

IEEE Standard for Information Technology— Portable Operating System Interface (POSIX™)— Part 1: System Application Program Interface (API)— Amendment 6: Tracing [C Language]

Sponsor

Portable Application Standards Committee
of the
IEEE Computer Society

Approved 21 September 2000

IEEE-SA Standards Board

Abstract: This amendment is part of the POSIX series of standards for applications and user interfaces to open systems. The purpose of this project is to specify a set of interfaces to allow portable access to underlying trace management services by application programs. These underlying services provide access to streams of event data, where the events can be generated by software and/or hardware, either from the application or the operating system. This standard is stated in terms of its C language binding.

Keywords: API, application portability, data processing, open systems, operating system, portable application, POSIX, tracing, instrumented operating system, instrumented application, trace control, trace analysis, real time, serviceability

POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright ©2001 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 20 July 2001. Printed in the United States of America.

Print:	ISBN 0-7381-2476-1	SH94847
PDF:	ISBN 0-7381-2477-X	SS94847

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied “**AS IS.**”

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE is the sole entity that may authorize the use of certification marks, trademarks, or other designations to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Contents

	PAGE
Introduction	iv
Organization of This Standard	iv
Conventions	v
Section 1: General	1
1.3 Conformance	1
1.3.1 Implementation Conformance	1
Section 2: Terminology and General Requirements	3
2.2 Definitions	3
2.2.2 General Terms	3
2.5 Primitive System Data Types	6
2.7 C Language Definitions	7
2.7.3 Headers and Function Prototypes	7
2.8 Numerical Limits	9
2.8.2 Minimum Values	9
2.8.4 Run-Time Invariant Values (Possibly Indeterminate)	10
2.9 Symbolic Constants	11
2.9.3 Compile-Time Symbolic Constants for Portability Specifications	11
Section 3: Process Primitives	13
3.1 Process Creation and Execution	13
3.1.1 Process Creation	13
3.1.2 Execute a File	14
3.2 Process Termination	14
3.2.2 Terminate a Process	14
3.3 Signals	15
3.3.1 Signals Concepts	15
Section 4: Process Environment	17
4.8 Configurable System Variables	17
4.8.1 Get Configurable System Variables	17
Section 18: Thread Cancellation	19
18.1 Thread Cancellation Overview	19
Section 21: Tracing	21
21.1 Introduction	21
21.2 Data Definitions	22
21.2.1 Structures	22
21.2.2 Trace Stream Attributes	27
21.2.3 Trace Event Type Definitions	28
21.3 Trace Functions	32

21.3.1	Introduction	32
21.3.2	Trace Stream Attributes Object Initialization	33
21.3.3	Functions to Retrieve and Set Information About a Trace Stream	34
21.3.4	Functions to Retrieve and Set the Behavior of a Trace Stream	36
21.3.5	Functions to Retrieve and Set Trace Stream Size Attributes	39
21.3.6	Trace Stream Initialization, Flush, and Shutdown from a Process	42
21.3.7	Clear Trace Stream and Trace Log	46
21.3.8	Manipulate Trace Event Type Identifier	47
21.3.9	Iterate over a Mapping of Trace Event Types	49
21.3.10	Manipulate Trace Event Type Sets	51
21.3.11	Set Filter of an Initialized Trace Stream	52
21.3.12	Trace Start and Stop	54
21.3.13	Trace Functions for Instrumenting Application Code	55
21.3.14	Trace Log Management	57
21.3.15	Functions to Retrieve the Trace Attributes or Trace Statuses	59
21.3.16	Functions to Retrieve a Trace Event	60
Annex B	(informative) Rationale and Notes	65
B.21.1	Introduction	65
B.21.2	Data Definitions	65
B.21.3	Trace Functions	66
B.21.4	Objectives	67
B.21.5	Objectives	71
B.21.6	Trace Model	75
B.21.7	Rationale on Trace for Debugging	89
B.21.8	Rationale on Trace Event Type Namespace	90
B.21.9	Rationale on Trace Events Type Filtering	92
B.21.10	Pthread API	94
B.21.11	Rationale on Triggering	95
B.21.12	Rationale on Timestamp Clock	96
B.21.13	Rationale on Different Overrun Conditions	96
Identifier Index		99
Alphabetic Topical Index		103

Introduction

(This introduction is not a normative part of IEEE Std 1003.1q-2000, IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 6: Tracing [C Language], but is included for information only.)

The purpose of this amendment is to specify a set of interfaces to allow portable access to underlying tracing management services by application programs. These underlying services provide access to streams of event data, where the events can be generated by software and/or hardware, either from the application or the operating system. This amendment is part of the POSIX series of standards for applications and user interfaces to open systems. It is intended for systems implementors and application software developers, and is complementary to:

- ISO/IEC 9945-1:1996 (IEEE Std 1003.1, 1996 Edition) (first in a family of “POSIX” standards), which specifies operating system interfaces and source code level functions based on the UNIX® system documentation¹⁾,
- ISO/IEC 9899:1990 which specifies the C Language,
- ISO/IEC 646:1991 which specifies the ISO 7-bit coded character.

The secondary purpose of this standard is to supplement the application portability interfaces to promote the “portability” of users and programmers between conforming systems. It is intended for end users, program developers, systems implementors, and application software developers.

The majority of this standard describes the functions that can interface with application programs. These language-independent service interfaces are temporarily described in terms of their C-language bindings. The C language assumed is that defined by the C Standard: ISO/IEC 9899:1990 {7}.

Organization of This Standard

This standard is divided into several parts:

- General, conformance requirement (Section 1).
- Definitions, general requirements, conventions, normative references, and the environment available to applications (Section 2).
- Impact on process primitives (Section 3).
- Impact on process environment (Section 4).
- Impact on thread cancellation (Section 18).
- Trace interface (Section 21).

1) UNIX is a registered trademark of The Open Group in the USA and other countries.

— Rationale (Annex B).

This introduction, any footnotes, notes accompanying the text, and the *informative* annexes are not considered part of this standard.

Conventions

This document uses the following typographic conventions:

- (1) The *italic* font is used for:
 - Cross references to defined terms within 1.3, 2.2.1, and 2.2.2; symbolic parameters that are generally substituted with real values by the application
 - C language data types and function names (except in function Synopsis sub-clauses)
 - Global external variable names
 - Function families; references to groups of closely related functions (such as *directory*, and *exec*)

- (2) The **bold** font is used with a word in all capital letters, such as

PATH

to represent an environment variable. It is also used for the term “**NULL** pointer.”

- (3) The **constant-width** (Courier) font is used:
 - For C language data types and function names within function Synopsis sub-clauses
 - To illustrate examples of system input or output where exact usage is depicted
 - For references to utility names and C language headers
 - For names of attributes in attributes objects

- (4) Symbolic constants returned by many functions as error numbers are represented as:

[ERRNO]

See 2.4.

- (5) Symbolic constants or limits defined in certain headers are represented as:

{OPEN_MAX}

See 2.8 and 2.9.

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.

The conventions listed previously are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

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).

Numerical quantities are presented in international style: comma is used as a decimal sign and units are from the International System (SI).

IEEE Std 1003.1q-2000 was prepared by the Systems Services Working Group—Realtime, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society. When the IEEE Standards Board approved this standard on 21 September 2000 the membership of the System Services Working Group was as follows:

**Portable Applications Standards Committee
(PASC)**

Lowell G. Johnson, *Chair*

Joseph M. Gwinn, *Vice Chair*

Jay Ashford, *Functional Chair*

Andrew Josey, *Functional Chair*

Curtis Royster, *Functional Chair*

Nick Stoughton, *Secretary*

IEEE System Services Working Group — Realtime

Joseph M. Gwinn, *Chair*

Karen D. Gordon, *Secretary*

Franklin C. Prindle, *Secretary (1996)*

François Riche, *Editor*

Pierre-Jean Arcos, *Technical Reviewer*

François Riche, *Technical Reviewer*

François Riche, *Ballot Coordinator*

Working Group

Pierre-Jean Arcos
Ted Baker
Robert Barned
Julian Blake
Steven J. Brosky
Steven J. Dovich
Christoph Eck
Michael Gonzalez
Karen D. Gordon
Robert Green

Joseph M. Gwinn
Steven A. Haaser
Steven R. Kleiman
Mike Lane
Kent Long
Robert D. Luken
James T. Oblinger
Diane Paul
Franklin C. Prindle
Jaikumar Ramanathan

François Riche
Helmut Roth
Allan Rowe
Richard Scalzo
Jim Shaffer
Steve Watt
Andrew E. Wheeler, Jr.
John Zolnowsky

The following persons of the 1003.1q Balloting Group voted on this standard. Balloters may have voted for approval, disapproval, or abstention:

Pierre-Jean Arcos
Robert Barned
Keith Bierman
David J. Blackwood
Julian Blake
Shirley Bockstahler-Brandt
Mark Brown
Eric Burger
Larry Chandler
Donald Cragun
John S. Davies
Juan Antonio de la Puente
Steven J. Dovich
Michael Gonzalez
Karen D. Gordon
Mars J. Gralia

Joseph M. Gwinn
Steven A. Haaser
Charles E. Hammons
Chris J. Harding
Bryan D. Harold
Barry Hedquist
Richard A. Jaenicke
Petr Janecek
Lowell G. Johnson
Andrew Josey
Martin J. Kirk
Steven R. Kleiman
Thomas M. Kurihara
Bruce Lewis
Roger J. Martin
James T. Oblinger
Diane Paul

John Pijanowski
Franklin C. Prindle
François Riche
Helmut Roth
Curtis Royster
Richard Scalzo
Herve Schauer
Diane Schleicher
Jacob Slonim
William R. Smith
Fred Starkey
Michael W. Vannier
Charlotte Wales
Dale G. Wolfe
Oren Yuen
Janusz Zalewski

The following organizational representative voted on this standard:

Andrew Josey

X/Open Co.Ltd.

When the IEEE-SA Standards Board approved this standard on 21 September 2000, it had the following membership:

Donald N. Heirman, *Chair*

James T. Carlo, *Vice Chair*

Judith Gorman, *Secretary*

Satish K. Aggarwal
Mark D. Bowman
Gary R. Engmann
Harold E. Epstein
H. Landis Floyd
Jay Forster*
Howard M. Frazier
Ruben D. Garzon

James H. Gurney
Richard J. Holleman
Lowell G. Johnson
Robert J. Kennelly
Joseph L. Koepfinger*
Peter H. Lips
L. Bruce McClung
Daleep C. Mohla

James W. Moore
Robert F. Munzner
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Akio Tojo
Donald W. Zispe

*Member emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Alan Cookson, *NIST Representative*

Donald R. Volzka, *TAB Representative*

Yvette Ho Sang
IEEE Standards Project Editor

IEEE Standard for Information Technology— Portable Operating System Interface (POSIX®) — Part 1: System Application Program Interface (API) — Amendment 6: Tracing [C Language]

Section 1: General

1.3 Conformance

1.3.1 Implementation Conformance

==> **1.3.1.3 Conforming Implementation Options** *Add to the table of implementation options that warrant requirement by applications or in specifications:*

{_POSIX_TRACE}	Trace option (in 2.9.3)
{_POSIX_TRACE_EVENT_FILTER}	Trace Event Filter option (in 2.9.3)
{_POSIX_TRACE_LOG}	Trace Log option (in 2.9.3)
{_POSIX_TRACE_INHERIT}	Trace Inheritance option (in 2.9.3)

Section 2: Terminology and General Requirements

2.2 Definitions

2.2.2 General Terms

==> **2.2.2 General Terms** *Modify the contents of 2.2.2 to add the following definitions in the correct sorted order [disregarding the subclause numbers shown here]:*

2.2.2.1 instrumented application: An application that contains at least one call to the trace point function `posix_trace_event()`. Each process of an instrumented application has a mapping of trace event names to trace event type identifiers. This mapping is used by the trace stream that is created for that process.

2.2.2.2 system trace event: A trace event that is generated by the implementation, in response either to a system-initiated action or to an application-requested action, except for a call to `posix_trace_event()`. When supported by the implementation, a system-initiated action generates a process-independent system trace event and an application-requested action generates a process-dependent system trace event. For a system trace event not defined by this standard, the associated trace event type identifier is derived from the implementation-defined name for this trace event, and the associated data is of implementation-defined content and length.

2.2.2.3 trace analyzer process: A process that extracts trace events from a trace stream to retrieve information about the behavior of an application. A trace controller process may also be a trace analyzer process. Trace analysis can be done concurrently with the traced process or can be done off-line, in the same or in a different platform.

2.2.2.4 trace controller process: A process that creates a trace stream for tracing a process. Only the trace controller process has control of the trace stream it has created. The control of the operation of a trace stream is done using its corresponding trace stream identifier. The trace controller process shall be able to:

- (1) initialize the attributes of a trace stream,
- (2) create the trace stream,

- (3) start and stop tracing,
- (4) know the mapping of the traced process,
- (5) if the Trace Event Filter option is supported, filter the type of trace events to be recorded,
- (6) shut the trace stream down.

A traced process may also be a trace controller process. Only the trace controller process can control its trace stream(s). A trace stream created by a trace controller process shall be shut down if its controller process terminates or executes another file.

2.2.2.5 trace event: A data object that represents an action executed by the system, and that is recorded in a trace stream. Each trace event is of a particular trace event type, and is associated with a trace event type identifier. The execution of a trace point shall generate a trace event if a trace stream has been created and started for the process that executed the trace point and if the corresponding trace event type identifier is not ignored by filtering.

A generated trace event should be recorded in a trace stream and optionally also in a trace log if a trace log was associated with the trace stream.

The only case in which a generated trace event is not recorded in the trace stream is when no resources are available for it in the trace stream. In this case, the trace event is lost.

The only two cases in which a generated trace event is not recorded in the trace log are when no resources are available for it in the trace log or when a flush operation does not succeed.

A trace event recorded in an active trace stream may be retrieved by an application having the appropriate privileges.

A trace event recorded in a trace log may be retrieved by an application having the appropriate privileges after opening the trace log as a pre-recorded trace stream, with the function *posix_trace_open()*.

When a trace event is reported it shall be possible to retrieve the following:

- (1) a trace event type identifier,
- (2) a timestamp,
- (3) the process id of the traced process, if the trace event is process-dependent,
- (4) any optional trace event data including its length,
- (5) if the Threads option is supported, the thread id, if the trace event is process-dependent,
- (6) the program address at which the trace point was invoked.

2.2.2.6 trace event type: A data object type that defines a class of trace event. A trace event type is identified on the one hand by a trace event type name, also referenced as a trace event name, and on the other hand by a trace event type identifier. A trace event name is a human-readable string. A trace event type identifier is an opaque identifier used by the trace system. There is a one-to-one relationship between trace event type identifiers and trace event names for a given trace stream and also for a given traced process. The trace event type identifier is generated automatically from a trace event name by the trace system either when a trace controller process invokes *posix_trace_trid_eventid_open()* or when an instrumented application

process invokes *posix_trace_eventid_open()*. Trace event type identifiers are used to filter trace event types, to allow interpretation of user data, and to identify the kind of trace point that generated a trace event.

2.2.2.7 trace event type mapping: A one-to-one mapping between trace event types and trace event names. One such mapping is associated with each trace stream. An active trace stream is associated with a traced process, and also with its children if the Trace Inheritance option is supported and also the inheritance policy is set to `POSIX_TRACE_INHERITED`. Therefore each traced process has a mapping of the trace event names to trace event type identifiers that have been defined for that process.

2.2.2.8 trace filter: A filter that allows the trace controller process to specify those trace event types that are to be ignored, that is, not generated. The operation of the filter is to filter out (ignore) selected trace events. By default, no trace events are filtered.

2.2.2.9 trace generation version: A data object that is an implementation-defined character string, generated by the trace system and describing the origin and version of the trace system.

2.2.2.10 trace log: The flushed image of a trace stream, if the trace stream is created with a trace log. The trace log is recorded when the *posix_trace_shutdown()* operation is invoked, or during tracing, depending on the tracing strategy which is defined by a log policy. After the trace stream has been shut down, the trace information can be retrieved from the associated trace log using the same interface used to retrieve information from an active trace stream.

2.2.2.11 trace point: An action that may cause a trace event to be generated. This may be an implementation-defined action such as a context switch, or an application-programmed action such as a call to a specific operating system service (e.g., *fork()*) or a call to *posix_trace_event()*.

2.2.2.12 tracing status of a trace stream: A status that describes the state of an active trace stream. The tracing status of a trace stream can be retrieved from the trace stream attributes. An active trace stream can be in one of two states: running or suspended.

2.2.2.13 trace stream: An opaque object that contains trace events plus internal data needed to interpret those trace events. The implementation and format of a trace stream are unspecified. A trace stream need not be and generally is not persistent. A trace stream may be either active or pre-recorded:

- an active trace stream is a trace stream that has been created and has not yet been shut down. It can be of one of the two following classes:
 - an active trace stream without a trace log that was created with the *posix_trace_create()* function, or
 - if the Trace Log option is supported, an active trace stream with a trace log that was created with the *posix_trace_create_withlog()* function;
- a pre-recorded trace stream is a trace stream that was opened from a trace log object using the *posix_trace_open()* function.

An active trace stream can loop. This behavior means that when the resources allocated by the trace system for the trace stream are exhausted, the trace system shall reuse the resources associated with the oldest recorded trace events to record new trace events.

If the Trace Log Option is supported, an active trace stream with a trace log can be flushed. This operation causes the trace system to write trace events from the trace stream to the associated trace log, following the defined policies or using an explicit function call. After this operation, the trace system may reuse the resources associated with the flushed trace events.

An active trace stream with or without a trace log can be cleared. This operation causes all the resources associated with this trace stream to be reinitialized. The trace stream behaves as if it was returning from its creation, except that the mapping of trace event type identifiers to trace event names is not cleared. If a trace log was associated with this trace stream, the trace log is also reinitialized.

2.2.2.14 trace stream identifier: A handle to manage tracing operations in a trace stream.

2.2.2.15 trace system: A system that allows both system and user trace events to be generated into a trace stream. These trace events can be retrieved later.

2.2.2.16 traced process: A process for which at least one trace stream has been created. A traced process is also called a “target process”. If the Trace Inheritance option is supported and the trace stream’s inheritance attribute is `POSIX_TRACE_INHERITED`, the initial targeted traced process shall be traced together with all of its future children. The “`posix_pid`” member of each trace event in a trace stream is the process id of the traced process.

2.2.2.17 user trace event: A trace event that is generated explicitly by the application as a result of a call to `posix_trace_event()`.

2.5 Primitive System Data Types

==> **2.5 Primitive System Data Types** *Add the following datatypes to the list of types for which there are no defined comparison or assignment operations:*

`trace_attr_t`, `trace_id_t`, `trace_event_id_t`, `trace_event_set_t`.

==> **2.5 Primitive System Data Types** *Add the following paragraphs after the paragraph starting “There are no defined comparison...” :*

An implementation need not provide the types `trace_attr_t`, `trace_id_t`, `trace_event_id_t`, and `trace_event_set_t` unless the Trace option is supported.

==> **2.5 Primitive System Data Types** *Modify the contents of 2.5 to add the following symbols into Table 2-2, in sorted order:*

Table 2-2 — Optional Primitive System Data Types

Defined Type	Description	Implementation Option
<i>trace_attr_t</i>	Used to define a trace stream attributes object	Trace
<i>trace_id_t</i>	Used for trace stream identifiers	Trace
<i>trace_event_id_t</i>	Used to identify a trace event type	Trace
<i>trace_event_set_t</i>	Used to define a set of trace event type identifiers	Trace

If the Trace option is supported, this standard does not define assignment or comparison operators for the type *trace_attr_t*. The type *trace_event_set_t* shall be used to represent a set of trace event type identifiers. Several functions used to manipulate objects of type *trace_event_set_t* are defined in 21.3.10.

2.7 C Language Definitions

2.7.3 Headers and Function Prototypes

==> **2.7.3 Headers and Function Prototypes** *Modify the contents of 2.7.3 to add the following header containing the following functions, in the correct sorted order.*

If the Trace option is supported:

```
<trace.h>    posix_trace_attr_destroy(), posix_trace_attr_getclockres(),
              posix_trace_attr_getcreatetime(),
              posix_trace_attr_getgenversion(),
              posix_trace_attr_getinherited(),
              posix_trace_attr_getlogfullpolicy(),
              posix_trace_attr_getlogsize(),
              posix_trace_attr_getmaxdatasize(),
              posix_trace_attr_getmaxsystemevents(),
              posix_trace_attr_getmaxuserevents(),
              posix_trace_attr_getname(),
              posix_trace_attr_getstreamfullpolicy(),
              posix_trace_attr_getstreamsize(), posix_trace_attr_init(),
              posix_trace_attr_setinherited(),
              posix_trace_attr_setlogfullpolicy(),
              posix_trace_attr_setlogsize(), posix_trace_attr_setmaxdatasize(),
              posix_trace_attr_setname(), posix_trace_attr_setstreamsize(),
              posix_trace_attr_setstreamfullpolicy(), posix_trace_clear(),
              posix_trace_close(), posix_trace_create(),
```

*posix_trace_create_withlog(), posix_trace_event(),
posix_trace_eventid_equal(), posix_trace_eventid_get_name(),
posix_trace_eventid_open(),
posix_trace_eventtypelist_getnext_id(),
posix_trace_eventtypelist_rewind(), posix_trace_eventset_add(),
posix_trace_eventset_del(), posix_trace_eventset_empty(),
posix_trace_eventset_fill(), posix_trace_eventset_ismember(),
posix_trace_flush(), posix_trace_get_attr(),
posix_trace_get_filter(), posix_trace_get_status(),
posix_trace_getnext_event(), posix_trace_open(),
posix_trace_rewind(), posix_trace_set_filter(),
posix_trace_shutdown(), posix_trace_start(), posix_trace_stop(),
posix_trace_timedgetnext_event(),
posix_trace_trid_eventid_open(), posix_trace_trygetnext_event().*

2.8 Numerical Limits

2.8.2 Minimum Values

==> **2.8.2 Minimum Values** *Modify the contents of 2.8.2 to add the following symbols into Table 2-4, in sorted order.*

Table 2-4 — Optional Minimum Values

Name	Description	Value	Implementation Option
{_POSIX_TRACE_SYS_MAX}	The number of trace streams that may simultaneously exist in the system.	8	Trace
{_POSIX_TRACE_USER_EVENT_MAX}	The number of user trace event type identifiers that may simultaneously exist in a traced process, including the predefined user trace event (21.2.3.2).	32	Trace
{_POSIX_TRACE_NAME_MAX}	The length, in bytes, of a trace generation version string or a trace stream name.	64	Trace
{_POSIX_TRACE_EVENT_NAME_MAX}	The length, in bytes, of a trace event name.	30	Trace

2.8.4 Run-Time Invariant Values (Possibly Indeterminate)

==> **2.8.4 Run-Time Invariant Values** *Modify the contents of 2.8.4 to add the following symbols into Table 2-6, in sorted order.*

Table 2-6 — Optional Run-Time Invariant Values

Name	Description	Minimum Value	Implementation Option
{TRACE_SYS_MAX}	The maximum number of trace streams that may simultaneously exist in the system.	{_POSIX_TRACE_SYS_MAX}	Trace
{TRACE_USER_EVENT_MAX}	The maximum number of user trace event type identifiers that may simultaneously exist in a traced process, including the predefined user trace event (21.2.3.2).	{_POSIX_TRACE_USER_EVENT_MAX}	Trace
{TRACE_NAME_MAX}	The maximum length of the trace generation version string or of the trace stream name.	{_POSIX_TRACE_NAME_MAX}	Trace
{TRACE_EVENT_NAME_MAX}	The maximum length of the trace event name.	{_POSIX_TRACE_EVENT_NAME_MAX}	Trace

2.9 Symbolic Constants

2.9.3 Compile-Time Symbolic Constants for Portability Specifications

==> 2.9.3 Compile-Time Symbolic Constants for Portability Specifications *Modify the contents of the 2.9.3 to add the following symbols in Table 2-11, in sorted order.*

Table 2-11 — Versioned Compile-Time Symbolic Constants

Name	Value	Description
{_POSIX_TRACE}	200009L	If this symbol is defined, the implementation supports the Trace option.
{_POSIX_TRACE_EVENT_FILTER}	200009L	If this symbol is defined, the implementation supports the Trace Event Filter option.
{_POSIX_TRACE_LOG}	200009L	If this symbol is defined, the implementation supports the Trace Log option.
{_POSIX_TRACE_INHERIT}	200009L	If this symbol is defined, the implementation supports the Trace Inheritance option.

==> 2.9.3 Compile-Time Symbolic Constants for Portability Specifications *Add the following paragraph:*

If any of the symbols {_POSIX_TRACE_EVENT_FILTER}, {_POSIX_TRACE_LOG}, or {_POSIX_TRACE_INHERIT} is defined, then the symbol {_POSIX_TRACE} shall also be defined.

Section 3: Process Primitives

3.1 Process Creation and Execution

3.1.1 Process Creation

==> **3.1.1.2 Description** *Add the following paragraphs at the end of the numbered list of items that are not an exact copy of the calling process's items [disregarding the number shown here]:*

(17) If the Trace option and the Trace Inheritance option are both supported:

If the calling process was being traced in a trace stream that had its inheritance policy set to `POSIX_TRACE_INHERITED`, the child process shall be traced into that trace stream, and the child process shall inherit the parent's mapping of trace event names to trace event type identifiers.

If the trace stream in which the calling process was being traced had its inheritance policy set to `POSIX_TRACE_CLOSE_FOR_CHILD`, the child process shall not be traced into that trace stream.

The inheritance policy is set by a call to the `posix_trace_attr_setinherited()` function (21.3.4).

Otherwise, if the Trace option is supported but the Trace Inheritance option is not supported:

The child process shall not be traced into any of the trace streams of its parent process.

If the Trace option is supported, the child process of a trace controller process shall not control the trace streams controlled by its parent process.

==> 3.1.1.5 Cross-References

Add the following cross-references:

`posix_trace_attr_setinherited()`, 21.3.4; `posix_trace_eventid_open()`, 21.3.13;
`posix_trace_trid_eventid_open()`, 21.3.8.

3.1.2 Execute a File

==> **3.1.2.2 Description** *Add the following paragraphs before the paragraph starting "The new process image also inherits the following attributes ...":*

If the Trace option is supported and the calling process image is being traced, the new process image continues to be traced into the same trace stream as the original process image, but the new process image shall not inherit the mapping of trace event names to trace event type identifiers that was defined by calls to the *posix_trace_eventid_open()* or the *posix_trace_trid_eventid_open()* functions in the calling process image.

If the Trace option is supported and the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described in the *posix_trace_shutdown()* function (21.3.6).

==> 3.1.2.5 Cross-References

Add the following cross-references:

posix_trace_shutdown(), 21.3.6; *posix_trace_trid_eventid_open()*, 21.3.8;
posix_trace_eventid_open(), 21.3.13.

3.2 Process Termination

3.2.2 Terminate a Process

==> **3.2.2.2 Description** *Add the following paragraphs at the end of the numbered list of items that are consequences of termination of a process [disregarding the number shown here]:*

(15) If the Trace option is supported and the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described by the *posix_trace_shutdown()* function (21.3.6). and any process's mapping of trace event names to trace event type identifiers built for these trace streams may be deallocated.

==> 3.2.2.4 Cross-References

Add the following cross-references:

posix_trace_shutdown(), 21.3.6; *posix_trace_trid_eventid_open()*, 21.3.8;
posix_trace_eventid_open(), 21.3.13.

3.3 Signals**3.3.1 Signals Concepts**

==> 3.3.1.3 Signal Actions *Add the following text after the table of functions that shall be async-signal safe:*

In addition, if the Trace option is supported, the following function shall also be async-signal safe:

posix_trace_event()

Section 4: Process Environment

4.8 Configurable System Variables

4.8.1 Get Configurable System Variables

==> **4.8.1.2 Get Configurable System Variables— Description** *Add the following text after the sentence “The implementation shall support all of the variables listed in Table 4-2 and may support others”, in the second paragraph:*

Support for some configuration variables is dependent on implementation options (see Table 4-3). Where an implementation option is not supported, the variable need not be supported.

==> **4.8.1.2 Get Configurable System Variables— Description** *In the second paragraph, replace the text “The variables in Table 4-2 come from ...” by the following:*

“The variables in Table 4-2 and Table 4-3 come from ...”

==> **4.8.1.2 Get Configurable System Variables— Description** *Add the following table:*

Table 4-3 — Optional Configurable System Variables

Variable	<i>name</i> Value	Required Option
{_POSIX_TRACE}	_SC_TRACE	Trace
{_POSIX_TRACE_EVENT_FILTER}	_SC_TRACE_EVENT_FILTER	Trace Event Filter
{_POSIX_TRACE_LOG}	_SC_TRACE_LOG	Trace Log
{_POSIX_TRACE_INHERIT}	_SC_TRACE_INHERIT	Trace Inheritance

Section 18: Thread Cancellation

18.1 Thread Cancellation Overview

==> **18.1.2 Cancellation Points** *Add the following functions to the list of functions for which a cancellation point may also occur:*

<i>posix_trace_clear()</i>	<i>posix_trace_close()</i>
<i>posix_trace_create()</i>	<i>posix_trace_create_withlog()</i>
<i>posix_trace_eventtypelist_getnext_id()</i>	<i>posix_trace_eventtypelist_rewind()</i>
<i>posix_trace_flush()</i>	<i>posix_trace_get_attr()</i>
<i>posix_trace_get_filter()</i>	<i>posix_trace_get_status()</i>
<i>posix_trace_getnext_event()</i>	<i>posix_trace_open()</i>
<i>posix_trace_rewind()</i>	<i>posix_trace_set_filter()</i>
<i>posix_trace_shutdown()</i>	<i>posix_trace_timedgetnext_event()</i>

Section 21: Tracing

21.1 Introduction

This section describes extensions to support tracing of user applications. The facilities defined in this chapter allow a process to select a set of trace event types, to activate a trace stream of the selected trace events as they occur in the flow of execution, and to retrieve the recorded trace events.

The tracing operation relies on three logically different components: the traced process, the controller process, and the analyzer process. During the execution of the traced process, when a trace point is reached, a trace event is recorded into the trace streams created for that process in which the associated trace event type identifier is not being filtered out. The controller process controls the operation of recording the trace events into the trace stream. It shall be able to:

- initialize the attributes of a trace stream,
- create the trace stream (for a specified traced process) using those attributes,
- start and stop tracing for the trace stream,
- filter the type of trace events to be recorded, if the Trace Event Filter option is supported,
- shut a trace stream down.

These operations can be done for an active trace stream. The analyzer process retrieves the traced events either at run time, when the trace stream has not yet been shut down, but is still recording trace events; or after opening a trace log that had been previously recorded and shut down. These three logically different operations can be performed by the same process, or can be distributed into different processes.

A trace stream identifier can be created by a call to *posix_trace_create()*, *posix_trace_create_withlog()*, or *posix_trace_open()* functions. The *posix_trace_create()* and *posix_trace_create_withlog()* functions should be used by a controller process. The *posix_trace_open()* function should be used by an analyzer process.

The tracing functions can serve different purposes. One purpose is debugging the possibly pre-instrumented code, while another is post-mortem fault analysis. These two potential uses differ in that the first requires pre-filtering capabilities to avoid overwhelming the trace stream and permits focusing on expected information; while the second needs comprehensive trace capabilities in order to be able to record all types of information.

The events to be traced belong to two classes:

- user trace events (generated by the application instrumentation), and

— system trace events (generated by the operating system).

The tracing interface defines several system trace event types associated with control of and operation of the trace stream. This small set of system trace events includes the minimum required to interpret correctly the trace event information present in the trace stream. Other desirable system trace events for some particular application profile may be implemented and are encouraged, e.g., process and thread scheduling, and signal occurrence.

Each traced process shall have a mapping of the trace event names to trace event type identifiers that have been defined for that process. Each active trace stream shall have a mapping that incorporates all the trace event type identifiers predefined by the trace system plus all the mappings of trace event names to trace event type identifiers of the processes that are being traced into that trace stream. These mappings are defined from the instrumented application by calling the *posix_trace_eventid_open()* function and from the controller process by calling the *posix_trace_trid_eventid_open()* function. For a pre-recorded trace stream, the list of trace event types is obtained from the pre-recorded trace log.

The *st_ctime* and *st_mtime* fields of a file associated with an active trace stream shall be marked for update every time any of the tracing operations modifies that file.

The *st_atime* field of a file associated with trace stream shall be marked for update every time any of the tracing operations causes data to be read from that file.

Results are undefined if the application performs any operation on a file descriptor associated with an active or pre-recorded trace stream until *posix_trace_shutdown()* or *posix_trace_close()* is called for that trace stream.

The main purpose of this section is to define a complete set of functions and concepts that allow a portable application to be traced from birth to death, whatever its realtime constraints and properties.

21.2 Data Definitions

21.2.1 Structures

The header `<trace.h>` shall define the *posix_trace_status_info* and *posix_trace_event_info* structures described below. Implementations may add extensions to these structures as permitted in 1.3.1.1, item (2).

21.2.1.1 *posix_trace_status_info* Structure

To facilitate control of a trace stream, information about the current state of an active trace stream can be obtained dynamically. This structure is returned by a call to the *posix_trace_get_status()* function (21.3.15).

If the Trace option is supported, the *posix_trace_status_info* structure defined in `<trace.h>` shall contain at least the following members:

Member Type	Member Name	Description
int	<code>posix_stream_status</code>	the operating mode of the trace stream
int	<code>posix_stream_full_status</code>	the full status of the trace stream
int	<code>posix_stream_overrun_status</code>	indicates whether trace events were lost in the trace stream

In addition if the Trace option and the Trace Log option are both supported, the *posix_trace_status_info* structure defined in `<trace.h>` shall contain at least the following additional members:

Member Type	Member Name	Description
int	<code>posix_stream_flush_status</code>	indicates whether a flush is in progress
int	<code>posix_stream_flush_error</code>	indicates whether any error occurred during the last flush operation
int	<code>posix_log_overrun_status</code>	indicates whether trace events were lost in the trace log
int	<code>posix_log_full_status</code>	the full status of the trace log

If the Trace option is supported, the *posix_stream_status* member indicates the operating mode of the trace stream and shall have one of the following values defined by manifest constants in the header `<trace.h>`:

`POSIX_TRACE_RUNNING`

Tracing is in progress; that is, the trace stream is accepting trace events.

`POSIX_TRACE_SUSPENDED`

The trace stream is not accepting trace events. The tracing operation has not yet started or has stopped, either following a *posix_trace_stop()* function call or because the trace resources are exhausted.

If the Trace option is supported, the *posix_stream_full_status* member indicates the full status of the trace stream, and it shall have one of the following values defined by manifest constants in the header `<trace.h>`:

`POSIX_TRACE_FULL`

The space in the trace stream for trace events is exhausted.

`POSIX_TRACE_NOT_FULL`

There is still space available in the trace stream.

If the Trace option is supported, the combination of the *posix_stream_status* and *posix_stream_full_status* members also indicates the actual status of the stream. The status shall be interpreted as follows:

`POSIX_TRACE_RUNNING` and `POSIX_TRACE_NOT_FULL`

This status combination indicates that tracing is in progress, and there is space available for recording more trace events.

POSIX_TRACE_RUNNING and POSIX_TRACE_FULL

This status combination indicates that tracing is in progress and that the trace stream is full of trace events. This status combination cannot occur unless the *stream-full-policy* is set to **POSIX_TRACE_LOOP**. The trace stream contains trace events recorded during a moving time window of prior trace events, and some older trace events may have been overwritten and thus lost.

POSIX_TRACE_SUSPENDED and POSIX_TRACE_NOT_FULL

This status combination indicates that tracing has not yet been started, has been stopped by the *posix_trace_stop()* function or has been cleared by the *posix_trace_clear()* function.

POSIX_TRACE_SUSPENDED and POSIX_TRACE_FULL

This status combination indicates that tracing has been stopped by the implementation because the *stream-full-policy* attribute was **POSIX_TRACE_UNTIL_FULL** and trace resources were exhausted, or that the trace stream was stopped by the function *posix_trace_stop()* at a time when trace resources were exhausted.

If the Trace option is supported, the *posix_stream_overrun_status* member indicates whether trace events were lost in the trace stream, and shall have one of the following values defined by manifest constants in the header `<trace.h>`:

POSIX_TRACE_OVERRUN

At least one trace event was lost and thus was not recorded in the trace stream.

POSIX_TRACE_NO_OVERRUN

No trace events were lost.

When the corresponding trace stream is created, the *posix_stream_overrun_status* member shall be set to **POSIX_TRACE_NO_OVERRUN**. Whenever an overrun occurs, the *posix_stream_overrun_status* member shall be set to **POSIX_TRACE_OVERRUN**. An overrun occurs when:

- the policy is **POSIX_TRACE_LOOP** and a recorded trace event is overwritten,
- the policy is **POSIX_TRACE_UNTIL_FULL** and the trace stream is full when a trace event is generated,
- or, if the Trace Log option is supported, the policy is **POSIX_TRACE_FLUSH** and at least one trace event is lost while flushing the trace stream to the trace log.

The *posix_stream_overrun_status* member is reset to zero after its value is read.

If the Trace option and the Trace Log option are both supported:

The *posix_stream_flush_status* member indicates whether a flush operation is being performed and shall have one of the following values defined by manifest constants in the header `<trace.h>`:

POSIX_TRACE_FLUSHING

The trace stream is currently being flushed to the trace log.

POSIX_TRACE_NOT_FLUSHING

No flush operation is in progress.

The *posix_stream_flush_status* member shall be set to `POSIX_TRACE_FLUSHING` if a flush operation is in progress either due to a call to the *posix_trace_flush()* function (explicit or caused by a trace stream shutdown operation) or because the trace stream has become full with the `stream-full-policy` attribute set to `POSIX_TRACE_FLUSH`. The *posix_stream_flush_status* member shall be set to `POSIX_TRACE_NOT_FLUSHING` if no flush operation is in progress.

The *posix_stream_flush_error* member shall be set to zero if no error occurred during flushing. If an error occurred during a previous flushing operation, *posix_stream_flush_error* member shall be set to the value of the first error that occurred. If more than one error occurs while flushing, error values after the first shall be discarded. The *posix_stream_flush_error* member is reset to zero after its value is read.

The *posix_log_overrun_status* member indicates whether trace events were lost in the trace log, and shall have one of the following values defined by manifest constants in the header `<trace.h>`:

`POSIX_TRACE_OVERRUN`
At least one trace event was lost.

`POSIX_TRACE_NO_OVERRUN`
No trace events were lost.

When the corresponding trace stream is created, the *posix_log_overrun_status* member shall be set to `POSIX_TRACE_NO_OVERRUN`. Whenever an overrun occurs, this status shall be set to `POSIX_TRACE_OVERRUN`. The *posix_log_overrun_status* member is reset to zero after its value is read.

The *posix_log_full_status* member indicates the full status of the trace log, and it shall have one of the following values defined by manifest constants in the header `<trace.h>`:

`POSIX_TRACE_FULL`
The space in the trace log is exhausted.

`POSIX_TRACE_NOT_FULL`
There is still space available in the trace log.

The *posix_log_full_status* is only meaningful if the `log-full-policy` attribute is either `POSIX_TRACE_UNTIL_FULL` or `POSIX_TRACE_LOOP`.

For an active trace stream without log, that is created by the *posix_trace_create()* function, the *posix_log_overrun_status* member shall be set to `POSIX_TRACE_NO_OVERRUN` and the *posix_log_full_status* member shall be set to `POSIX_TRACE_NOT_FULL`.

Otherwise:

the *posix_stream_flush_status*, *posix_stream_flush_error*, *posix_log_overrun_status*, and *posix_log_full_status* members are undefined.