# A Summary of the POSIX Trace Standard

by
Andrés Terrasa, Ana García-Fornes, and Agustín Espinosa
{aterrasa,agarcia,aespinos}@dsic.upv.es

September 10, 2002

## Contents

## A Summary of the POSIX Trace Standard



{aterrasa,agarcia,aespinos}@dsic.upv.es
Departamento de Sistemas Informáticos y Computación
Technical University of Valencia
Cno. de Vera s/n, 46022 Valencia (SPAIN)



**Notes:**

I hope the information contained in this document will be useful for those people interested in the POSIX Trace standard.

"POSIX" and "1003.1" are registered trademarks of the Institute of Electrical and Electronic Engineers, Inc.

Please report any suggestion, comment or error to my e-mail address above.

A. Terrasa

# 1 Introduction

Recently, IEEE has introduced *tracing* to the facilities defined by the POSIX standard. The result is called the POSIX Trace standard. According to the Webster's Revised Unabridged Dictionary, one of the definitions of *trace* is:

> "*to follow by some mark that has been left by a person or thing which has preceded; to follow by footsteps, tracks, or tokens*".

Adapting this definition to the context of a computer program, tracing can be defined as the combination of two activities: the generation of *tracing* information by a running process, and the collection of this information in order to be analyzed. Thus, the usual purpose of tracing is to find out the sequence of significant steps taken by a running program (or by the operating system in the program's behalf), in order to test, debug, tune, etc., this program. Of course, the definition of these steps and the information to be generated in each of them entirely depends on both the program characteristics and the specific purpose of the analysis to be carried out.

The POSIX Trace standard tries to standardize the way applications use tracing. In particular, it has two main objectives. Using its own words, the first objective is *"to specify a set of interfaces to allow portable access to underlying trace management services by application programs"*; and the second one is *"to supplement the application portability interfaces to promote the "portability" to users and programmers between conforming systems"*. These two objectives basically mean that a POSIX-conforming operating system should provide the application programs with an appropriate set of portable facilities for tracing. The facilities should be as general as possible in order to permit the tracing of programs of any kind and to perform different types of tracing (on-line or off-line, partial or exhaustive, etc.).

The POSIX Trace standard was initially approved in Sep. 200 by the IEEE-SA Standards Board as the amendment named "POSIX 1003.1q (tracing)". However, in Dec. 2001, IEEE finally approved the "1003.1$^{TM}$ IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®)" (IEEE Std. 1003.1-2001), a single document which superseded the two previous POSIX standards (approved in 1992 and 1996, respectively) and all the later amendments, including the tracing amendment[1]. The Trace standard is currently an option in POSIX, meaning that conforming system may freely support it or not.

The rest of this document is organized in the following way: Section 2 introduces the standard main concepts, including the main data types and facilities that the standard introduces to support tracing. Section 3 explains the different levels of support that the standard defines. These levels can individually be adopted by conforming systems as implementation options. Finally, Section 4 reviews the set of interface functions defined by the Trace standard, including a brief description and the information that in which level(s) of support they have to be included.

[1]The IEEE Std. 1003.1-2001 standard has also been approved by the Open Group under the name of "the Single UNIX® Specification (version 3)". Both the IEEE and the Open Group have the copyright of this standard.

# 2 Main Concepts

The POSIX Trace standard is founded on two main data types (trace event and trace stream) and is also based on three different roles which are played during the tracing activity: the trace controller process (the process who sets the tracing system up), the traced or target process (the process which is actually being traced), and the trace analyzer process (the process who retrieves the tracing information in order to analyze it). All these concepts are detailed in the following sections.

## 2.1 Data Types

### 2.1.1 Trace Event

When a program needs to be traced, it has to generate some information each time it reaches a "significant step" (certain instruction in the program's source code). In the POSIX Trace standard terminology, this step is called a *trace point*, and the tracing information which is generated at that point is called a *trace event*. A program containing one or more of this trace points is named *instrumented application*.

A *trace event* can be thus defined as a data object representing an action which is executed by either a running process or by the operating system. In this sense, there are two classes of trace events: *user* trace events, which are explicitly generated by an instrumented application, and *system* trace events, which are generated by the operating system[2].

Any trace event, being either system or user, belongs to a certain *trace event type* (an internal identifier, of type `trace_event_id_t`) and it is associated with a *trace event name* (a human-readable string). For system events, the definition of event types and the mapping between these types and their corresponding names is hard-coded in the implementation of the trace system. Therefore, this event types are common for all the instrumented applications and never change (they are always traced). The trace standard predefines some event types, which are related to the trace system itself, and permits the operating system designer to add some others which may be interesting to that system. The definition of user event types is very different. When an instrumented application wants to generate trace event of a particular type, it has first to create this type. This is done by invoking a particular function (`posix_trace_open()`) that, given a *new* trace event name, returns a *new* trace event type; then, events of this type can be generated from that moment on. If the event name was already registered for that application, then the previously associated identifier is returned. The mapping between user event types and their names is private to each instrumented program and lasts while the program is running.

The generation of a trace event is done internally by the trace system for a system event and explicitly (by the application when invoking `posix_trace_event()`) for a user trace event. In both

[2] System trace events can be related to an action executed by the operating system in the process' behalf (such as a system call being invoked) or due to an internal action not related to any particular process (e.g., a hardware timer expiration).

cases, the standard defines that the trace system has to store some information for each trace event being generated, including, at least, the following:

a) the trace event type identifier,

b) a timestamp,

c) the process identifier of the traced process (if the event is process-dependent),

d) the thread identifier (of the thread related to the event), if the event is process-dependent and the O.S. supports threads,

e) the program address at which the event was generated,

f) any extra data that the system or the instrumented application wants to associate with the event, along with the data size[3].

### 2.1.2 Trace Stream

When the system or an application trace an event, all the information related to it has to be stored somewhere before it can be retrieved, in order to be analyzed. This place is a *trace stream*. Formally speaking, a *trace stream* is defined as a non-persistent, internal (opaque) data object containing a sequence of trace events plus some internal information to interpret those trace events. The standard does not define a stream as a persisten object and thus it is assumed to be *volatile*, that is, to reside in main memory.

The standard establishes that, before any event can be stored for a process, a trace stream has to be explicitly created to trace that particular process (the process' pid is one of the arguments of the stream creation function). In the most general case, the relationship between streams and processes is many to many. On the one hand, many processes can be traced in a single stream; in particular, this happens if the target process *forks* after a stream has been created for the (parent) process. On the other hand, the standard permits that many streams are created to trace the same process; if so, each event generated by the process (or by the operating system) is registered in *all* these streams.

Streams also support *filtering*. The application can define and apply a filter to a trace stream. Basically, the filter establishes which event types the stream is accepting (and hence storing) and which are not. Therefore, trace events corresponding to types which are filtered out from a certain stream will not be stored in the stream. Each stream in the system(even if associated with the same process) can potentially be applied a different filter. This filter can be applied, removed or changed at any time.

The standard defines two classes of trace streams: active and pre-recorded, which are described below.

[3]The standard sets that the maximum size of extra information that an application can associate with a trace event is configurable by the application, according to its needs. If the size of the data given with a certain trace event exceeds this upper limit, then the trace system can truncate this data.

**a) Active trace stream**. This is a stream that has been created for tracing events and has not yet been shut down. This means that it is now accepting events to store. An active trace stream can be of two different types, depending on whether it has been created with or without a *log*.

In a *trace stream with log*, the stream is created along with a *log*. A log is a persistent object (that is, a file) in which the events stored in the stream are saved each time the stream is *flushed* by the trace system. The trace controller process can create such a stream by calling the function `posix_trace_create_withlog()`. Thus, events traced from the target process are stored in the stream until it is flushed, either automatically by the trace system or when the trace controller process invokes the `posix_trace_flush()` function. In either case, the flushing then frees the resources previously occupied by the events just written to the log, making these resources available for new events to be stored. This is shown in Figure 2-(a). In streams with a log, events are never directly retrieved from the stream but from the log (see "Pre-recorded trace stream" below), once the stream has been shut down. That is, the log is not available for retrieving the events until the tracing of events is over.

In a *trace stream without log* (created by calling `posix_trace_create()`), trace events are never written to any persistent media, but instead they remain in the stream (in memory) until they are explicitly retrieved. Thus, the stream is accessed *concurrently* for storing (target process) and retrieving (trace analyzer process) events. These accesses can be done only while the stream is active (that is, before it is shut down) since, after that, all the stream resources are freed. Therefore, an active trace stream without a log is used for *on-line* analysis of events, as shown in Figure 1.

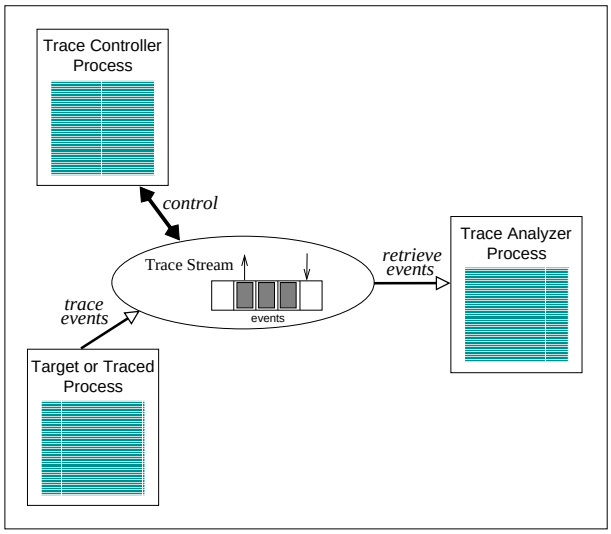


**Figure 1:** On-line tracing of events in an active stream without log.

The standard establishes that the trace analyzer process retrieves the events one by one, with the trace system always reporting the oldest stored event first. When this oldest event has been reported, the resources that it was using in the stream have to be freed and then become

available for new events to be traced.

If the rate at which events are being traced is higher than the rate at which the trace analyzer process is retrieving them from the stream, then the stream may become full. If an active stream without log becomes full, it may either stop accepting events or loop; this depends on the so-called stream *full policy*, which is one of its attributes. In the former case, the stream will start accepting events again when a certain amount of events in the stream have been retrieved, hence freeing resources for the new ones to be stored. In the latter (loop) case, when the stream is full, the oldest recorded events in the stream are lost as new events are stored (that is, the oldest events are overwritten).

**b) Pre-recorded trace stream**. A stream of this class is used for retrieving trace events which were previously stored in a log. In particular, the log file is *opened* into a (pre-recorded) stream from which events are then retrieved. Thus, *off-line* analysis of events is performed in two steps: first, events are traced into an active stream with log; second, after this stream is shut down, the log can be opened into a pre-recorded stream from which the events are retrieved. This process is shown in Figure 2.

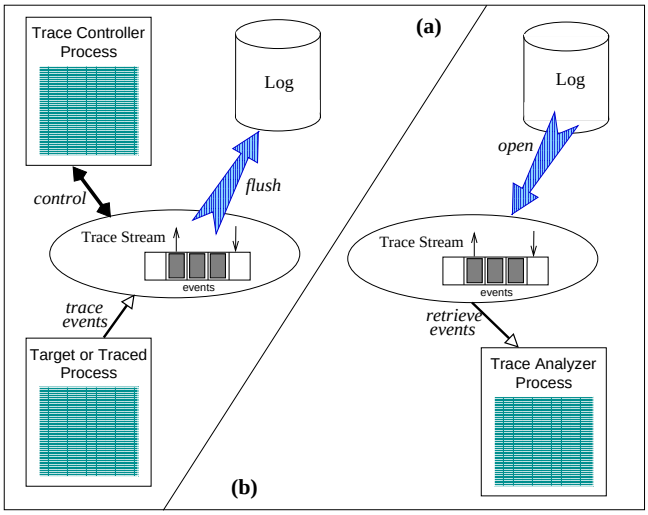


**Figure 2:** Off-line tracing: (a) Tracing of events in an active stream with log. (b) Retrieval of events from a pre-recorded stream.

## 2.2 Processes Involved in the Tracing Activity

The standard defines that up to three different roles can be played in each tracing activity: trace controller process, traced (or target) process and trace analyzer process. In the most general case, each of these roles is executed by a separate process. However, nothing in the standard prevents from having two (or even the three) of these roles executed by the same process. In a small, multi-threaded

application, we can have, for example, the three roles played by different threads inside the same process. These roles are now explained in detail.

### 2.2.1 Trace Controller Process

The *trace controller process* is the process that sets the tracing system up in order to trace a (target) process, which can be the same process or a different one. In particular, this process is in charge of, at least, the following actions:

a) *Creating* a trace stream with its particular attributes (e.g, if the stream is with or without a log, the stream full policy, etc.). This is further detailed below.

b) *Starting* and *stopping* tracing when necessary. This is done by calling `posix_trace_start()` and `posix_trace_stop()`, respectively. Each active stream can be in two different states: running or suspended. These two states determine whether or not the stream is accepting events to be stored. The trace controller process can start and stop the stream as many times as it wants. If the stream full policy is to trace until full (`POSIX_TRACE_UNTIL_FULL`), the trace system will automatically stop the stream when full and start it again when some (or all) of its stored events have been retrieved.

c) *Filtering* the types of events to be traced. Each stream is initially created with an empty filter (that is, without filtering any event type). If this is not the required behavior, the trace controller process can build a set of event types (`trace_event_set_t`), include the appropriate event types in it, and apply it as a filter to the stream (by invoking `posix_trace_set_filter()`). After that, the stream will reject any event whose type is in the filter set.

d) *Shutting* the stream *down*, when the tracing is over (`posix_trace_shutdown()`). The standard requires that shutting a stream down must free all the stream resources. That is, the stream is destroyed and no more operations can be done on it.

Among all these basic actions, the creation of the stream is the most complex one. This action is done in two steps:

1. Create a stream attribute object (`trace_attr_t`) and set each of its attributes appropriately. Since this type is also opaque to the user(that is, internal to the trace system), the standard provides a function to initialize an attribute object and then pairs of functions to get and set each of the individual attributes included in the object. Some of these attributes are: the stream name, the stream minimum size, the event data maximum size, the stream full policy, etc. This setting up is performed before invoking the call to create the stream.

2. Create the stream (`trace_id_t`). There are two different functions to create an active stream, depending on whether it has to be with or without a log. Respectively, these functions are

`posix_trace_create_withlog()` and `posix_trace_create()`. In either case, the arguments of the creation function are the stream attribute object, previously initialized and set (see above), and the target process' *pid* (process identifier).

The main implication of this is that the target process has to exist before the trace controller process can create a stream to trace it. Besides, it has to have "enough privileges" over the target to do it. The exact definition of this latter requirement depends on the implementation of the trace system.

The stream identifier returned in this function can *only* be used by the process that has created the stream. Only this process can thus directly access the stream in any way. This establishes some limitations that will be commented in Section 2.2.3 below.

Optionally, the trace controller process can also perform other actions on the stream, once the stream has been created:

- *Clearing* the stream (`posix_trace_clear()`). This clears all the events that are now in the stream, but leaves its behavior (attributes) intact. Clearing the stream makes it exactly in the same state that it was just after being created.
- *Flushing* the stream (`posix_trace_flush()`). If the stream is created with a log, this action produces an automatic flushing of all the events which are now in the stream to the log. Otherwise, an error is returned.
- *Querying* the stream attributes (`posix_trace_get_attr()`) and the stream current status (`posix_trace_get_status()`). The stream status includes whether the stream is currently running or suspended, whether or not an overrun has occurred, etc.
- *Retrieving* the list of event types defined for the stream. The list is retrieved in order, since the function `posix_trace_eventtypelist_getnext_id()` returns the first event type when it is invoked for the first time, and the *next* event type in subsequent calls. At any time, the retrieval of event types can be initialized by calling `posix_trace_eventtypelist_rewind()`. Actually, the standard establishes that the event types are not actually associated with a particular stream, but to a particular *target* process. In other words, the list of event types is the same for all the streams which are tracing the same target.
- *Mapping* event names to event types (`posix_trace_trid_eventid_open()`). This is normally performed by the target process in order to create its own user event types. However, the trace controller process can use the mapping function in the opposite way: given a well-known user trace event *name*, the mapping function will return the event type identifier; then, the trace controller process can use that identifier to set up a stream filter, for example.

### 2.2.2 The Traced or Target Process

The *traced* or *target process* is the process that is being traced, that is, is the process for which a trace stream has been created and set up.

According to the standard, only two functions can actually be called from a target process:

a) A function to *register* a new user event type for this process (`posix_trace_eventid_open()`). The input argument of this function is the (new) event type *name*. If this name has already being registered for that target, then the previously mapped event type identifier is returned. If not, then a new identifier is internally associated with this name and returned. If an implementation-defined maximum amount of user event types had already been registered for that target process, then a predefined event type called `POSIX_TRACE_UNNAMED_USEREVENT` is returned. If successful, this registration is valid for all the streams that have been created, or *will* be created, to trace the target process (even if no stream has still been created for that target).

   From the user viewpoint, therefore, the identification of user event types is done in a per-name basis (instead of using integer values, for example). This allows for a name space wide enough to avoid collisions when independent pieces of instrumented code are linked together into a single application. This include, for example, the case of linking an instrumented third-party library to our code, even when we do not have the library's source code.

b) A function to trace an event (`posix_trace_event()`). This function has three input arguments: the event type, which must have been previously registered (see above), a pointer to any extra data that has to be stored along with the event, and the size of this data[4]. The event is stored in all the streams created for that particular target which are currently running and which do not have the event's type being filtered out.

It is important to point out that neither of these functions accepts a stream identifier as a parameter. That is, according to the standard philosophy, the target is programmed to invoke these functions without being aware (and independently) of actually being traced or not. The result is that calling the `posix_trace_event()` function has no effect if no stream has been created for the target. In other words, an instrumented running program does not actually *become* a target process until at least one stream has been created for it. The case of the `posix_trace_eventid_open()` function is different since, as explained above, the trace system will register any new event type for the program even when no stream has been created for tracing the process.

This philosophy completely decouples the target from the trace controller process, with many interesting advantages. For example, imagine an application that runs for long periods of time without stop (a real-time application or a database, for instance). It may be interesting to know, every once in a while, how this application is performing. Therefore, this (instrumented) application can be the target of an inspector (trace controller) program that, periodically, creates one or more streams to trace it, gets the resulting events, and then destroys the stream(s). Depending on the application characteristics, this occasional tracing may be good enough to check how the application is behaving, and does not overload the system with a continuous tracing.

[4] Internally, the trace system will store all this information along with some mandatory internal data defined by the standard (see Section 2.1.1). All this data will be reported when the event is retrieved by the trace analyzer process.

### 2.2.3 Trace Analyzer Process

This process is in charge of retrieving the stored events in order to analyze them. The standard defines three alternative retrieval functions to be used by the trace analyzer process:

a) `posix_trace_getnext_event()`. This function retrieves one event from the stream whose identifier is provided as a parameter. If no event is immediately available, the function blocks the invoking process (or thread) until an event is available.

b) `posix_trace_timedgetnext_event()`. This function works in a similar fashion than the previous one, but, when no event is immediately available, it blocks the process until either an event is available or an absolute timeout is reached (whatever of both happens first). If the timeout is produced first, the invoking process gets the corresponding error code.

c) `posix_trace_trygetnext_event()`. This function never blocks the invoking process: it either return a retrieved event or an error code, if no event is available at the moment.

If successful, any of these functions retrieve the oldest event stored in the stream which has not still been reported. The age of each event is calculated according to the automatic timestamp performed by the trace system when the event is recorded.

As explained above, the events can be only be retrieved from two different places: (1) from an active stream without log; (2) from the log of a (previously destroyed) stream with log, once this log has been opened into a (pre-recorded) trace stream. This defines the two kinds of analysis that the standard supports:

**a) On-line analysis**. In this kind of analysis, the trace analyzer process retrieves the events from an active trace stream (without log). As stated above, the retrieval function (any of them) needs to provide the stream's identifier; however, according to the standard, this identifier can only be used within the process that *created* the stream. This forces that, in an on-line analysis, the trace analyzer process and the trace controller process have to be the *same one*.

**b) Off-line analysis**. As explained in Section 2.1.2, this analysis is done in two steps: in the first step, events are recorded into an active trace stream with log that, automatically or under request of the trace controller process, flushes these events to the log (file). Once this step is over, the trace analyzer process *opens* the log into a private, pre-recorded stream (`posix_trace_open()`), from which it can start retrieving the events. Only the first of the three retrieval functions mentioned above can actually be used in a pre-recorded stream. Obviously, in this case, this function will never make the trace analyzer process to block, since all the events are already stored in the stream.

From a pre-recorded stream, events are always reported in order (according to the recording timestamp) but they are not erased from the stream after being retrieved. If necessary, the trace analyzer process can start retrieving the events again from the oldest one by rewinding the stream (`posix_trace_rewind()`), without having to re-open the log.

In addition, the trace analyzer process can also retrieve other information of the stream (either active or pre-recorded), including the list of registered event types and its names, the stream attribute object (and then each of its individual attributes), the stream current status (for an active stream), etc. All this information is intended to make the trace analyzer process able to correctly interpret the trace events which it is retrieving.

# 3 Levels of Support

The POSIX Trace standard splits the tracing support to be provided by the operating system in four different subsets or levels of support (named *implementation options*). Each of these levels can be optionally implemented by the system depending on the particular trace functionality that the system wants to support. Normally, each implementation option makes the trace system to support a certain subset of the tracing data types, constants, functions, etc.; occasionally, it also changes the semantics of some other functions supported by other options. Overall, this follows a general philosophy that POSIX has adopted in many system facilities, allowing for the implementation a great variety of conforming operating systems with very different levels of support.

In particular, the standard establishes four different implementation options, which are now discussed:

**a)** **Trace option**. This option correspond to the most basic level of trace support. It is not really optional, since any conforming system must provide, at least, all the system facilities included in this option. In particular, this option requires only tracing into streams without a log with no filtering of events, and it only permits one process to be traced per trace stream. In other words, this option only supports on-line tracing of single target processes without event filtering.

**b)** **Trace Log option**. This option adds the possibility of creating trace streams with log to the Trace option, hence allowing for off-line analysis.

**c)** **Trace Inheritance option**. This option adds the possibility of tracing multiple target processes in one stream to the Trace option. Having that this option is supported, the tracing of multiple target processes in the same stream can only occur in one scenario: if a target process *forks* into one or more child processes and the stream on which the (parent) target process is being traced was created with an specific inheritance attribute set. If so, all these processes will concurrently being traced in the same stream. Otherwise, children of a target process are not traced.

**d)** **Trace Event Filter option**. This option adds the possibility of filtering events to the Trace option. As explained above, the trace controller process can apply a set of event types as a filter on a particular stream, making any event whose type is in the set to be discarded (of that stream) when traced. Filtering of events is specially useful in on-line analysis, in order to permit the trace analyzer process to retrieve all the important events without loss. In general, filtering of not relevant events is always a good idea in order to prevent the tracing system to process an overwhelming number of events.

# 4 Review of the Trace Functions

This section briefly reviews the trace functions defined in the standard. In the following tables (Table 1 to 3), each function is cited along with the implementation options on which it is defined, the process or processes which may use it, and a brief description. In these tables, the following terminology has been used:

- In the column headers, **L**, **I**, and **F** stand for the three additional implementation options that can be supported, that is, Trace Log, Trace Inheritance and Trace Event Filter options. For each trace function (in rows), a tick ($\surd$) in one of these three columns means that the function belongs to this option (or, in other words, that the function has only to be provided if the tracing system wants to support that option).
- A plus sign (+) may optionally be placed instead of a tick ($\surd$) to associate a function with an implementation option (see above). This means that the function actually belongs to another option, but supporting *this* option makes the semantics of the function to change. For example, the function `posix_trace_attr_setstreamfullpolicy()` establishes the full policy in a stream attribute object. If only the Trace option is provided, this attribute can be set to one of two alternative values: `POSIX_TRACE_LOOP` and `POSIX_TRACE_UNTIL_FULL`. However, if the Trace Log option is also supported, then this attribute can also be set to a third alternative value (`POSIX_TRACE_FLUSH`).
- Inside tables, TC stands for Trace Controller Process, TP stands for Traced (or Target) Process, and TA stands for Trace Analyzer Process.

| Function | L | I | F | Processes | Description |
|---|---|---|---|---|---|
| posix_trace_attr_init | | | | TC | Initializes a trace stream attribute object. |
| posix_trace_attr_destroy | | | | TC | Destroys the trace stream attribute object. |
| posix_trace_attr_getenversion | | | | TC, TA | Gets the generation version of an attribute object (origin and version of the trace system). |
| posix_trace_attr_getname | | | | TC, TA | Gets the name of an attribute object (the name of the trace stream). |
| posix_trace_attr_setname | | | | TC | Sets the name of an attribute object (the name of the trace stream). |
| posix_trace_attr_getcreatetime | | | | TC, TA | Gets the creation time of the trace stream. |
| posix_trace_attr_getclockres | | | | TC, TA | Gets the resolution of the timestamping clock. |
| posix_trace_attr_getinherited | | √ | | TC, TA | Gets the inheritance policy of the trace stream (trace also children of the target process or not). |
| posix_trace_attr_setinherited | | √ | | TC | Sets the inheritance policy of the trace stream (trace also children of the target process or not). |
| posix_trace_attr_getstreamfullpolicy | + | | | TC, TA | Gets the trace stream full policy (loop, until full or flush). This latter only if Trace Log supported. |
| posix_trace_attr_setstreamfullpolicy | + | | | TC | Sets the trace stream full policy (loop, until full or flush). This latter only if Trace Log supported. |
| posix_trace_attr_getlogfullpolicy | √ | | | TC, TA | Gets the trace log full policy (loop, until full or append). |
| posix_trace_attr_setlogfullpolicy | √ | | | TC | Sets the trace log full policy (loop, until full or append). |
| posix_trace_attr_getmaxusereventsize | | | | TC, TA | Gets the maximum amount of memory to store a single user trace event. |
| posix_trace_attr_getmaxsystemeventsize | | | | TC, TA | Gets the maximum amount of memory to store a single system trace event. |
| posix_trace_attr_getmaxdatasize | | | | TC, TA | Gets the maximum size allowed of the data attached to a single trace event. |
| posix_trace_attr_setmaxdatasize | | | | TC | Sets the maximum size allowed of the data attached to a single trace event. |
| posix_trace_attr_getstreamsize | | | | TC, TA | Gets the minimum size of the trace stream used to store trace events. |
| posix_trace_attr_setstreamsize | | | | TC | Sets the minimum size of the trace stream used to store trace events. |
| posix_trace_attr_getlogsize | √ | | | TC, TA | Gets the minimum size of the trace log used to store trace events. |
| posix_trace_attr_setlogsize | √ | | | TC | Sets the minimum size of the trace log used to store trace events. |

**Table 1:** Trace functions defined in the POSIX Trace standard (i).

| Function | L | I | F | Processes | Description |
|---|---|---|---|---|---|
| posix_trace_create | | | | TC | Creates trace stream, using a stream attribute object. |
| posix_trace_create_withlog | √ | | | TC | Creates trace stream with a log, using a stream attribute object. |
| posix_trace_flush | √ | | | TC | Flushes the stream into its log. |
| posix_trace_shutdown | | | | TC | Stops the tracing in the stream and shuts the stream down. |
| posix_trace_clear | | | | TC | Reinitializes the stream, deleting all the events currently stored (if any). The stream status remains unchanged, as well as the mappings between trace event types and their names. |
| posix_trace_trid_eventid_open | | + | √ | TC | Returns the event type corresponding to an event name event type in a given stream. The mappings actually belong to the target process (or processes, if the Trace Inheritance option is supported). |
| posix_trace_eventid_get_name | | | | TC, TA | Gets the event name mapped with an event type in a given stream. |
| posix_trace_eventid_equal | | | | TC, TA | Compares two event types. |
| posix_trace_eventtypelist_getnext_id | | | | TC, TA | Gets the first/next event type defined for a given stream. |
| posix_trace_eventtypelist_rewind | | | | TC, TA | Rewinds the list of type events defined for a given stream. |
| posix_trace_eventset_empty | | | √ | TC | Returns an empty set of event types. |
| posix_trace_eventset_fill | | | √ | TC | Returns a set of event types which is full with all the event types, all the user event types or all the system event types. |
| posix_trace_eventset_add | | | √ | TC | Adds an event type to a set. |
| posix_trace_eventset_del | | | √ | TC | Deletes an event type from a set. |
| posix_trace_eventset_ismember | | | √ | TC | Tests whether an event type is a member of an event set. |
| posix_trace_get_filter | | | √ | TC | Retrieves the current filter from a given stream. |
| posix_trace_set_filter | | | √ | TC | Applies a filter to a given trace stream by using an event set. |
| posix_trace_start | | | | TC | Makes a given stream to start tracing. |
| posix_trace_stop | | | | TC | Stops a given trace stream. |
| posix_trace_eventid_open | | | | TP | Associates a trace event name with a trace event type. |
| posix_trace_event | | | | TP | Traces an event into all the streams created for the target on which the type of the event is not being filtered out. |

**Table 2:** Trace functions defined in the POSIX Trace standard (ii).

| Function | L | I | F | Processes | Description |
|---|---|---|---|---|---|
| posix_trace_open | √ | | | TA | Opens a log into a (pre-recorded) stream. |
| posix_trace_rewinds | √ | | | TA | Rewinds a (pre-recorded) stream. |
| posix_trace_close | √ | | | TA | Closes a (pre-recorded) stream. |
| posix_trace_get_attr | + | | | TC, TA | Gets the attribute object from a given active stream. (This can also be a pre-recorded stream, if the Trace Log option is supported). |
| posix_trace_get_status | + | | | TC, TA | Gets the current status of a given active stream. (This can also be a pre-recorded stream, if the Trace Log option is supported). |
| posix_trace_getnext_event | + | | | TA | Returns the oldest (non-retrieved) trace event from an active stream or blocks until such event is available. (The stream can also be a pre-recorded stream, if the Trace Log option is supported). |
| posix_trace_timedgetnext_event | | | | TA | Returns the oldest (non-retrieved) trace event from an active stream or blocks until either such event is available or a timeout is produced. |
| posix_trace_trygetnext_event | | | | TA | Returns the oldest (non-retrieved) trace event from an active stream or a significative value, but it never blocks. |

**Table 3:** Trace functions defined in the POSIX Trace standard (iii).