

Network Working Group  
Request for Comments: 2783  
Category: Informational

J. Mogul  
Compaq WRL  
D. Mills  
University of Delaware  
J. Brittonson  
Sun  
J. Stone  
Stanford  
U. Windl  
Universitaet Regensburg  
March 2000

## Pulse-Per-Second API for UNIX-like Operating Systems, Version 1.0

### Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

### Copyright Notice

Copyright (C) The Internet Society (2000). All Rights Reserved.

### Abstract

RFC 1589 describes a UNIX kernel implementation model for high-precision time-keeping. This model is meant for use in conjunction with the Network Time Protocol (NTP, RFC 1305), or similar time synchronization protocols. One aspect of this model is an accurate interface to the high-accuracy, one pulse-per-second (PPS) output typically available from precise time sources (such as a GPS or GOES receiver). RFC 1589 did not define an API for managing the PPS facility, leaving implementors without a portable means for using PPS sources. This document specifies such an API.

## Table of Contents

1	Introduction.....	2
2	Data types for representing timestamps.....	4
2.1	Resolution.....	4
2.2	Time scale.....	5
3	API.....	5
3.1	PPS abstraction.....	6
3.2	New data structures.....	7
3.3	Mode bit definitions.....	10
3.4	New functions.....	12
3.4.1	New functions: obtaining PPS sources.....	13
3.4.2	New functions: setting PPS parameters.....	14
3.4.3	New functions: access to PPS timestamps.....	16
3.4.4	New functions: disciplining the kernel timebase.....	18
3.5	Compliance rules.....	20
3.5.1	Functions.....	20
3.5.2	Mode bits.....	20
3.6	Examples.....	21
4	Security Considerations.....	24
5	Acknowledgements.....	24
6	References.....	25
7	Authors' Addresses.....	26
A.	Extensions and related APIs.....	27
A.1	Extension: Parameters for the "echo" mechanism.....	27
A.2	Extension: Obtaining information about external clocks.....	27
A.3	Extension: Finding a PPS source.....	28
B.	Example implementation: PPSDISC Line discipline.....	29
B.1	Example.....	29
C.	Available implementations.....	30
	Full Copyright Statement.....	31

## 1 Introduction

RFC 1589 [4] describes a model and programming interface for generic operating system software that manages the system clock and timer functions. The model provides improved accuracy and stability for most workstations and servers using the Network Time Protocol (NTP) [3] or similar time synchronization protocol. The model supports the use of external timing sources, such as the precision pulse-per-second (PPS) signals typically available from precise time sources (such as a GPS or GOES receiver).

However, RFC 1589 did not define an application programming interface (API) for the PPS facility. This document specifies such an interface, for use with UNIX (or UNIX-like) operating systems. Such systems often conform to the "Single UNIX Specification" [5], sometimes known as POSIX.

One convenient means to provide a PPS signal to a computer system is to connect that signal to a modem-control pin on a serial-line interface to the computer. The Data Carrier Detect (DCD) pin is frequently used for this purpose. Typically, the time-code output of the time source is transmitted to the computer over the same serial line. The computer detects a signal transition on the DCD pin, usually by receiving an interrupt, and records a timestamp as soon as possible.

Although existing practice has focussed on the use of serial lines and DCD transitions, PPS signals might also be delivered by other kinds of devices. The API specified in this document does not require the use of a serial line, although it may be somewhat biased in that direction.

The typical use of this facility is for the operating system to record ("capture") a high-resolution timestamp as soon as possible after it detects a PPS signal transition (usually indicated by an interrupt). This timestamp can then be made available, with less stringent delay constraints, to time-related software. The software can compare the captured timestamp to the received time-code to accurately discover the offset between the system clock and the precise time source.

The operating system may also deliver the PPS event to a kernel procedure, called the "in-kernel PPS consumer." One example would be the "hardpps()" procedure, described in RFC 1589, which is used to discipline the kernel's internal timebase.

The API specified in this document allows for one or more signal sources attached to a computer system to provide PPS inputs, at the option of user-level software. User-level software may obtain signal-transition timestamps for any of these PPS sources. User-level software may optionally specify at most one of these PPS sources to be used to discipline the system's internal timebase.

Although the primary purpose of this API is for capturing true pulse-per-second events, the API may also be used for accurately timestamping events of other periods, or even aperiodic events, when these can be expressed as signal transitions.

This document does not define internal details of how the API must be implemented, and does not specify constraints on the accuracy, resolution, or latency of the PPS feature. However, the utility of this feature is inversely proportional to the delay (and variance of delay), and implementors are encouraged to take this seriously.

In principle, the rate of events to be captured, or the frequency of the signals, can range from once per day (or less often) to several thousand per second. However, since in most implementations the timestamping function will be implemented as a processor interrupt at a relatively high priority, it is prudent to limit the rate of such events. This may be done either by mechanisms in the hardware that generates the signals, or by the operating system.

## 2 Data types for representing timestamps

Computer systems use various representations of time. Because this API is concerned with the provision of high-accuracy, high-resolution time information, the choice of representation is significant. (Here we consider only binary representations, not human-format representations.)

The two interesting questions are:

1. what is the resolution of the representation?
2. what time scale is represented?

These questions often lead to contentious arguments. Since this API is intended for use with NTP and POSIX-compliant systems, however, we can limit the choices to representations compatible with existing NTP and POSIX practice, even if that practice is considered "wrong" in some quarters.

### 2.1 Resolution

In the NTP protocol, "timestamps are represented as a 64-bit unsigned fixed-point number, in seconds relative to 0h on 1 January 1900. The integer part is in the first 32 bits and the fraction part in the last 32 bits [...] The precision of this representation is about 200 picoseconds" [3].

However, most computer systems cannot measure time to this resolution (this represents a clock rate of 5 GHz). The POSIX `gettimeofday()` function returns a "struct timeval" value, with a resolution of 1 microsecond. The POSIX `clock_gettime()` function returns a "struct timespec" value, with a resolution of 1 nanosecond.

This API uses an extensible representation, but defaults to the "struct timespec" representation.

## 2.2 Time scale

Several different time scales have been proposed for use in computer systems. UTC and TAI are the two obvious candidates.

Some people would prefer the use of TAI, which is identical to UTC except that it does not correct for leap seconds. Their preference for TAI stems from the difficulty of computing precise time differences when leap seconds are involved, especially when using times in the future (for which the exact number of leap seconds is, in general, unknowable).

However, POSIX and NTP both use UTC, albeit with different base dates. Given that support for TAI would, in general, require other changes to the POSIX specification, this API uses the POSIX base date of 00:00 January 1, 1970 UTC, and conforms to the POSIX use of the UTC time scale.

## 3 API

A PPS facility can be used in two different ways:

1. An application can obtain a timestamp, using the system's internal timebase, for the most recent PPS event.
2. The kernel may directly utilize PPS events to discipline its internal timebase, thereby providing highly accurate time to all applications.

This API supports both uses, individually or in combination. The timestamping feature may be used on any number of PPS sources simultaneously; the timebase-disciplining feature may be used with at most one PPS source.

Although the proper implementation of this API requires support from the kernel of a UNIX system, this document defines the API in terms of a set of library routines. This gives the implementor some freedom to divide the effort between kernel code and library code (different divisions might be appropriate on microkernels and monolithic kernels, for example).

### 3.1 PPS abstraction

A PPS signal consists of a series of pulses, each with an "asserted" (logical true) phase, and a "clear" (logical false) phase. The two phases may be of different lengths. The API may capture an "assert timestamp" at the moment of the transition into the asserted phase, and a "clear timestamp" at the moment of the transition into the clear phase.

The specific assignment of the logical values "true" and "false" with specific voltages of a PPS signal, if applicable, is outside the scope of this specification. However, these assignments SHOULD be consistent with applicable standards. Implementors of PPS sources SHOULD document these assignments.

Reminder to implementors of DCD-based PPS support: TTL and RS-232C (V.24/V.28) interfaces both define the "true" state as the one having the highest positive voltage. TTL defines a nominal absence of voltage as the "false" state, but RS-232C (V.24/V.28) defines the "false" state by the presence of a negative voltage.

The API supports the direct provision of PPS events (and timestamps) to an in-kernel PPS consumer. This could be the function called "hardpps()", as described in RFC 1589 [4], but the API does not require the kernel implementation to use that function name internally. The current version of the API supports at most one in-kernel PPS consumer, and does not provide a way to explicitly name it. The implementation SHOULD impose access controls on the use of this feature.

The API optionally supports an "echo" feature, in which events on the incoming PPS signal may be reflected through software, after the capture of the corresponding timestamp, to an output signal pin. This feature may be used to discover an upper bound on the actual delay between the edges of the PPS signal and the capture of the timestamps; such information may be useful in precise calibration of the system.

The designation of an output pin for the echo signal, and sense and shape of the output transition, is outside the scope of this specification, but SHOULD be documented for each implementation. The output pin MAY also undergo transitions at other times besides those caused by PPS input events.

Note: this allows an implementation of the echo feature to generate an output pulse per input pulse, or an output edge per input pulse, or an output pulse per input edge. It also allows the same signal pin to be used for several purposes simultaneously.

Also, the API optionally provides an application with the ability to specify an offset value to be applied to captured timestamps. This can be used to correct for cable and/or radio-wave propagation delays, or to compensate for systematic jitter in the external signal. The implementation SHOULD impose access controls on the use of this feature.

### 3.2 New data structures

The data structure declarations and symbol definitions for this API will appear in the header file `<sys/timepps.h>`. The header file MUST define all constants described in this specification, even if they are not supported by the implementation.

The API includes several implementation-specific types:

```
typedef ... pps_handle_t;          /* represents a PPS source */

typedef unsigned ... pps_seq_t; /* sequence number */
```

The "pps\_handle\_t" type is an opaque scalar type used to represent a PPS source within the API.

The "pps\_seq\_t" type is an unsigned integer data type of at least 32 bits.

The precise declaration of the pps\_handle\_t and pps\_seq\_t types is system-dependent.

The API imports the standard POSIX definition for this data type:

```
struct timespec {
    time_t  tv_sec;          /* seconds */
    long    tv_nsec;        /* nanoseconds */
};
```

The API defines this structure as an internal (not "on the wire") representation of the NTP "64-bit unsigned fixed-point" timestamp format [3]:

```
typedef struct ntp_fp {
    unsigned int    integral;
    unsigned int    fractional;
} ntp_fp_t;
```

The two fields in this structure may be declared as any unsigned integral type, each of at least 32 bits.

The API defines this new union as an extensible type for representing times:

```
typedef union pps_timeu {
    struct timespec tspec;
    ntp_fp_t        ntpfp;
    unsigned long    longpad[3];
} pps_timeu_t;
```

Future revisions of this specification may add more fields to this union.

Note: adding a field to this union that is larger than 3\*sizeof(long) will break binary compatibility.

The API defines these new data structures:

```
typedef struct {
    pps_seq_t    assert_sequence;        /* assert event seq # */
    pps_seq_t    clear_sequence;        /* clear event seq # */
    pps_timeu_t  assert_tu;
    pps_timeu_t  clear_tu;
    int          current_mode;           /* current mode bits */
} pps_info_t;

#define assert_timestamp    assert_tu.tspec
#define clear_timestamp    clear_tu.tspec

#define assert_timestamp_ntpfp  assert_tu.ntpfp
#define clear_timestamp_ntpfp  clear_tu.ntpfp

typedef struct {
    int          api_version;            /* API version # */
    int          mode;                   /* mode bits */
    pps_timeu_t  assert_off_tu;
    pps_timeu_t  clear_off_tu;
} pps_params_t;

#define assert_offset    assert_off_tu.tspec
#define clear_offset    clear_off_tu.tspec

#define assert_offset_ntpfp    assert_off_tu.ntpfp
#define clear_offset_ntpfp    clear_off_tu.ntpfp
```

The "pps\_info\_t" type is returned on an inquiry to PPS source. It contains the timestamps for the most recent assert event, and the most recent clear event. The order in which these events were actually received is defined by the timestamps, not by any other



aspect of the specification. Each timestamp field represents the value of the operating system's internal timebase when the timestamped event occurred, or as close as possible to that time (with the optional addition of a specified offset). The `current_mode` field contains the value of the mode bits (see section 3.3) at the time of the most recent transition was captured for this PPS source. An application can use `current_mode` to discover the format of the timestamps returned.

The `assert_sequence` number increases once per captured assert timestamp. Its initial value is undefined. If incremented past the largest value for the type, the next value is zero. The `clear_sequence` number increases once per captured clear timestamp. Its initial value is undefined, and may be different from the initial value of `assert_sequence`. If incremented past the largest value for the type, the next value is zero. Due to possible signal loss or excessive signal noise, the `assert-sequence` number and the `clear-sequence` number might not always increase in step with each other.

Note that these sequence numbers are most useful in applications where events other than PPS transitions are to be captured, which might be involved in a precision stopwatch application, for example. In such cases, the sequence numbers may be used to detect overruns, where the application has missed one or more events. They may also be used to detect an excessive event rate, or to detect that an event has failed to occur between two calls to the `time_pps_fetch()` function (defined later).

In order to obtain an uninterrupted series of sequence numbers (and hence of event timestamps), it may be necessary to sample the `pps_info_t` values at a rate somewhat faster than the underlying event rate. For example, an application interested in both `assert` and `clear` timestamps may need to sample at least twice per second. Proper use of the sequence numbers allows an application to discover if it has missed any event timestamps due to an insufficient sampling rate.

The `pps_params_t` data type is used to discover and modify parameters of a PPS source. The data type includes a mode field, described in section 3.3. It also includes an `api_version` field, a read-only value giving the version of the API. Currently, the only defined value is:

```
#define PPS_API_VERS_1 1
```

This field is present to enable binary compatibility with future versions of the API.

Note: the term "read-only" in this specification means that an application cannot modify the relevant data item; only the implementation can modify the value. The implementation MUST ignore attempts by the application to modify a read-only field.

As an OPTIONAL feature of the API, the implementation MAY support adding offsets to the timestamps that are captured. (Values of type "struct timespec" can represent negative offsets.) The `assert_offset` field of a `pps_params_t` value specifies a value to be added to generate a captured `assert_timestamp`. The `clear_offset` of a `pps_params_t` value field specifies a value to be added to generate a captured `clear_timestamp`. Since the offsets, if any, apply to all users of a given PPS source, the implementation SHOULD impose access controls on the use of this feature; for example, allowing only the super-user to set the offset values. The default value for both offsets is zero.

### 3.3 Mode bit definitions

A set of mode bits is associated with each PPS source.

The bits in the mode field of the `pps_params_t` type are:

```
/* Device/implementation parameters */
#define PPS_CAPTUREASSERT      0x01
#define PPS_CAPTURECLEAR      0x02
#define PPS_CAPTUREBOTH       0x03

#define PPS_OFFSETASSERT      0x10
#define PPS_OFFSETCLEAR      0x20

#define PPS_CANWAIT           0x100
#define PPS_CANPOLL           0x200

/* Kernel actions */
#define PPS_ECHOASSERT        0x40
#define PPS_ECHOCLEAR        0x80

/* Timestamp formats */
#define PPS_TSFMT_TSPEC       0x1000
#define PPS_TSFMT_NTPFP      0x2000
```

These mode bits are divided into three categories:

1. Device/implementation parameters: These are parameters either of the device or of the implementation. If the implementation allows these to be changed, then these bits are read/write for users with sufficient privilege (such as the super-user), and

read-only for other users. If the implementation does not allow these bits to be changed, they are read-only.

2. Kernel actions: These bits specify certain kernel actions to be taken on arrival of a signal. If the implementation supports one of these actions, then the corresponding bit is read/write for users with sufficient privilege (such as the super-user), and read-only for other users. If the implementation does not support the action, the corresponding bit is always zero.
3. Timestamp formats: These bits indicate the set of timestamp formats available for the device. They are always read-only.

In more detail, the meanings of the Device/implementation parameter mode bits are:

PPS\_CAPTUREASSERT

If this bit is set, the assert timestamp for the associated PPS source will be captured.

PPS\_CAPTURECLEAR

If this bit is set, the clear timestamp for the associated PPS source will be captured.

PPS\_CAPTUREBOTH Defined as the union of PPS\_CAPTUREASSERT and PPS\_CAPTURECLEAR, for convenience.

PPS\_OFFSETASSERT

If set, the assert\_offset value is added to the current value of the operating system's internal timebase in order to generate the captured assert\_timestamp.

PPS\_OFFSETCLEAR If set, the clear\_offset value is added to the current value of the operating system's internal timebase in order to generate the captured clear\_timestamp.

PPS\_CANWAIT If set, the application may request that the time\_pps\_fetch() function (see section 3.4.3) should block until the next timestamp arrives. Note: this mode bit is read-only.

PPS\_CANPOLL This bit is reserved for future use. An application SHOULD NOT depend on any functionality implied either by its presence or by its absence.

If neither PPS\_CAPTUREASSERT nor PPS\_CAPTURECLEAR is set, no valid timestamp will be available via the API.

The meanings of the Kernel action mode bits are:

PPS\_ECHOASSERT    If set, after the capture of an assert timestamp, the implementation generates a signal transition as rapidly as possible on an output signal pin. This MUST NOT affect the delay between the PPS source's transition to the asserted phase and the capture of the assert timestamp.

PPS\_ECHOCLEAR    If set, after the capture of a clear timestamp, the implementation generates a signal transition as rapidly as possible on an output signal pin. This MUST NOT affect the delay between the PPS source's transition to the clear phase and the capture of the clear timestamp.

The timestamp formats are:

PPS\_TS\_FMT\_TSPEC    Timestamps and offsets are represented as values of type "struct timespec". All implementations MUST support this format, and this format is the default unless an application specifies otherwise.

PPS\_TS\_FMT\_NTPFP    Timestamps and offsets are represented as values of type "ntp\_fp\_t", which corresponds to the NTP "64-bit unsigned fixed-point" timestamp format [3]. Support for this format is OPTIONAL.

Other timestamp format bits may be defined as fields are added to the "pps\_timeu\_t" union.

The operating system may implement all of these mode bits, or just a subset of them. If an attempt is made to set an unsupported mode bit, the API will return an error. If an attempt is made to modify a read-only mode bit, the API will return an error.

### 3.4 New functions

In the description of functions that follows, we use the following function parameters:

filedes            A file descriptor (type: int), for a serial line or other source of PPS events.

<code>ppshandle</code>	A variable of type <code>"pps_handle_t"</code> , as defined in section 3.2.
<code>ppsinfobuf</code>	A record of type <code>"pps_info_t"</code> , as defined in section 3.2.
<code>ppsparams</code>	A record of type <code>"pps_params_t"</code> , as defined in section 3.2.
<code>tsformat</code>	An integer with exactly one of the timestamp format bits set.

#### 3.4.1 New functions: obtaining PPS sources

The API includes functions to create and destroy PPS source "handles".

##### SYNOPSIS

```
int time_pps_create(int filedes, pps_handle_t *handle);
int time_pps_destroy(pps_handle_t handle);
```

##### DESCRIPTION

All of the other functions in the PPS API operate on PPS handles (type: `pps_handle_t`). The `time_pps_create()` is used to convert an already-open UNIX file descriptor, for an appropriate special file, into a PPS handle.

The definition of what special files are appropriate for use with the PPS API is outside the scope of this specification, and may vary based on both operating system implementation, and local system configuration. One typical case is a serial line, whose DCD pin is connected to a source of PPS events.

The mode in which the UNIX file descriptor was originally opened affects what operations are allowed on the PPS handle. The `time_pps_setparams()` and `time_pps_kcbind()` functions (see sections 3.4.2 and 3.4.4) SHOULD be prohibited by the implementation if the descriptor is open only for reading (`O_RDONLY`).

Note: operations on a descriptor opened with an inappropriate mode might fail with `EBADF`.

The `time_pps_destroy()` function makes the PPS handle unusable, and frees any storage that might have been allocated for it. It does not close the associated file descriptor, nor does it change any of the parameter settings for the PPS source.

Note: If this API is adapted to an operating system that does not follow UNIX conventions for representing an accessible PPS source as an integer file descriptor, the `time_pps_create()` function may take different parameters from those shown here.

#### RETURN VALUES

On successful completion, the `time_pps_create()` function returns 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

If called with a valid handle parameter, the `time_pps_destroy()` function returns 0. Otherwise, it returns -1.

#### ERRORS

If the `time_pps_create()` function fails, `errno` may be set to one of the following values:

- |              |   |
|--------------|---|
| [EBADF]      | The <code>filedes</code> parameter is not a valid file descriptor.                        |
| [EOPNOTSUPP] | The use of the PPS API is not supported for the file descriptor.                          |
| [EPERM]      | The process's effective user ID does not have the required privileges to use the PPS API. |

#### 3.4.2 New functions: setting PPS parameters

The API includes several functions use to set or obtain the parameters of a PPS source.

#### SYNOPSIS

```
int time_pps_setparams(pps_handle_t handle,
                      const pps_params_t *ppsparms);
int time_pps_getparams(pps_handle_t handle,
                      pps_params_t *ppsparms);
int time_pps_getcap(pps_handle_t handle, int *mode);
```

#### DESCRIPTION

A suitably privileged application may use `time_pps_setparams()` to set the parameters (mode bits and timestamp offsets) for a PPS source. The `pps_params_t` type is defined in section 3.2; mode bits are defined in section 3.3. An application may use `time_pps_getparams()` to discover the current settings of the PPS parameters. An application that needs to change only a subset of the existing

parameters must first call `time_pps_getparams()` to obtain the current parameter values, then set the new values using `time_pps_setparams()`.

Note: a call to `time_pps_setparams()` replaces the current values of all mode bits with those specified via the `ppsparams` argument, except those bits whose state cannot be changed. Bits might be read-only due to access controls, or because they are fixed by the implementation.

The timestamp format of the `assert_offset` and `clear_offset` fields is defined by the mode field. That is, on a call to `time_pps_setparams()`, the kernel interprets the supplied offset values using the timestamp format given in the mode field of the `ppsparams` argument. If the requested timestamp format is not supported, the `time_pps_setparams()` function has no effect and returns an error value. On a call to `time_pps_getparams()`, the kernel provides the timestamp format of the offsets by setting one of the timestamp format bits in the mode field.

Note: an application that uses `time_pps_getparams()` to read the current offset values cannot specify which format is used. The implementation SHOULD return the offsets using the same timestamp format as was used when the offsets were set.

An application wishing to discover which mode bits it may set, with its current effective user ID, may call `time_pps_getcap()`. This function returns the set of mode bits that may be set by the application, without generating an `EINVAL` or `EPERM` error, for the specified PPS source. It does not return the current values for the mode bits. A call to `time_pps_getcap()` returns the mode bits corresponding to all supported timestamp formats.

The `time_pps_getcap()` function MAY ignore the mode in which the associated UNIX file descriptor was opened, so the application might still receive an `EBADF` error on a call to `time_pps_setparams()`, even if `time_pps_getcap()` says that the chosen mode bits are allowed.

The mode bits returned by `time_pps_getcap()` for distinct PPS handles may differ, reflecting the specific capabilities of the underlying hardware connection to the PPS source, or of the source itself.

#### RETURN VALUES

On successful completion, the `time_pps_setparams()`, `time_pps_getparams()`, and `time_pps_getcap()` functions return 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

## ERRORS

If the `time_pps_setparams()`, `time_pps_getparams()`, or `time_pps_getcap()` function fails, `errno` may be set to one of the following values:

- |              |  |
|--------------|--|
| [EBADF]      | The handle parameter is not associated with a valid file descriptor, or the descriptor is not open for writing.          |
| [EFAULT]     | A parameter points to an invalid address.  |
| [EOPNOTSUPP] | The use of the PPS API is not supported for the associated file descriptor.  |
| [EINVAL]     | The operating system does not support all of the requested mode bits.  |
| [EPERM]      | The process's effective user ID does not have the required privileges to use the PPS API, or to set the given mode bits. |

## 3.4.3 New functions: access to PPS timestamps

The API includes one function that gives applications access to PPS timestamps. As an implementation option, the application may request the API to block until the next timestamp is captured. (The API does not directly support the use of the `select()` or `poll()` system calls to wait for PPS events.)

## SYNOPSIS

```
int time_pps_fetch(pps_handle_t handle,
                  const int tsformat,
                  pps_info_t *ppsinfobuf,
                  const struct timespec *timeout);
```

## DESCRIPTION

An application may use `time_pps_fetch()` to obtain the most recent timestamps captured for the PPS source specified by the handle parameter. The `tsformat` parameter specifies the desired timestamp format; if the requested timestamp format is not supported, the call fails and returns an error value. The application **MUST** specify exactly one timestamp format.



This function blocks until either a timestamp is captured from the PPS source, or until the specified timeout duration has expired. If the timeout parameter is a NULL pointer, the function simply blocks until a timestamp is captured. If the timeout parameter specifies a delay of zero, the function returns immediately.

Support for blocking behavior is an implementation option. If the PPS\_CANWAIT mode bit is clear, and the timeout parameter is either NULL or points to a non-zero value, the function returns an EOPNOTSUPP error. An application can discover whether the feature is implemented by using `time_pps_getcap()` to see if the PPS\_CANWAIT mode bit is set.

The result is stored in the `ppsinfobuf` parameter, whose fields are defined in section 3.2. If the function returns as the result of a timeout or error, the contents of the `ppsinfobuf` are undefined.

If this function is invoked before the system has captured a timestamp for the signal source, the `ppsinfobuf` returned will have its timestamp fields set to the time format's base date (e.g., for PPS\_TSFMT\_TSPEC, both the `tv_sec` and `tv_nsec` fields will be zero).

#### RETURN VALUES

On successful completion, the `time_pps_fetch()` function returns 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

#### ERRORS

If the `time_pps_fetch()` function fails, `errno` may be set to one of the following values:

[EBADF]	The handle parameter is not associated with a valid file descriptor.
[EFAULT]	A parameter points to an invalid address.
[EINTR]	A signal was delivered before the time limit specified by the timeout parameter expired and before a timestamp has been captured.
[EINVAL]	The requested timestamp format is not supported.
[EOPNOTSUPP]	The use of the PPS API is not supported for the associated file descriptor.
[ETIMEDOUT]	The timeout duration has expired.

#### 3.4.4 New functions: disciplining the kernel timebase

The API includes one OPTIONAL function to specify if and how a PPS source is provided to a kernel consumer of PPS events, such as the code used to discipline the operating system's internal timebase.

##### SYNOPSIS

```
int time_pps_kcbind(pps_handle_t handle,
                    const int kernel_consumer,
                    const int edge,
                    const int tsformat);
```

##### DESCRIPTION

An application with appropriate privileges may use `time_pps_kcbind()` to bind a kernel consumer to the PPS source specified by the handle.

The kernel consumer is identified by the `kernel_consumer` parameter. In the current version of the API, the possible values for this parameter are:

```
#define PPS_KC_HARDPPS      0
#define PPS_KC_HARDPPS_PLL  1
#define PPS_KC_HARDPPS_FLL  2
```

with these meanings:

`PPS_KC_HARDPPS` The kernel's `hardpps()` function (or equivalent).

`PPS_KC_HARDPPS_PLL` A variant of `hardpps()` constrained to use a phase-locked loop.

`PPS_KC_HARDPPS_FLL` A variant of `hardpps()` constrained to use a frequency-locked loop.

Implementation of any or all of these values is OPTIONAL.

The `edge` parameter indicates which edge of the PPS signal causes a timestamp to be delivered to the kernel consumer. It may have the value `PPS_CAPTUREASSERT`, `PPS_CAPTURECLEAR`, or `PPS_CAPTUREBOTH`, depending on particular characteristics of the PPS source. It may also be zero, which removes any binding between the PPS source and the kernel consumer.

The `tsformat` parameter specifies the format for the timestamps delivered to the kernel consumer. If this value is zero, the implementation MAY choose the appropriate format, or return `EINVAL`. The implementation MAY ignore a non-zero value for this parameter.

The binding created by this call persists until it is changed by a subsequent call specifying the same `kernel_consumer`. In particular, a subsequent call to `time_pps_destroy()` for the specified handle does not affect the binding.

The binding is independent of any prior or subsequent changes to the `PPS_CAPTUREASSERT` and `PPS_CAPTURECLEAR` mode bits for the device. However, if either the edge or the `tsformat` parameter values are inconsistent with the capabilities of the PPS source, an error is returned. The implementation MAY also return an error if the `tsformat` value is unsupported for `time_pps_kcbind()`, even if it is supported for other uses of the API.

The operating system may enforce two restrictions on the bindings created by `time_pps_kcbind()`:

1. the kernel MAY return an error if an attempt is made to bind a kernel consumer to more than one PPS source a time.
2. the kernel MAY restrict the ability to set bindings to processes with sufficient privileges to modify the system's internal timebase. (On UNIX systems, such modification is normally done using `settimeofday()` and/or `adjtime()`, and is restricted to users with superuser privilege.)

Warning: If this feature is configured for a PPS source that does not have an accurate 1-pulse-per-second signal, or is otherwise inappropriately configured, use of this feature may result in seriously incorrect timekeeping for the entire system. For best results, the 1-PPS signal should have much better frequency stability than the system's internal clock source (usually a crystal-controlled oscillator), and should have jitter (variation in interarrival time) much less than the system's clock-tick interval.

See RFC 1589 [4] for more information about how the system's timebase may be disciplined using a PPS signal.

#### RETURN VALUES

On successful completion, the `time_pps_kcbind()` function returns 0. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

## ERRORS

If the `time_pps_kcbind()` function fails, `errno` may be set to one of the following values:

[EBADF]	The handle parameter is not associated with a valid file descriptor, or the descriptor is not open for writing.
[EFAULT]	A parameter points to an invalid address.
[EINVAL]	The requested timestamp format is not supported.
[EOPNOTSUPP]	The use of the PPS API is not supported for the associated file descriptor, or this OPTIONAL function is not supported.
[EPERM]	The process's effective user ID does not have the required privileges to set the binding.

### 3.5 Compliance rules

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

Some features of this specification are OPTIONAL, but others are REQUIRED.

#### 3.5.1 Functions

An implementation MUST provide these functions:

- `time_pps_create()`
- `time_pps_destroy()`
- `time_pps_setparams()`
- `time_pps_getparams()`
- `time_pps_getcap()`
- `time_pps_fetch()`

An implementation MUST provide this function, but it may be implemented as a function that always return an EOPNOTSUPP error, possibly on a per-source basis:

- `time_pps_kcbind()`

### 3.5.2 Mode bits

An implementation MUST support at least one of these mode bits for each PPS source:

- PPS\_CAPTUREASSERT
- PPS\_CAPTURECLEAR

and MAY support both of them. If an implementation supports both of these bits for a PPS source, it SHOULD allow them to be set simultaneously.

An implementation MUST support this timestamp format:

- PPS\_TSFMT\_TSPEC

An implementation MAY support these mode bits:

- PPS\_ECHOASSERT
- PPS\_ECHOCLEAR
- PPS\_OFFSETASSERT
- PPS\_OFFSETCLEAR

An implementation MAY support this timestamp format:

- PPS\_TSFMT\_NTPFP

### 3.6 Examples

A very simple use of this API might be:

```
int fd;
pps_handle_t handle;
pps_params_t params;
pps_info_t infobuf;
struct timespec timeout;

/* Open a file descriptor and enable PPS on rising edges */
fd = open(PPSfilename, O_RDWR, 0);
time_pps_create(fd, &handle);
time_pps_getparams(handle, &params);
if ((params.mode & PPS_CAPTUREASSERT) == 0) {
    fprintf(stderr, "%s cannot currently CAPTUREASSERT\n",
            PPSfilename);
    exit(1);
}

/* create a zero-valued timeout */
```

```

timeout.tv_sec = 0;
timeout.tv_nsec = 0;

/* loop, printing the most recent timestamp every second or so */
while (1) {
    sleep(1);
    time_pps_fetch(handle, PPS_TSFMT_TSPEC, &infobuf, &timeout);
    printf("Assert timestamp: %d.%09d, sequence: %ld\n",
           infobuf.assert_timestamp.tv_sec,
           infobuf.assert_timestamp.tv_nsec,
           infobuf.assert_sequence);
}

```

Note that this example omits most of the error-checking that would be expected in a reliable program.

Also note that, on a system that supports PPS\_CANWAIT, the function of these lines:

```

sleep(1);
time_pps_fetch(handle, PPS_TSFMT_TSPEC, &infobuf, &timeout);

```

might be more reliably accomplished using:

```

timeout.tv_sec = 100;
timeout.tv_nsec = 0;
time_pps_fetch(handle, PPS_TSFMT_TSPEC, &infobuf, &timeout);

```

The (arbitrary) timeout value is used to protect against the possibility that another application might disable PPS timestamps, or that the hardware generating the timestamps might fail.

A slightly more elaborate use of this API might be:

```

int fd;
pps_handle_t handle;
pps_params_t params;
pps_info_t infobuf;
int avail_mode;
struct timespec timeout;

/* Open a file descriptor */
fd = open(PPSfilename, O_RDWR, 0);
time_pps_create(fd, &handle);

/*
 * Find out what features are supported
 */

```

```
time_pps_getcap(handle, &avail_mode);
if ((avail_mode & PPS_CAPTUREASSERT) == 0) {
    fprintf(stderr, "%s cannot CAPTUREASSERT\n", PPSfilename);
    exit(1);
}
if ((avail_mode & PPS_OFFSETASSERT) == 0) {
    fprintf(stderr, "%s cannot OFFSETASSERT\n", PPSfilename);
    exit(1);
}

/*
 * Capture assert timestamps, and
 * compensate for a 675 nsec propagation delay
 */

time_pps_getparams(handle, &params);
params.assert_offset.tv_sec = 0;
params.assert_offset.tv_nsec = 675;
params.mode |= PPS_CAPTUREASSERT | PPS_OFFSETASSERT;
time_pps_setparams(handle, &params);

/* create a zero-valued timeout */
timeout.tv_sec = 0;
timeout.tv_nsec = 0;

/* loop, printing the most recent timestamp every second or so */
while (1) {
    if (avail_mode & PPS_CANWAIT) {
        time_pps_fetch(handle, PPS_TSFMT_TSPEC, &infobuf, NULL);
        /* waits for the next event */
    } else {
        sleep(1);
        time_pps_fetch(handle, PPS_TSFMT_TSPEC, &infobuf,
            timeout);
    }
    printf("Assert timestamp: %d.%09d, sequence: %ld\n",
        infobuf.assert_timestamp.tv_sec,
        infobuf.assert_timestamp.tv_nsec,
        infobuf.assert_sequence);
}
```

Again, most of the necessary error-checking has been omitted from this example.

## 4 Security Considerations

This API gives applications three capabilities:

- Causing the system to capture timestamps on certain events.
- Obtaining timestamps for certain events.
- Affecting the system's internal timebase.

The first capability should not affect security directly, but might cause a slight increase in interrupt latency and interrupt-handling overhead.

The second capability might be useful in implementing certain kinds of covert communication channels.

In most cases, neither of these first two issues is a significant security threat, because the traditional UNIX file protection facility may be used to limit access to the relevant special files. Provision of the PPS API adds minimal additional risk.

The final capability is reserved to highly privileged users. In UNIX systems, this means those with superuser privilege. Such users can evade protections based on file permissions; however, such users can in general cause unbounded havoc, and can set the internal timebase (and its rate of change), so this API creates no new vulnerabilities.

## 5 Acknowledgements

The API in this document draws some of its inspiration from the LBL "ppsclock" distribution [2], originally implemented in 1993 by Steve McCanne, Craig Leres, and Van Jacobson. We also thank Poul-Henning Kamp, Craig Leres, Judah Levine, and Harlan Stenn for helpful comments they contributed during the drafting of this document.



## 6 References

1. Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
2. Steve McCanne, Craig Leres, and Van Jacobson. PPSCLOCK.  
<ftp://ftp.ee.lbl.gov/ppsclock.tar.Z>.
3. Mills, D., "Network Time Protocol (Version 3): Specification, Implementation and Analysis", RFC 1305, March 1992.
4. Mills, D., "A Kernel Model for Precision Timekeeping", RFC 1589, March, 1994.
5. The Open Group. The Single UNIX Specification, Version 2 - 6 Vol Set for UNIX 98. Document number T912, The Open Group, February, 1997.

## 7 Authors' Addresses

Jeffrey C. Mogul  
Western Research Laboratory  
Compaq Computer Corporation  
250 University Avenue  
Palo Alto, California, 94305, U.S.A.

Phone: 1 650 617 3304 (email preferred)  
EMail: mogul@wrl.dec.com

David L. Mills  
Electrical and Computer Engineering Department  
University of Delaware  
Newark, DE 19716

Phone: (302) 831-8247  
EMail: mills@udel.edu

Jan Brittenson  
Sun Microsystems, Inc.  
901 San Antonio Rd M/S MPK17-202  
Palo Alto, CA 94303  
Email: Jan.Brittenson@Eng.Sun.COM

Jonathan Stone  
Stanford Distributed Systems Group  
Stanford, CA 94305

Phone: (650) 723-2513  
EMail: jonathan@dsg.stanford.edu

Ulrich Windl  
Universitaet Regensburg, Klinikum

EMail: ulrich.windl@rz.uni-regensburg.de

## A. Extensions and related APIs

The API specified in the main body of this document could be more useful with the provision of several extensions or companion APIs.

At present, the interfaces listed in this appendix are not part of the formal specification in this document.

### A.1 Extension: Parameters for the "echo" mechanism

The "echo" mechanism described in the body of this specification leaves most of the details to the implementor, especially the designation of one or more output pins.

It might be useful to extend this API to provide either or both of these features:

- A means by which the application can discover which output pin is echoing the input pin.
- A means by which the application can select which output pin is echoing the input pin.

### A.2 Extension: Obtaining information about external clocks

The PPS API may be useful with a wide variety of reference clocks, connected via several different interface technologies (including serial lines, parallel interfaces, and bus-level interfaces). These reference clocks can have many features and parameters, some of which might not even have been invented yet.

We believe that it would be useful to have a mechanism by which an application can discover arbitrary features and parameters of a reference clock. These might include:

- Clock manufacturer, model number, and revision level
- Whether the clock is synchronized to an absolute standard
- For synchronized clocks,
  - \* The specific standard
  - \* The accuracy of the standard
  - \* The path used (direct connection, shortwave, longwave, satellite, etc.)
  - \* The distance (offset) and variability of this path

- For PPS sources,
  - \* The pulse rate
  - \* The pulse shape
  - \* Which edge of the pulse corresponds to the epoch
- The time representation format

This information might best be provided by an API analogous to the standard "curses" API, with a database analogous to the standard "terminfo" database. That is, a "clockinfo" database would contain a set of (attribute, value) pairs for each type of clock, and the API would provide a means to query this database.

Additional mechanisms would allow an application to discover the clock or clocks connected to the local system, and to discover the clockinfo type of a specific clock device.

### A.3 Extension: Finding a PPS source

Although the clockinfo database described in section A.2, together with the discover mechanisms described there, would allow an application to discover the PPS source (or sources) connected to a system, it might be more complex than necessary.

A simpler approach would be to support a single function that provides the identity of one or more PPS sources.

For example, the function might be declared as

```
int time_pps_findsource(int index,
                        char *path, int pathlen,
                        char *idstring, int idlen);
```

The index argument implicitly sets up an ordering on the PPS sources attached to the system. An application would use this function to inquire about the Nth source. The function would return -1 if no such source exists; otherwise, it would return 0, and would place the pathname of the associated special file in the path argument. It would also place an identification string in the idstring argument. The identification string could include the clock make, model, version, etc., which could then be used by the application to control its behavior.

This function might simply read the Nth line from a simple database, containing lines such as:

```
/dev/tty00      "TrueTime 468-DC"
/dev/pps1       "Homebrew rubidium frequency standard"
```

allowing the system administrator to describe the configuration of PPS sources.

## B. Example implementation: PPSDISC Line discipline

One possible implementation of the PPS API might be to define a new "line discipline" and then map the API onto a set of `ioctl()` commands. Here we sketch such an implementation; note that this is not part of the specification of the API, and applications should not expect this low-level interface to be available.

In this approach, the set of line disciplines is augmented with one new line discipline, PPSDISC. This discipline will act exactly the same as the TTYDISC discipline, except for its handling of modem DCD interrupts.

Once the `TIOCSETD` `ioctl()` has been used to select this line discipline, PPS-related operations on the serial line may be invoked using new `ioctl()` commands. For example (values used only for illustration):

```
#define PPSFETCH      _IOR('t', 75, pps_info_t)
#define PPSSETPARAM  _IOW('t', 76, pps_params_t)
#define PPSGETPARAM  _IOR('t', 77, pps_params_t)
#define PPSGETCAP    _IOR('t', 78, int)
```

### B.1 Example

A typical use might be:

```
int ldisc = PPSDISC;
pps_params_t params;
pps_info_t infobuf;

ioctl(fd, TIOCSETD, &ldisc);    /* set discipline */

/*
 * Check the capabilities of this PPS source to see
 * if it supports what we need.
 */
ioctl(fd, PPSGETCAP, &params);
if ((params.mode & PPS_CAPTUREASSERT) == 0) {
    fprintf(stderr, "PPS source is not suitable\n");
    exit(1);
}

/*
 * Set this line to timestamp on a rising-edge interrupt
```

```
    */
    ioctl(fd, PPSGETPARAMS, &params);
    params.mode |= PPS_CAPTUREASSERT;
    ioctl(fd, PPSSETPARAMS, &params);

    sleep(2);          /* allow time for the PPS pulse to happen */

    /* obtain most recent timestamp and sequence # for this line */
    ioctl(fd, PPSFETCH, &infobuf);
```

Again, this example imprudently omits any error-checking.

#### C. Available implementations

Several available implementations of this API are listed at <http://www.ntp.org/ppsapi/PPSImpList.html>. Note that not all of these implementations correspond to the current version of the specification.

## Full Copyright Statement

Copyright (C) The Internet Society (2000). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

## Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

