test: The call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

8.4.3 Documentation Assertions

Documentation assertions ensure that the PCD is complete, thereby addressing the needs of portable application writers. The PCD ensures that the features associated with the variability of PASC base standards are identified for the IUT.

POSIX standards specify when conformance documentation text is required and when it may be required. Some POSIX phrases that spawn documentation assertions and their associated conforming test result codes are stated in Table 8-2.

Table 8-2 – Phrases Denoting Allowable Test Result Codes

Phrase	Allowable Test Result Codes	
implementation defined	Documented	
may vary	Documented	No_Documentation
unspecified	Documented	No_Documentation
undefined	Documented	No_Documentation
shall document	Documented	

Text similar to these phrases - such as "shall be documented" - which are similar to "shall document" also spawn documentation assertions.

When a base standard specifies that a feature or a behavior shall be documented, a documentation assertion is required. For these assertions, vendors need not provide documentation in the conformance document and the allowable test result code are stated in Table 8-3.

Table 8-3 – Features Denoting Allowable Test Result Codes

Features	Allowable Test Result Codes	
features	Documented	No_Documentation
behaviors	Documented	No_Documentation

Example: ISO/IEC 9945-1:1990, 6.4.2.2, lines 214-216

Base Standard Text:

If *write*() is interrupted by a signal after it successfully writes some data, either it shall return -1 with *errno* set to [EINTR], or it shall return the number of bytes written.

Test Method Assertion:

D_03

If the conditions under which *write*(), when interrupted by a signal after it has successfully written some data, returns -1 and sets *errno* to [EINTR] or returns the number of bytes read are documented:

Test: The details are contained in 6.4.2.2 of the PCD.

Else No_Option

8.5 Macros

The following is an example of how a macro might be used to express an assertion derived from a base standard. Note that the “macro language” is not defined here and that this serves only as an example of how a macro might appear. Users are required per 3.4 to specify, or define, any macro convention used.

Example: ISO/IEC 9945-1:1990, 6.4.2.2, lines 214-216, and 6.4.2.4, lines 277-279

Base Standard Text:

If *write*() is interrupted by a signal after it successfully writes some data, either it shall return -1 with *errno* set to [EINTR], or it shall return the number of bytes written.

[EINTR] The write operation was interrupted by a signal, and either no data was transferred or the implementation does not report partial transfers for this file.

Test Method Assertion:

```
#define M_write_interrupted(XXX)
  If {PCD_WRITE_INTERRUPTED} is XXX, then
    If PCTS_GTI_DEVICE, then
     Setup:   Interrupt with a signal a write() to a terminal device file after
              successfully writing some data.
     Test:    The call to write()

#define M_NO_OPTION__NO_TEST_SUPPORT
          Else No_Implicit_Option
     Else No_Test_Support

#define M_NO_OPTION
     Else No_Implicit_Option
```

12
M_write_interrupted(TRUE) returns the number of bytes written.
M_NO_OPTION__NO_TEST_SUPPORT
13
M_write_interrupted(FALSE) returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].
M_NO_OPTION__NO_TEST_SUPPORT
14
M_write_interrupted("not documented") either returns the number of bytes written or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].
M_NO_OPTION__NO_TEST_SUPPORT
15
M_write_interrupted("TRUE && the implementation supports character special files") returns the number of bytes written.
M_NO_OPTION
16
M_write_interrupted("FALSE && the implementation supports character special files") returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].
M_NO_OPTION
17
M_write_interrupted("not documented && the implementation supports character special files") either returns the number of bytes written or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].
M_NO_OPTION

(Blank page)

Section 9: Comprehensive Examples

9.1 Specification of Allowable Test Result Codes

The conforming test result codes are provided by the test method specification. Figure 9-1 provides an example of how the conforming test result codes may be presented for the examples provided in 9.1.2.

9.1.1 Example: C Binding Allowable Test Result Codes for write()

Figure 9-1 – Entity versus Allowable Test Result Code

Element	Assertion #	Allowable Test Result Codes		
write	01	PASS[1,2]	NO_APPLICABLE_STANDARD	
write	02	PASS[1,2]	NO_APPLICABLE_STANDARD	
write	03	PASS	NO_OPTION	
write	04	PASS	NO_OPTION	
write	05	PASS		
write	06	PASS		
write	D01	PASS	NO_OPTION	
write	07	PASS		
write	08	PASS		
write	09	PASS		
write	10	PASS		
write	D02	PASS	NO_OPTION	
write	11	PASS	NO_TEST,4	
write	12	PASS	NO_TEST_SUPPORT	NO_OPTION
write	D03	PASS	NO_OPTION	
write	13	PASS[13-15]	NO_OPTION	NO_TEST_SUPPORT
write	14	PASS[13-15]	NO_OPTION	NO_TEST_SUPPORT
write	15	PASS[13-15]	NO_OPTION	NO_TEST_SUPPORT
write	16	PASS	NO_OPTION	NO_TEST,1
write	17	PASS	NO_OPTION	NO_TEST,1
write	18	PASS	NO_OPTION	NO_TEST,1
write	D04	PASS		
write	19	PASS		
write	20	PASS		
write	21	PASS	NO_TEST,2	
write	22	PASS		
write	23	PASS	NO_TEST,3	
write	24	PASS	NO_OPTION	NO_TEST,1
write	25	PASS	NO_OPTION	NO_TEST,1
write	26	PASS		
write	27	PASS		
write	28	PASS	NO_TEST_SUPPORT	
write	29	PASS	NO_TEST_SUPPORT	
write	30	PASS	NO_TEST_SUPPORT	
write	31	PASS	NO_OPTION	NO_TEST,1
write	32	PASS	NO_OPTION	NO_TEST,1
write	33	PASS	NO_OPTION	NO_TEST,1
write	34	PASS	NO_OPTION	NO_TEST,1
write	35	PASS		

9.1.2 Example: C Binding Assertions for write()

The following assertions are those of *write*() from IEEE Std 2003.1-1992 {4} and have been reformatted to conform to this standard.

01
If ISO/IEC 9899:1990 Programming Language — C:

Setup: Include the header `<unistd.h>`.

Test: The function prototype `ssize_t write(int, const void *, size_t)` is declared.
See GA36 in 2.7.3.

Else No_Applicable_Standard

02

If Programming Language Common-Usage C:

Setup: Include the header `<unistd.h>`.

Test: The function *write*() is declared with the result type *ssize_t*.
See GA36 in 2.7.3.

Else No_Applicable_Standard

03

If *write*() is defined as a macro when the header `<unistd.h>` is included:

Setup: Invoke the macro *write*() with the correct argument types (or compatible argument types in the case that C Standard {3} support is provided).

Test: The macro *write*() expands to an expression with the result type *ssize_t*.
See GA37 in 2.7.3.

Else No_Option

04

If *write*() is defined as a macro when the header `<unistd.h>` is included:

Setup: Invoke the macro *write*() with the correct argument types (or compatible argument types in the case that C Standard {3} support is provided).

Test: The arguments for *write*() are evaluated only once and are fully protected by parentheses when necessary, and the result value is protected with extra parentheses when necessary.
See GA01 in 1.3.4.

Else No_Option

05

Test: A call to *write*(*fildes, buf, nbyte*) writes *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor argument *fildes* and returns the number of bytes written.

06

Setup: File's type is a regular file.

Test: A call to *write*(*fildes, buf, 0*) returns zero and does not mark for update the *st_ctime* and *st_mtime* fields of the file, and no bytes are written.

TR: Test for both O_APPEND flag clear and O_APPEND flag set.

D_01

If the results of a call to *write*() when *nbyte* is zero and the file is not a regular file is documented:

Test: The details are contained in 6.4.2.2 of the CD.

Else No_Option

07

Setup: Use a regular file type or other file type capable of seeking.

Test: Call to *write*(*fildes, buf, name*) starts writing at a position in the file given by the file offset associated with the file descriptor argument *fildes*.

08

Test: Before a successful return from a call to *write*(), the file offset is incremented by the number of bytes actually written.

09

Setup: The current file offset, for a regular file type, after a successful return from a call to *write*() is greater than the length of the file before the call.

Test: The length of the file after the call is set to this file offset.

10

Setup: Open a file not capable of seeking.

Test: Call to *write*() starts at the current position in the file.

D_02

If the value of the file offset after a call to *write*() for a file type that is not capable of seeking is documented:

Test: The details are contained in 6.4.2.2 of the CD.

Else No_Option

11

Setup: When the O_APPEND flag of the file status flags is set.

Test: then a call to *write*() sets the file offset to the end of the file prior to each *write*().

Test: Test for files opened in O_WRONLY and O_RDWR modes.

12

Setup: The file's type is a regular file, and a *write*() requests that more bytes be written than there is room for.

Test: Call to *write*() writes only as many bytes as there is room for.

R_01

Setup: Interrupt *write*() by a signal before any data is written.

Test: The call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].

See: [Assertion(s) 39 in 6.4.2.4.]

D_03

If the conditions under which *write*(), when interrupted by a signal after it has successfully written some data, returns -1 and sets *errno* to [EINTR] or returns the number of bytes read are documented:

Test: The details are contained in 6.4.2.2 of the CD.

Else No_Option

13

If {PCD_WRITE_INTERRUPTED} is **TRUE**:

If PCTS_GTI_DEVICE:

Setup: Interrupt *write*() to a terminal device file by a signal after successfully writing some data.

Test: The call to *write*() returns the number of bytes written.
Else No_Test_Support
Else No_Option

14
If {PCD_WRITE_INTERRUPTED} is **FALSE**:
If PCTS_GTI_DEVICE:
Setup: Interrupt *write*() to a terminal device file by a signal after successfully writing some data.
Test: The call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].
Else No_Test_Support
Else No_Option

15
If {PCD_WRITE_INTERRUPTED} is not documented:
If PCTS_GTI_DEVICE:
Setup: Interrupt *write*() to a terminal device file by a signal after successfully writing some data.
Test: The call to *write*() either returns the number of bytes written or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].
Else No_Test_Support
Else No_Option

16
If {PCD_WRITE_INTERRUPTED} is **TRUE** and implementation supports character special file:
Setup: When *write*() to a character special file is interrupted by a signal after successfully writing some data.
Test: The call to *write*() returns the number of bytes written.
Else No_Option

17
If {PCD_WRITE_INTERRUPTED} is **FALSE** and implementation supports character special file:
Setup: Interrupt *write*() to a character special file by a signal after successfully writing some data.
Test: The call to *write*() returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].
Else No_Option

18
If {PCD_WRITE_INTERRUPTED} is not documented and implementation supports character special file:
Setup: Interrupt *write*() to a character special file by a signal after successfully writing some data.
Test: The call to *write*() either returns the number of bytes written or returns a value of (*ssize_t*)-1 and sets *errno* to [EINTR].
Else No_Option

D_04

Test: When the value of *nbyte* is greater than {SSIZE_MAX}, the details describing the results of a call to *write*() are contained in 6.4.2.2 of the CD.

19

Setup: A *write*() to a regular file has returned successfully.

Test: A successful *read*() from any byte position that was modified by the previous *write*() returns the data written to that position by that previous *write*() until such byte positions are again modified.

20

Setup: A regular file already contains data at the position referenced by a successful call to *write*().

Test: The data at the position referenced are overwritten.

21

Setup: The file's type is a pipe or a FIFO and *write*() has transferred some data and *nbyte* is less than or equal to {PIPE_BUF}.

Test: Call to *write*(*fildes, buf, nbyte*) does not return with *errno* set to [EINTR].

TR: Test for both a pipe and a FIFO.

22

Setup: File's type is a pipe or a FIFO.

Test: Call to *write*() appends to the end of the pipe or FIFO.

TR: Test for both a pipe and a FIFO.

23

Setup: File's type is a pipe or a FIFO and *nbyte* is less than or equal to {PIPE_BUF}.

Test: Call to *write*() does not interleave with data from other processes doing writes on the same pipe or FIFO.

TR: Test for both a pipe and a FIFO.

R_02

Setup: File's type is a pipe or a FIFO, the O_NONBLOCK flag is set, and no data can be accepted at the time of the *write*().

Test: Call to *write*() does not block the process.

See: [Assertion(s) 28 in 6.4.2.2.]

24

If the implementation supports character special files with nonblocking I/O:

Setup: File's type is a character special file, the O_NONBLOCK flag is set, and no data can be accepted at the time of the *write*().

Test: Call to *write*() does not block the process.

Else No_Option

Note: The case of a terminal device file is covered by Assertion 43 in 7.1.1.8.

25

If the implementation supports block special files with nonblocking I/O:

Setup: File's type is a block special file, the O_NONBLOCK flag is set, and no data can be accepted at the time of the *write*().

Test: Call to *write*() does not block the process.

Else No_Option

26

Setup: File's type is a pipe or a FIFO, the O_NONBLOCK flag is set, *nbyte* is less than or equal to {PIPE_BUF}, and space exists in the pipe or FIFO for *nbyte* bytes of data.

Test: Call to *write*(*fildes*, *buf*, *nbyte*) succeeds completely and returns *nbyte*.

TR: Test for both a pipe and a FIFO.

TR: When {PIPE_BUF} > {PCTS_PIPE_BUF}, test with values of *nbyte* up to and including {PCTS_PIPE_BUF}.

27

Setup: When the file's type is a pipe or a FIFO, the O_NONBLOCK flag is clear, and *nbyte* is less than or equal to {PIPE_BUF}.

Test: Call to *write*(*fildes*, *buf*, *nbyte*) blocks until space is available to complete the *write*() or the *write*() is interrupted by a signal.

TR: When {PIPE_BUF} > {PCTS_PIPE_BUF}, test with values of *nbyte* up to and including {PCTS_PIPE_BUF}.

28

If {PIPE_BUF} ≤ {PCTS_PIPE_BUF}:

Setup: File's type is a pipe or a FIFO, the O_NONBLOCK flag is set, *nbyte* is greater than {PIPE_BUF} bytes, and at least one byte can be written.

Test: Call to *write*() transfers what it can and returns the number of bytes written.

TR: Test for both a pipe and a FIFO.

Else No_Test_Support

29

If {PIPE_BUF} ≤ {PCTS_PIPE_BUF}:

Setup: File's type is a pipe or a FIFO, the O_NONBLOCK flag is set, *nbyte* is greater than {PIPE_BUF} bytes, and no data can be written.

Test: Call to *write*() returns a value of [(*ssize_t*)-1], sets *errno* to [EAGAIN], and transfers no data.

TR: Test for both a pipe and a FIFO.

Else No_Test_Support

30

If {PIPE_BUF} ≤ {PCTS_PIPE_BUF}:

Setup: File's type is a pipe or a FIFO, the O_NONBLOCK flag is set, and *nbyte*> {PIPE_BUF} bytes and all data previously written to the pipe or FIFO has been read.

Test: Call to *write*(*fildes*, *buf*, *nbyte*) transfers at least {PIPE_BUF} bytes.

TR: Test for both a pipe and a FIFO.
Else No_Test_Support

31
If the implementation supports character special files with nonblocking I/O:
Setup: File's type is a character special file, the O_NONBLOCK flag is clear, and the file cannot accept the data immediately.
Test: Call to *write*() blocks until the data can be accepted.
Else No_Option
Note: The case of a terminal device file is covered by Assertion 43 in 7.1.1.8.

32
If the implementation supports block special files with nonblocking I/O:
Setup: When the file's type is a block special file, the O_NONBLOCK flag is clear, and the file cannot accept the data immediately.
Test: Call to *write*() blocks until the data can be accepted.
Else No_Option

33
If the implementation supports character special files with nonblocking I/O:
Setup: File's type is a character special file, the O_NONBLOCK flag is set, and some data can be written without blocking the process.
Test: Call to *write*() either writes what it can and returns the number of bytes written, or returns a value of (*ssize_t*)-1, sets *errno* to [EAGAIN], and transfers no data.
Else No_Option
Note: The case of a terminal device file is covered by Assertion 44 in 7.1.1.8.

34
If the implementation supports block special files with nonblocking I/O:
Setup: File's type is a block special file, the O_NONBLOCK flag is set, and some data can be written without blocking the process.
Test: Call to *write*() either writes what it can and returns the number of bytes written, or returns a value of (*ssize_t*)-1, sets *errno* to [EAGAIN], and transfers no data.
Else No_Option

35
Setup: Successful call to *write*() of more than zero bytes.
Test: Marks for update the *st_ctime* and *st_mtime* fields of the file.

R_03
Test: When a call to *write*() completes successfully, then the number of bytes written is returned.
See: [Assertion(s) 4 in 6.4.2.2.]

R_04
Test: When a call to *write*() completes unsuccessfully, then a value of

(*ssize_t*)-1 is returned and *errno* is set to indicate the error.

See: [Assertion(s) 35-42 in 6.4.2.4.]

9.1.3 Example: Ada Binding Assertions for write()

The following assertions are those of *write*() from IEEE Std 1003.5-1992, and have been rewritten to conform to this standard.

01

Setup: The package `POSIX_IO` is used.

Test: The procedure `Write` is defined as follows:

```
procedure Write
    ( File : in File Descriptor;
      Buffer : in IO_Buffer;
      Last : out POSIX.IO_Count;
      Masked_Signals: in POSIX.Signal_Masking := POSIX.RTS_Signals);
```

02

Test: A call to *Write*(*file, buff, last, masked_signals*) will write the entire *buff* contents to the file associated with the open file descriptor *file* and will put the index into *buff* of the last POSIX character written into *last*, and the exception [POSIX_Error] shall not be raised.

TR: Test with *masked_signals* set to each of the three defined values and with the parameter left at its default and not used.

R_01

Setup: When no POSIX characters can be written and the file is in blocking mode.

Test: A call to *write*() a POSIX character to the file raises an exception.

See: [Assertion(s) 5-12 in X.Y.Z.]

03

Setup: Open a file with the *Blocking* option set to *False* and set up a *Write* that would otherwise block.

Test: A call *Write*(*file, buff, last, masked_signals*) to such a file will return immediately and set *last* to *Buffer'first-1*.

TR: Test with *masked_signals* set to each of the three defined values and with the parameter left at its default and not used.

04

Setup: Create a pipe and FIFO to be used as `file` in the test.

Test: A call *Write*(*file, buff, last, masked_signals*) will transfer *Limit* POSIX characters in a single operation.

TR: Test for both a pipe and FIFO *file* and for *CLimit* as *POSIX_Configurable_File_limits.Pipe_Length_Limit-1*, *POSIX_Configurable_File_limits.Pipe_Length_Limit*, *POSIX_Configurable_File_limits.Pipe_Length_Limit+1*. Test with *masked_signals* set to each of the three defined values and with the parameter left at its default and not used.

Note: See Reason 3 in Section 5 of POSIX.3.

See: [Assertion(s) GA in X.Y.Z.]

05

Setup: Ensure *file* is not open.

Test: A call *Write*(*file, buff, last, masked_signals*) raises the exception [POSIX_Error], and [Get_Error_Code] returns the value *Bad_File_Descriptor*.

06

Setup: Open *file* that is not open for writing.

Test: A call *Write*(*file, buff, last, masked_signals*) raises the exception [POSIX_Error], and [Get_Error_Code] returns the value *Bad_File_Descriptor*.

07

Setup: Interrupt, by a signal, a call to *Write*(*file, buff, last, masked_signals*).

Test: The call to *Write*() raises the exception [POSIX_Error], and [Get_Error_Code] returns the value *Interrupted_Operation*.

08

If Job Control is supported:

Setup: Create a process in a background process group.

Test: A call to *write*() to the control terminal from a process in a background process group raises the exception [POSIX_Error], and [Get_Error_Code] returns the value *Input_Output_Error*.

TR: Test for both an orphaned process group and ignoring the *POSIX_Signals.Signal_Terminal_Output* signal. Test with *masked_signals* set to each of the three defined values and with the parameter left at its default and not used.

Else No_Option

09

Setup: Create both a pipe and a FIFO and open only the write end.

Test: A call *Write*(*file, buff, last, masked_signals*) raises the exception [POSIX_Error] and [Get_Error_Code] returns the value [Broken_Pipe].

TR: Test for both a pipe and FIFO as *file*. Test with *masked_signals* set to each of the three defined values and with the parameter left at its default and not used.

10

Setup: Create a file that is just under the implementation limit on maximum file size but still can fit on the device on which it resides. Write to the file with the maximum implementation limit exceeded.

Test: A call to *Write*(*file, buff, last, masked_signals*) raises the exception [POSIX_Error] and [Get_Error_Code] returns the value *File_Too_Large*.

TR: Test with *masked_signals* set to each of the three defined values and with the parameter left at its default and not used.

Note: This may be untestable on systems with no limit on file size or with a very large limit.