



OPEN

Compute Project

Precise Time API

Revision 1.0

Authors:

Georgi Chalakov, gchalakov@nvidia.com

Ahmad Byagowi, clk@meta.com

Elad Wind, elad@nvidia.com

Table of Contents

License	3
OCP CLA	3
Acknowledgements	4
Compliance with OCP Tenets	4
Openness	4
Efficiency	4
Impact	4
2.4. Scale	4
Change Log	5
Background	6
Problem Statement	6
Requirements	6
Scenarios	6
Scenario A – Order past events chronologically	7
Scenario B – Guarantee order of operations	7
Scenario C – Single way delay	7
Scenario D – Coexist with established time API	7
Design Goals	7
Design Overview	8
Programming Language	8
Error Handling	8
Instant	8
Get now	9
Zero Instant	9
Equal Instants	9
Happened Before	10
Duration	10
Conversions to Unix time	11
Time Context	12
Time Context Properties	12
Design Alternatives	14
True Time API	14
References	15
Appendix A - Precise Time Application Programming Interface	16
Appendix B – Examples	18

License

OCF CLA

Contributions to this Specification are made under the terms and conditions set forth in Open Compute Project Contribution License Agreement (“OCF CLA”) (“Contribution License”) by:

NVIDIA, Meta

You can review the Contributor License(s) for this Specification on the OCP website at <https://www.opencompute.org/legal-documents>. For actual executed copies of either agreement, please contact OCP directly.

Usage of this Specification is governed by the terms and conditions set forth in **[select one:] Open Web Foundation Modified Final Specification Agreement (“OWFa 1.0”) or Open Compute Project Hardware License – Permissive (“OCPHL Permissive”) or Open Compute Project Hardware License – Reciprocal (“OCPHL Reciprocal”)** also known as a “Specification License”.

Notes:

- 1) The above license does not apply to the Appendix or Appendices. The information in the Appendix or Appendices is for reference only and non-normative in nature.

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY LIABILITY FOR LOST PROFITS OR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT, SPECIAL OR PUNITIVE DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING

NEGLIGENCE), OR OTHERWISE, AND EVEN IF OCP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Acknowledgements

The Contributors of this Specification would like to acknowledge the following companies for their feedback:

Meta
NVIDIA

Compliance with OCP Tenets

Openness

Precise Time API fosters collaboration, interoperability, innovation, and market expansion of Time-Aware applications in the ecosystem by enabling seamless integration, sharing functionalities, and empowering third-party developers to build upon existing platforms and services.

Efficiency

Precise Time API reduces R&D and integration efforts by saving time and resources that would otherwise be spent on building everything from scratch.

Impact

Precise Time API will achieve impact across the OCP ecosystem by allowing developers access to agreed mature APIs for time services across switches, servers, appliances and domains.

Scale

The availability of an open API promotes interoperability, collaboration, and innovation, making the software more versatile and appealing to a wider range of users and developers

2.5. Sustainability

Adoption of time aware applications tends to improve overall efficiency of distributed applications, driving lower consumption of compute, storage and networking components, and positively impacting DC utilization.

Change Log

Date	Version #	Author	Description
05/23/2023	1.0	Georgi Chalakov	Submission for OCP IC Approval
06/01/2022	0.3	Georgi Chalakov	Typos; Shared with OCP
05/16/2022	0.2	Georgi Chalakov	Remove explicit context prop
04/04/2022	0.1	Georgi Chalakov	Initial version

Background

The concept of temporal ordering of events pervades our thinking about systems [10]. When we use physical clocks to order events, if event A happened at an earlier time than event B, the timestamp of A should be lower than the timestamp of B. Unfortunately, even with accurate and precise physical clocks, this is not always possible. We can only order events when the distance between timestamps exceeds the maximum error of the measurement. To order events correctly, we must account for the measurement error.

In existing time APIs [1], time is encoded as an offset from an epoch. The epoch in Unix is Jan 1st, 1970 [2], and in Windows, it is Jan 1st, 1601 [3]. In both cases, time is represented as an exact value with no information about the error that may have been introduced by the time measurement and synchronization processes.

Today, time management technologies [4][5] not only expose the time as an offset from an epoch but also the error between the reference clock and the synchronized clock. This error represents the time uncertainty at the synchronized clock, which is never zero. It is a result of accumulated imprecisions in the network equipment, uncertainties in the OS scheduler, and differences in the supported libraries.

Problem Statement

Precise time is represented as an offset from an epoch and is accompanied by an upper bound of accumulated error. Unfortunately, many time APIs ignore the error, making it difficult to confidently order events based on their timestamps.

To make effective use of precise time in applications and services, we need a better API that exposes the clock time, the uncertainty window, and data from the synchronization process to the application. This information is critical for accurately ordering events and improving the reliability of time-sensitive systems.

Requirements

Scenarios

This is the list of scenarios that Precise time API will be evaluated against. The proposed API must allow a simple implementation that is easy to make it right, and difficult to make it wrong.

Scenario A – Order past events chronologically

This scenario is about system behavior reasoning in the case of failure or unexpected results. To analyze past system behavior, we collect events from different machines in a common place.

Precise event timestamping allows us to order events that occur close to each other. This requires precise timestamping of the events, transporting both the events and timestamps, and ordering the events based on their timestamps.

Scenario B – Guarantee order of operations

We can guarantee the order of two operations on different machines if we wait long enough or exchange messages between machines. With precise time, we can minimize the wait and avoid communication altogether. However, the wait time for each operation affects not only the latency of that operation but also, in most cases where the operations depend on each other, the total throughput of the system.

Scenario C – Single way delay

Usually, we use half the round-trip time to measure one-way delay. Precise time allows us to measure one-way delays by calculating the duration between the timestamps at the beginning and end of a process. The duration between the two events can be measured with the precision of the time technology.

Scenario D – Coexist with established time API

Existing time APIs provide functionality for formatting, displaying, and converting between different time zones. The client should be able to convert time from existing API's both to and from Precise Time API.

Design Goals

The list of goals (order is not based on the priority):

1. The API is a read-only API.
2. The API is easy to use correctly and difficult to use incorrectly.
3. The API coexists and cooperates with existing time APIs.
4. The API encourages implementations to choose the best internal representation.
5. The API is extensible. In the future more data and functionality can be exposed to the clients without breaking existing clients.

Design Overview

Programming Language

We define the API in terms of modern C language. In a different programming language, the API would provide the same functionality with the syntax that is native for the language. For example, the error in Rust is returned as `Result<...>`. In the Go version of the API, the error is the second returned value.

Error Handling

Functions that may fail return an error code. When a function succeeds, the return value is 0. The exact error values are implementation specific. When a function returns error code different from success, any other returned values are undefined.

Instant

We can start introducing the new API by capturing the main difference between the existing time APIs and the **Precise Time API**: the error of the measurement. Following the example of some modern programming languages [6][7], we introduce a new type: **Instant**.

A value of type **Instant** captures the following attributes: a point of time, the unit of time measurement, the error of the measurement, and the unit of the error measurement. For example, 37 hours from the epoch with error measurement of 1 minute is `Instant(37, hours, 1, minutes)`.

An implementation may choose different internal representations. For example, the instant can be represented as a time interval or as a partial calculation. The representation is not exposed to the client and the implementation has complete freedom to choose the best representation.

```
// Point of time, the error, and the corresponding time units
// Values of type Instant must be Copyable and Movable
struct Instant;
```

Requirements for a value of type **Instant** are Copyable and Movable. The memory layout may vary between different implementations or different versions of the same implementation.

Get now

An **Instant** can be initialized with the current time by calling the function `now()`. The second parameter of `now()` is the time context. Time context defines the source of time. There might be multiple time contexts in the same application. For example, NTP, System, and PTP time contexts.

```
// Source of time context
struct TimeContext;
```

```
// If the return value is 0, the instant tm is initialized with
// the time point immediately after the functions now() returns
int now(Instant* tm, const TimeContext*);
```

The time stored in the instant corresponds to the time point after the return of the function. For example, in the following case, the time tm is going to correspond to time t_0 .

```
t-1:   int err = now(&tm, &context);
t0:   if (err) {
t1:       ...
```

Zero Instant

There is a special value that represents the zero **Instant**. Zero instant is not necessarily the epoch time point. Zero instant is the smallest possible instant that an implementation supports. The implementation is encouraged, but not required to initialize new **Instant** to Zero.

```
// The Zero Instant
Instant Zero;

// Return true if the Instant is the Zero.
bool IsZero(const Instant*);
```

Equal Instants

The internal representation may vary between different instants, especially if they are initialized with the time from different time contexts.

A comparison between two time points or errors should take into consideration the error from the internal representations. When the difference is below the precision of the measurement, the values are equal.

```
// *t0 is equal to *t1, iff returned value is 0 and *result is true.
int equal(const Instant* t0, const Instant* t1, bool* result);
```

Happened Before

The result of **before()** is true, if and only if, the t_0 time point plus t_0 error is strictly before the t_1 time point minus the t_1 error. The comparison between two Instants may fail if any of the pointers is null or the content of the instant is invalid.

```
// *t0 happened before *t1, iff returned value is 0 and *result is true.
int before(const Instant* t0, const Instant* t1, bool* result);
```

Duration

```
// Duration between two Instants
// Values of type Duration must be Copyable and Movable
struct Duration;
```

A value of type Duration represents the distance between two points of time, the error, and the unit of the measurement. A Duration is negative when the points of time are ordered reverse chronologically.

The implementations are encouraged but not required to initialize new instances of Duration to Zero Duration.

```
// The Zero Duration
Duration Zero;

// Return true if the Duration is Zero.
bool IsZero(const Duration*);
```

Duration values can be compared, negated, added, and multiplied by a floating-point number.

```
// *result is such that *lhs + result = Zero, if returned value is 0
int neg(const Duration* d, Duration* result);

// *result is *d0 + *d1, if the returned value is 0
int add(const Duration* d0, const Duration* d1, Duration* result);

// *result is c * *lhs, if the returned value is 0
int mul(const Duration* lhs, double c, Duration* result);
```

A comparison between two Durations should consider the measurement of the precision. If the difference is below the precision, the durations are considered equal.

```
// *d0 is equal to *d1, iff returned value is 0 and *result is true.
int equal(const Duration* d0, const Duration* d1, bool* result);
```

```
// iff *d0 value + *d0 error is strictly less than *d1 value - *d1 error
int less(const Duration* d0, const Duration* d1, bool* result);
```

Conversions to Unix time

Precise Time API need functionality for:

- String representation that includes rich text formatting for printing and logging.
- Serialization and deserialization for storing and transmitting the values.
- Conversion from and to standard OS time API.

The most common representation that works across OS and languages is the standard C struct `timespec` [8]. The representation is a pair of numbers: seconds from the epoch Jan 1, 1970, and number of nanoseconds within the second. The nanosecond value is an offset within the second in the range [0, 999999999]

```
#include <time.h>

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

Converting an Instant and Duration to **struct timespec** gives us access to a large set of time related functions. It also allows API clients to store and transport the values in a well-defined format.

```
// conversion from timespec to Instant/Duration
int to_timespec(const Instant* instant, timespec* t, timespec* error);
int to_timespec(const Duration* duration, timespec* t, timespec* error);

// conversion from Instant/Duration to time point and error
int from_timespec(const timespec* t, const timespec* error, Instant* res);
int from_timespec(timespec* t, timespec* error, Duration* res);
```

Time Context

```
// The time context where now() is executed.
struct TimeContext;
```

The time context represents the hardware and the software that is used for time measurement and synchronization. An application may create multiple time contexts and use them concurrently.

The internal representation of a **TimeContext** is implementation specific and may change between different versions of the implementation. The client cannot access the internal representation.

Precise Time API implementation may provide a few implementations of **TimeContext**, for example PTP, NTP and **SystemClock** time context. When the time context is NULL, the implementation should use the **SystemClock** time context. The system clock time context should be present in every implementation.

In addition to the standard time contexts – **SystemClock**, NTP, PTP, an implementation may provide custom time context for unique environments and hardware.

```
// Create system clock[8] (realtime, thread, monotonic) time context.
int create_context_system(clockid_t clk_id, TimeContext* result);

// Create NTP clock time context.
int create_context_ntp(TimeContext* result);

// Create PTP clock time context.
int create_context_ptp(const char* device, TimeContext* result);

// Or, implementation specific time context.
int create_context_(void* p, TimeContext* result);
```

Time Context Properties

Time contexts properties are a set of values that describe the capabilities of the implementation and synchronization parameters.

A time context may expose more of the internal implementation through a call to **get_property()**. The property names and value types are implementation specific. For example, “clock-type”, “oscillator-temperature-in-celsius”, and others.

```
int get_property(const TimeContext*, const char* name, void* value, int size);
```

Design Alternatives

True Time API

```
struct window_of_uncertainty {  
    int64 earliest_ts;  
    int64 latest_ts;  
}  
  
int get_time(window_of_uncertainty* tm);
```

Pros:

- Simple and straightforward
- The client has total freedom to use and interpret the returned values.

Cons:

- It is easy to use it incorrectly. The client must create correct implementation for comparison, calculation, interpretation of corner cases and any operations over the returned values.
- Expose the internal representation and by that limits the implementation to a pair of values. For example, the internal representation may use floating point numbers or may never calculate the exact values for ordering and the operations.
- Does not provide a simple and correct way of connecting with the existing time API. Operations as conversion, printing, logging, serialization are expected to come from the client.

References

1. C17 standard (ISO/IEC 9899:2018):
<https://en.cppreference.com/w/c/chrono>
2. Linux man page - Time
<https://linux.die.net/man/2/time>
3. Win32 API FILETIME
<https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-filetime>
4. IEEE 1588-2008, IEEE 1588-2019
https://en.wikipedia.org/wiki/Precision_Time_Protocol
5. Hardware timestamping
<https://docs.nvidia.com/networking/display/MLNXOFEDv541030/Time-Stamping>
6. Go - Time library
<https://pkg.go.dev/time>
7. Rust – Temporal quantification
<https://doc.rust-lang.org/std/time/index.html>
8. clock_gettime(3) – Linux man page
https://linux.die.net/man/3/clock_gettime
9. How to Design a Good API and Why it Matters, Joshua Bloch
<https://storage.googleapis.com/pub-tools-public-publication-data/pdf/32713.pdf>
10. [Time, Clocks, and the Ordering of Events in a Distributed System](#). Leslie Lamport, 1978.

Appendix A - Precise Time Application Programming Interface

```
// include timespec, clockid
#include <time.h>

// Instant represents a point of time, the error, and the corresponding units.
// Values of type Instant must be Copyable and Movable
struct Instant;

// The context where the time API is executed.
// Values of type TimeContext must be Movable
struct TimeContext;

// If the return value is 0, the instant tm is initialized with
// the time point that is immediately after the function returns
int now(Instant* tm, const TimeContext*);

// The Zero value Instant
Instant Zero;

// Return true if the Instant is Zero instant.
bool IsZero(const Instant*);

// *lhs is equal to *rhs, iff returned value is 0 and *res is true.
int equal(const Instant* lhs, const Instant* rhs, bool* result);

// *lhs happened before *rhs, iff returned value is 0 and *res is true.
int before(const Instant* lhs, const Instant* rhs, bool* result);

// Duration between two Instants.
// Values of type Duration must be Copyable and Movable
struct Duration;

// The content in *res is valid iff the returned value is 0
int add(const Instant* instant, const Duration* Duration, Instant* result);

// The Zero value Duration
Duration Zero;

// Return true if the Duration is Zero.
bool IsZero(const Duration*);

// Returns Duration such as *lhs + Duration = Zero, if returned value is 0
int negate(const Duration* lhs, Duration* result);

// Returns *lhs + *rhs, if the returned value is 0
int add(const Duration* lhs, const Duration* rhs, Duration* result);

// Returns *lhs * c, if the returned value is 0
int mul(const Duration* lhs, double c, Duration* result);
```

```

// *lhs is equal to *rhs, iff returned value is 0 and *res is true.
int equal(const Duration* lhs, const Duration* rhs, bool* result);

// iff *lhs value + *lhs error is strictly less than *rhs value - *rhs error
int less(const Duration* lhs, const Duration* rhs, bool* result);

// conversion from timespec to Instant/Duration
int to_timespec(const Instant* instant, timespec* tm, timespec* error);
int to_timespec(const Duration* duration, timespec* tm, timespec* error);

// conversion from Instant/Duration to time point and error
int from_timespec(const timespec* tm, const timespec* error, Instant* result);
int from_timespec(timespec* tm, timespec* error, Duration* result);

// Time API execution context
struct TimeContext;

// Get internal TimeContext property - clock-type, temperatur, others
int get_property(const TimeContext*, const char* name, void* value, int size);

// Create system (realtime, monotonic) time context as it is defined in [8].
int create_context_system(clockid_t clk_id, TimeContext*);

// Create NTP clock time context.
int create_context_ntp(TimeContext*);

// Create PTP clock time context.
int create_context_ptp(const char* device, TimeContext*);

```

Appendix B – Examples

Scenario A - Order past events chronologically

Scenario B – Guarantee order of operations

Scenario C – Single way delay

Scenario D – Coexist with existing time API.