ISO/IEC JTC 1/SC 22/WG14 **N2402**

June 30, 2019 **v 1**

Modernize time.h functions

It's about time

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

The interfaces in `time.h` are inconsistent, partially underspecified, subject to undetectable overflow, not reentrant, not thread-safe and present security issues. Along the lines of ISO 9945 (POSIX), we propose to modernize the interfaces to avoid these problems.

Contents

1. INTRODUCTION

1.1. Problem description

The interfaces in `time.h` to manipulate time values have grown mostly unattended over the years and present several problems that could be easily avoided with more modern, redesigned interfaces. The main problems are as follows:

(1) The function **clock** is subject to undetectable overflows and has non-standard semantics on one of the major legacy implementations.
(2) The function **time** does not specify the encoding that is used in the **time_t** type and the resolution of this time has no query interface. There is only the function **difftime** to obtain the relative difference between two time measurements, but there is no interface to know about the granularity that we can expect.
(3) The function **timespec_get** has a resolution for which there is no query interface. Its use of **time_t** is not necessarily consistent with the use of the same type by **time**.
(4) It is not specified, if the function **timespec_get** when called with **TIME_UTC** is sensible to changes of the system clock or not. (Besides that, **TIME_UTC** is a misnomer.)
(5) The standard allows implementations to add more time bases than **TIME_UTC** but gives no guidance in which direction to go with such new base values.
(6) The return value specification of **timespec_get** does not allow to dissociate different types of errors.
(7) The function **asctime** has undefined behavior when it is called with time values that are out-of-bounds. Since the output format and length for this function is prescribed to the byte, this is an unnecessary loop hole that can easily be fixed.
(8) The function **ctime** has implicit undefined behavior when it is called with an argument that is not presentable as local time.
(9) The functions **asctime**, **ctime**, **gmtime** and **localtime** refer to static state and are thus neither reentrant nor thread-safe. In addition, giving write access to a static variable in the program state provides an exploitable attack vector for buffer overflow attacks.

1.2. Strategy

Most of these problems have already been addressed by ISO 9945 (POSIX) (after which the most recent addition of **timespec_get** has been modeled) so we propose a simple and straight forward solution: adopt and adapt the interfaces from there as far as possible.
We **do not** propose to promote interfaces of Annex K to Clause 7.27, because this would in turn introduce two new major problems:

(1) Annex K interfaces need the infrastructure of constraint handlers. We don't think that the attempt to repair `time.h` justifies us to force implementations to introduce new infrastructure.
(2) Because of that globally shared infrastructure, Annex K interfaces are inherently thread-unsafe.

In addition, the runtime constraints that are covered for the corresponding time functions in Annex K are mostly size constraints of the arrays that are provided as arguments. Such constraints can be expressed in the syntax, and can, for many cases, be detected at compile time. Therefore, in accordance with the C2x charter, we propose to update the existing interfaces syntactically such that they are more friendly to static analysis.
Because the problems and solutions are much intertwined, we propose all these additions in this single paper. For those in WG14 that prefer to have smaller bits and pieces to swallow, we have divided the paper into sections that all have their specific questions (potential straw polls for the committee) at the end, such that WG14 may cherry-pick modifications to the standard as seems fit.

In case of adoption of any of the new functions[1] or the changes to the return value of **timespec_get** we also propose to add a feature test macro `__STDC_VERSION_TIME_H__`.

Question 0. *Shall a feature test macro* `__STDC_VERSION_TIME_H__` *as proposed in N2402 be added to 7.27.1?*

2. PUT THE BOUNDS INTO THE INTERFACES

According to the C2x charter,[2] we should aim that interfaces specify the constraints on the corresponding functions as thorough as possible:

> 15. **Application Programming Interfaces (APIs) should be self-documenting when possible.** In particular, the order of parameters in function declarations should be arranged such that the size of an array appears before the array. The purpose is to allow Variable-Length Array (VLA) notation to be used. This not only makes the code's purpose clearer to human readers, but also makes static analysis easier. Any new APIs added to the Standard should take this into consideration.

In the case of `time.h` we can already apply this strategy to the existing interfaces to make them analysis friendly. *E.g* the function

```
time_t mktime(struct tm *timeptr);
```

can be expressed with the equivalent prototype

```
time_t mktime (struct tm timeptr [static 1]);
```

that emphasizes on the fact that the function expects a non-null pointer to at least one element as an argument.
Such changes (adding **static** bound constraints) can be added to most of the existing functions in `time.h`. Since these interfaces are rewritten during compilation, such additions are always compatible with existing code.
On the other hand, coding such restrictions into header files allows compilers to diagnose the most flagrant violations of the requirements. In particular, compilers can track pointers they know to be null, and may then diagnose an invalid call to a `time.h` function.
There is only one of the functions that should not be rewritten with a `[static]` parameter instead of a pointer, the **time** function. Here, it is specified that a null pointer is a valid argument.
All these changes are straight forward and we refer to the annex for the concrete formulation of that change.

Question 1. *Shall we adopt* `[static]` *array parameter specifications for the* `time.h` *header as proposed in N2402?*

Question 2. *If yes, does WG14 want to see a proposal that changes pointer parameters of library functions to* `[static]` *array parameters?*

3. MAKE THE UB OF `CTIME` EXPLICIT

In C17, the call `ctime(timer)` is declared to be functionally equivalent to

[1]Proposed are `asctime_r`, `ctime_r`, `localtime_r`, `gmtime_r`, and `timespec_getres`.
[2]See `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2086.htm`

```
        asctime(localtime(timer))
```

This definition has one surprising nob: `localtime` has conditions under which it returns a null pointer, but `asctime` is not allowed to receive such a pointer. Such a return happens, if `timer` cannot be interpreted as local time (for whatever reasons). Thus, `ctime` is implicitly undefined in that situation.
We see two possibilities to improve that situation:

(1) Make this behavior explicitly undefined; or
(2) widen the specification of `asctime` and impose that it should accept a null pointer and then also would return a null pointer to propagate the error return.

The second approach would only make programs that previously have been (implicitly) undefined, defined. But it would also impose implementations to change their code.
We propose to make that undefined behavior explicit by reformulating a phrase in 7.27.3.2 p2:

> The calendar time specified by `timer[0]` shall be convertible to local time.

Question 3. *Shall we add a phrase that spells out the undefined behavior of* `ctime` *as proposed in N2402?*

Question 4. *If not, shall we change* `asctime` *such that it accepts null pointer arguments?*

3.1. Avoid UB for `asctime`

The `asctime` function writes a textual representation of a broken down time in to a `char` buffer. Due to the exact output format, the buffer size that is needed is exactly 26. In our proposal (7.27.3.1) we use that fact in two places:

— We change the use of `sprintf` in the operational specification to `snprintf` with a second argument set to 26.
— Later, we augment the syntactical specification of the newly introduced `asctime_r`.

The first has the advantage that we then can guarantee that the input buffer will never be accessed out-of-bounds, and that we can get rid of undefined behavior for invalid time values. We propose to replace

> ~~If any of the members of the broken-down time contain values that are outside their normal ranges,the behavior of the `asctime` functions is undefined. Likewise, if the calculated year exceeds four digits or is less than the year 1000, the behavior is undefined.~~

by

> The return value points to a zero terminated string of length at most 25 and no write beyond the 26^{th} byte occurs. If any of the members of the broken-down time contain values that are outside their normal ranges, or if the calculated year exceeds four digits or is less than the year 1000, the contents of the returned string is otherwise unspecified. .

Question 5. *Shall we change the specification of* `asctime` *to use* `snprintf` *and modify the undefined behavior to unspecified behavior as proposed in N2402?*

4. MAKE THE RETURN OF THE CONVERSION FUNCTIONS `CONST` QUALIFIED

The functions **`asctime`**, **`ctime`**, **`gmtime`** and **`localtime`** return pointers to static state and are thus neither reentrant nor thread-safe. In addition, exposing a modifiable static variable in the program state provides an exploitable attack vector for buffer overflow attacks. The corresponding return values of the functions are not even **`const`** qualified, but they cannot not be put in a read-only section, anyhow, because their contents is subject to change with different calls.
So, changing the return types of these functions will not help against malicious overflow attacks but at least it could prevent these buffers from accidental overwrites.

QUESTION 6. *Shall the return types of functions* **`asctime`**, **`ctime`**, **`gmtime`** *and* **`localtime`** *be changed to pointer-to* **`const`** *qualified types as as proposed in N2402?*

5. ADD CONVERSION FUNCTIONS THAT ARE REENTRANT AND THREAD-SAFE

ISO 9945 has four simple replacement functions for the conversion functionalities. They are suffixed with `_r` and just add a pointer to a buffer that is also returned to the parameters. Provided that any of these functions is integrated we propose to add an explicit requirement as a new first paragraph of 7.27.3:

> Functions with a `_r` suffix shall be reentrant and shall not be subject to data races. Unless specified otherwise, they place the result of the conversion into the buffer referred by `buf` and return that pointer.

5.1. The `asctime_r` function

We augment the syntactical specification of the newly introduced `asctime_r` from ISO 9945 by an explicit requirement about the size of the buffer by using `[static restrict 26]`. In ISO 9945 there is already text that requires this size for the buffer, but there is not syntax to make this detectable.

QUESTION 7. *Shall we adopt* `asctime_r` *as proposed in N2402?*

5.2. The `localtime_r` function

A `localtime_r` function can be specified easily. To simplify the text, we propose that the operational definition is directly given in terms of `asctime_r`, see 7.27.3.1.

QUESTION 8. *Shall we adopt* `localtime_r` *as proposed in N2402?*

5.3. The `ctime_r` function

To add the `ctime_r` function we have to specify in addition how the chaining of `localtime_r` and `asctime_r` is to be performed. For 2.27.3.2 we propose

```
        asctime_r(localtime_r(timer, (struct tm[1]){ 0 }), buf)
```

That is, the implicit creation of an otherwise unaccessible temporary object that is used to transfer the broken down time.

QUESTION 9. *Shall we adopt* `ctime_r` *as proposed in N2402?*

5.4. The `gmtime_r` function

QUESTION 10. *Shall we adopt* `gmtime_r` *as proposed in N2402?*

6. ADD NEW OPTIONAL TIME BASES

C11 and C17 left the addition of new time bases completely to the implementation. Although it is a good principle to leave room for extensions, certain of them already have a connotation in other normative context. In particular, ISO 9945 already provides specifications for four different time bases, two for elapsed time measurement, and two for CPU time.

We propose to add optional macros for these time bases to the standard, such that the names, if defined, bind implementations to a particular semantic. ISO 9945 and ISO 9899 differ slightly in their interfaces, we propose to have macros were we replace a `CLOCK` prefix by `TIME` for of the four different clocks defined in ISO 9945. Since these will be generally different from the values provided by ISO 9945 (there the constants have type `clockid_t`) we can impose that the corresponding values are positive without invalidating components of ISO 9945.

> Time bases other than **`TIME_UTC`** are optional; all time bases evaluate to values greater than 0.

6.1. Elapsed time

ISO 9945 has two different “clocks” for measurement of elapsed time, `CLOCK_REALTIME` and `CLOCK_MONOTONIC`. They differ eventually in the starting point of the measurement (*epoch* vs. boot time) and, more importantly, concerning their behavior when the system time is set:

— `CLOCK_REALTIME` changes when the clock is set to a new value, *e.g* if a background time daemon adjusts to a drift indicated by a time servers, or if calendar time is adjusted with a leap second. This is the only clock in ISO 9945 that is mandatory, and as such plays a similar role as **`TIME_UTC`** for ISO 9899.

— `CLOCK_MONOTONIC` is guaranteed not to be affected by such changes of the system clock and to measure physical time as perceived by the platform.

Which of these two (if any) would best to model the behavior of current C implementations when using **`TIME_UTC`** could be subject to debade. We propose not to go into such discussion, but to leave such details to the implementations.

The addition of the macros `TIME_REALTIME` and `TIME_MONOTONIC` is straight forward, see 7.27.1 p2. We then propose the additional text in 7.27.2.5 p3:

> If `base` is `TIME_REALTIME` the behavior is the similar, only that the result is affected by implementation-defined functions that set the system time, if any. For `TIME_MONOTONIC` the reference point may or may not be the same epoch or any other implementation-defined time point. This point shall not change during the program execution and the result shall not be affected by any implementation-defined functions that set the system time, if any. If such functions are provided, it is implementation-defined if **`TIME_UTC`** behaves like the first or the second.

QUESTION 11. *Shall we adopt* `TIME_REALTIME` *as proposed in N2402?*

QUESTION 12. *Shall we adopt* `TIME_MONOTONIC` *as proposed in N2402?*

QUESTION 13. *Shall we relate* **`TIME_UTC`** *to the new optional time bases as proposed in N2402?*

6.2. CPU time

In C17, CPU time of a program execution can be measured by means of the **`clock`** function. Unfortunately this functions has several problems, the most sever being that it may overflow

without notice. Another disadvantage of **clock** is that there is one legacy C implementation that gets this function fundamentally wrong when compared to the C standard: it accounts for elapsed (wallclock) time instead of CPU time. This repeatably leads to confusion when code is ported from or to conforming platforms. For these reasons we think that **clock** is best deprecated and replaced by an appropriate time base for **timespec_get**.
ISO 9945 has two such "clocks" which we propose to adapt to the needs of the C standard. Because implementations might need to dynamically distinguish different values for these bases for concurrent program executions (processes) or threads, the specifications of the values exempts them from being compile time constants and we add in 7.25.1 p3:

> The value of TIME_PROCESS_CPUTIME_ID shall be different from the above and shall not change during the same program execution. The macro TIME_THREAD_CPUTIME_ID shall not be defined if the implementation does not support threads; its value shall be different from the above, shall be the same for all invocations from the same thread, and the value provided for one thread shall not be used by a different thread as base argument of **timespec_get**.

For **timespec_get** itself the text proposal in 7.27.2.5 is then quite simple:

> For base set to TIME_PROCESS_CPUTIME_ID and TIME_THREAD_CPUTIME_ID the result is similar, but the call measures the amount of processor time associated with the program execution or thread, respectively.

Calls with TIME_PROCESS_CPUTIME_ID could replace calls of **clock**, provided we knew the resolution of this time base.
Calls with TIME_THREAD_CPUTIME_ID would implement a new feature that allows to distinguish the cost of threads individually.

QUESTION 14. *Shall we adopt* TIME_PROCESS_CPUTIME_ID *as proposed in N2402?*

QUESTION 15. *Shall we adopt* TIME_THREAD_CPUTIME_ID *as proposed in N2402?*

7. ADD AN INTERFACE TO QUERY RESOLUTION OF TIME BASES

Already for **TIME_UTC**, C17 has no interface that would allow to query the resolution of the resulting time. If on the long run we want to replace **clock** with **timespec_get**() we have to ensure that we also have a tool that provides a functionality similar to CLOCKS_PER_SECOND. Because of the genericity of **timespec_get**, the interface to query resolutions should not be a series of macros:

— User functions may have a time base as a parameter, so they cannot decided at compile time which resolution would be to query.
— The resolution may not be part of the platform ABI but be dependent of a particular version of the CPU or operating system.
— The resolution for a specific time base should not change during program execution. Therefore performance critical code can easily cache these values at program startup or thread startup if they need to.

ISO 9945 has a function that is capable to capture resolutions of predefined bases (which could probably be done with a macro) and also of all implementation-defined bases, the clock_getres function.
We propose to model such a function, timespec_getres, accordingly. The specification is straight forward and can be inspected in the appendix, see the new clause 7.27.2.6 (and also an crossreference for **timespec_get** in 7.27.2.5 p2).

QUESTION 16. *Shall we add function* timespec_getres *as proposed in N2402?*

8. DEFINE THE RETURN OF `TIMESPEC_GETRES` (AND/OR `TIMESPEC_GET`) IF CALLED WITH A NON-SUPPORTED BASE

As much as the resolution of a particular time base may not be part of the platform ABI, the whole support of such a base may be subject to conditions that can only be detected at runtime. The easiest way to deal with such situations is to provide well-specified error returns to functions that use time bases.
ISO 9945 uses **`errno`** to distinguish different error returns, and in particular a value **`EINVAL`** for this particular error. We propose to stay with the same error codes, which implies to add **`EINVAL`** to `errno.h` (7.5 p2).

Question 17. *Shall we add the return value* **`-EINVAL`** *to* `timespec_getres` *if it is called with a non-supported base as proposed in N2402?*

9. ALLOW `TIMESPEC_GET` TO DIFFERENTIATE ERROR RETURNS

The **`timespec_get`** can fail for different reasons, such as an invalid time base or an overflow. Currently, such errors can only be modeled with one return value, 0. ISO 9945 also only has one explicit error return (-1) but will then distinguish different errors by changing **`errno`**. We don't think that we should follow the lead here an introduce the dependency of a complicated thread specific state. Instead we should widen the possible return values to negative values, which could encode the negative of the value that would otherwise be found in **`errno`**.
Again, the specification is straight forward and can be inspected in the appendix, see the clauses 7.27.2.5 p5 and 7.26.2.6 p3.

Question 18. *Shall we change the possible error returns of function* **`timespec_get`** *(and* `timespec_getres` *if adopted) to allow negative values as proposed in N2402?*

If the answer is yes, we can to a similar addition as for `timespec_getres`.

Question 19. *Shall we change the return of* **`timespec_get`** *to* **`-EINVAL`** *if it is called with a non-supported base as proposed in N2402?*

10. RECOMMEND CONSISTENCY BETWEEN THE DIFFERENT TIME INTERFACES

As mentioned above there are several consistency issues between different interfaces. So we propose to add a "Recommended practice" section to **`timespec_get`** that encourages to have **`clock`** and **`time`** consistent with their respective counterparts for **`timespec_get`**.
We also add a recommendation to have the global and the thread-wise CPU time consistent, such that the sum of the thread-wise times should be the same as the global CPU time.
The exact wording is 7.27.2.5 p6.

Question 20. *Shall we add a recommendation for consistency between the legacy interfaces and* **`timespec_get`** *as proposed in N2402?*

Question 21. *Shall we add a recommendation for consistency amoung the CPU time interfaces of* **`timespec_get`** *as proposed in N2402?*

11. DEPRECATE `CLOCK`

As has been observed over the last revision cycle, the **`clock`** function has a severe problem with the fact that its return value can silently overflow without giving any indication to the caller. Additionally there is still a major legacy platform that gets the semantics of it fundamentally wrong by providing elapsed time instead of CPU time.
Therefore we think that **`clock`** should just be phased out. The proposed changes to 7.27.2.1 proposes that add recommends to use the new interface.

Question 22. *Should we make* **`clock`** *obsolescent as proposed in N2402?*

12. DEPRECATE THE NON-REENTRANT CONVERSION FUNCTIONS

The non-reentrant conversion functions are generally problematic and have to many design flaws. We propose to deprecated them:

QUESTION 23. *Should we make* **`asctime`**, **`ctime`**, **`localtime`**, *and* **`gmtime`** *obsolescent as proposed in N2402?*

13. APPENDIX: PAGES WITH DIFFMARKS OF THE PROPOSED CHANGES

The following page numbers are from the particular snapshot and may vary once the changes are integrated.

7.5 Errors `<errno.h>`

1 The header `<errno.h>` defines several macros, all relating to the reporting of error conditions.

2 The macros are

```
EDOM
EILSEQ
EINVAL
ERANGE
```

which expand to integer constant expressions with type **`int`**, distinct positive values, and which are suitable for use in **`#if`** preprocessing directives; and

```
errno
```

which expands to a modifiable lvalue[214] that has type **`int`** and thread local storage duration, the value of which is set to a positive error number by several library functions. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name **`errno`**, the behavior is undefined.

3 The value of **`errno`** in the initial thread is zero at program startup (the initial value of **`errno`** in other threads is an indeterminate value), but is never set to zero by any library function.[215] The value of **`errno`** may be set to nonzero by a library function call whether or not there is an error, provided the use of **`errno`** is not documented in the description of the function in this document.

4 Additional macro definitions, beginning with **`E`** and a digit or **`E`** and an uppercase letter,[216] may also be specified by the implementation.

---

[214] The macro **`errno`** need not be the identifier of an object. It might expand to a modifiable lvalue resulting from a function call (for example, `*errno()`).

[215] Thus, a program that uses **`errno`** for error checking would set it to zero before a library function call, then inspect it before a subsequent library function call. Of course, a library function can save the value of **`errno`** on entry and then set it to zero, as long as the original value is restored if **`errno`**'s value is still zero just before the return.

[216] See "future library directions" (7.31.3).

7.27 Date and time <time.h>

7.27.1 Components of time

1 The header <time.h> defines several macros, and declares types and functions for manipulating time. Many functions deal with a *calendar time* that represents the current date (according to the Gregorian calendar) and time. Some functions deal with *local time*, which is the calendar time expressed for some specific time zone, and with *Daylight Saving Time*, which is a temporary change in the algorithm for determining local time. The local time zone and Daylight Saving Time are implementation-defined.

2 The feature test macro `__STDC_VERSION_TIME_H__` expands to the token *yyyymm*`L`. The other macros defined are `NULL` (described in 7.19);

```
CLOCKS_PER_SEC
```

which expands to an expression with type `clock_t` (described below) that is the number per second of the value returned by the `clock` function; and time bases for `timespec_get` representable in `int`

```
TIME_UTC
TIME_REALTIME
TIME_MONOTONIC
```

which ~~expands to an integer constant greater than 0 that designates the UTC time base.~~are integer constant expressions suitable for use in `#if` preprocessing directives, and

```
TIME_PROCESS_CPUTIME_ID
TIME_THREAD_CPUTIME_ID
```

which may not be constants.[338)]

3 Time bases other than `TIME_UTC` are optional; all time bases evaluate to values greater than 0. If defined, `TIME_REALTIME` and `TIME_MONOTONIC` have different values, but `TIME_UTC` may share one of these values. The value of `TIME_PROCESS_CPUTIME_ID` shall be different from the above and shall not change during the same program execution. The macro `TIME_THREAD_CPUTIME_ID` shall not be defined if the implementation does not support threads; its value shall be different from the above, shall be the same for all invocations from the same thread, and the value provided for one thread shall not be used by a different thread as `base` argument of `timespec_get`.

4 The types declared are `size_t` (described in 7.19);

```
clock_t
```

and

```
time_t
```

which are real types capable of representing times;

```
struct timespec
```

which holds an interval specified in seconds and nanoseconds (which may represent a calendar time based on a particular epoch); and

```
struct tm
```

which holds the components of a calendar time, called the *broken-down time*.

338) Implementations can define additional time bases, but are only required to support a real time clock. See "future library directions" (7.31.16).

5 The range and precision of times representable in **clock_t** and **time_t** are implementation-defined. The **timespec** structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.[339)]

```
time_t tv_sec;  // whole seconds -- ≥ 0
long   tv_nsec; // nanoseconds -- [0, 999999999]
```

The **tm** structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.[340)]

```
int tm_sec;   // seconds after the minute -- [0, 60]
int tm_min;   // minutes after the hour -- [0, 59]
int tm_hour;  // hours since midnight -- [0, 23]
int tm_mday;  // day of the month -- [1, 31]
int tm_mon;   // months since January -- [0, 11]
int tm_year;  // years since 1900
int tm_wday;  // days since Sunday -- [0, 6]
int tm_yday;  // days since January 1 -- [0, 365]
int tm_isdst; // Daylight Saving Time flag
```

The value of **tm_isdst** is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

7.27.2 Time manipulation functions

7.27.2.1 The clock function

Synopsis

1
```
#include <time.h>
clock_t clock(void);
```

Description

2 The **clock** function determines the processor time used. It is an obsolescent feature.[341)]

Returns

3 The **clock** function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the **clock** function should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available, the function returns the value **(clock_t)**(-1). If the value cannot be represented, the function returns an unspecified value.[342)]

Recommended practice

4 Programs should prefer the use of the **timespec_get** function with a **base** argument of **TIME_PROCESS_CPUTIME_ID** whenever that functionality is defined by the implementation.

Forward references: the **timespec_get** function (7.27.2.5).

7.27.2.2 The difftime function

Synopsis

1
```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

[339)] The **tv_sec** member is a linear count of seconds and might not have the normal semantics of a **time_t**.

[340)] The range [0, 60] for **tm_sec** allows for a positive leap second.

[341)] See "future library directions" (7.31.16).

[342)] This could be due to overflow of the **clock_t** type.

Description

2 The **difftime** function computes the difference between two calendar times: `time1 - time0`.

Returns

3 The **difftime** function returns the difference expressed in seconds as a **double**.

7.27.2.3 The mktime function

Synopsis

1
```
#include <time.h>
~~time_t mktime(struct tm *timeptr);~~
time_t mktime(struct tm timeptr[static 1]);
```

Description

2 The **mktime** function converts the broken-down time, expressed as local time, in the structure pointed to by `timeptr` into a calendar time value with the same encoding as that of the values returned by the **time** function. The original values of the **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above.[343] On successful completion, the values of the **tm_wday** and **tm_yday** components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined.

Returns

3 The **mktime** function returns the specified calendar time encoded as a value of type **time_t**. If the calendar time cannot be represented, the function returns the value `(time_t)`(-1).

4 **EXAMPLE** What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/* ... */

time_str.tm_year   = 2001 - 1900;
time_str.tm_mon    = 7 - 1;
time_str.tm_mday   = 4;
time_str.tm_hour   = 0;
time_str.tm_min    = 0;
time_str.tm_sec    = 1;
time_str.tm_isdst  = -1;
if (mktime(&time_str) == (time_t)(-1))
      time_str.tm_wday = 7;
printf("%s\n", wday[time_str.tm_wday]);
```

7.27.2.4 The time function

Synopsis

1
```
#include <time.h>
time_t time(time_t *timer);
```

[343] Thus, a positive or zero value for **tm_isdst** causes the **mktime** function to presume initially that Daylight Saving Time, respectively, is or is not in effect for the specified time. A negative value causes it to attempt to determine whether Daylight Saving Time is in effect for the specified time.

Description

2 The `time` function determines the current calendar time. The encoding of the value is unspecified.

Returns

3 The `time` function returns the implementation's best approximation to the current calendar time. The value `(time_t)`(-1) is returned if the calendar time is not available. If `timer` is not a null pointer, the return value is also assigned to the object it points to.

7.27.2.5 The `timespec_get` function

Synopsis

1
```
#include <time.h>
~~int timespec_get(struct timespec *ts, int base);~~
int timespec_get(struct timespec ts[static 1], int base);
```

Description

2 The `timespec_get` function sets the interval pointed to by `ts` to hold the current calendar time based on the specified time base. For all supported bases, the resolution of the returned time values is implementation-defined and can be queried with `timespec_getres`.

3 If `base` is `TIME_UTC`, the `tv_sec` member is set to the number of seconds since an ~~implementation defined~~ implementation-defined *epoch*, truncated to a whole value and the `tv_nsec` member is set to the integral number of nanoseconds, rounded to the resolution of the system clock.[344] If `base` is `TIME_REALTIME` the behavior is the similar, only that the result is affected by implementation-defined functions that set the system time, if any. For `TIME_MONOTONIC` the reference point may or may not be the same epoch or any other implementation-defined time point. This point shall not change during the program execution and the result shall not be affected by any implementation-defined functions that set the system time, if any. If such functions are provided, it is implementation-defined if `TIME_UTC` behaves like the first or the second.

4 For `base` set to `TIME_PROCESS_CPUTIME_ID` and `TIME_THREAD_CPUTIME_ID` the result is similar, but the call measures the amount of processor time associated with the program execution or thread, respectively.

Returns

5 If the `timespec_get` function is successful, it returns the ~~nonzero~~ positive value `base`; if `base` is not supported it returns `-EINVAL` ; otherwise, it returns ~~zero.~~ another value less than or equal to zero.

Recommended practice

6 The following should be consistent whenever possible:

— The results of calls with `base` set to `TIME_UTC` and the return values of `time`.

— If defined, the results of calls with `base` `TIME_PROCESS_CPUTIME_ID` and the return values of `clock`.

— If both are defined, the results of calls with `base` set to `TIME_PROCESS_CPUTIME_ID` and `TIME_THREAD_CPUTIME_ID` , such that the sum of all thread specific processor times is as close to the processor time for the execution as possible.

Forward references: the `timespec_getres` function (7.27.2.6).

7.27.2.6 The `timespec_getres` function

Synopsis

1
```
#include <time.h>
int timespec_getres(struct timespec ts[static 1], int base);
```

[344] Although a `struct timespec` object describes times with nanosecond resolution, the available resolution is system dependent and could even be greater than 1 second.

Description

2 The **timespec_getres** function stores the implementation-defined resolution of the time provided by the **timespec_get** function for base in ts[0]. For each fixed value of base, the result shall be invariant during the program execution.

Returns

3 If the **timespec_getres** function is successful, it returns the positive value base; if base is not supported it returns-**EINVAL** ; otherwise, it returns another value less than or equal to zero.

7.27.3 Time conversion functions

1 ~~Except for the function~~Functions with a **_r** suffix shall be reentrant and shall not be subject to data races. Unless specified otherwise, they place the result of the conversion into the buffer referred by buf and return that pointer.

2 Obsolescent functions **asctime**, **ctime**, **gmtime**, and **localtime** are the same as their counterparts suffixed with **_r**.[345] In place of the parameter buf , these functions ~~each return~~ use a pointer to ~~one of two types of static objects: a~~ a static object and return it: one or two broken-down time ~~structure~~ structures (for **gmtime** and **localtime**) or an array of **char** (commonly used by **asctime** and **ctime**). Execution of any of the functions that return a pointer to one of these ~~object types~~ static objects may overwrite the information ~~in any object of the same type pointed to by the value~~ returned from any previous call to ~~any of them and the~~ one of these functions that uses the same object. These functions are not reentrant and are not required to avoid data races with each other. The implementation shall behave as if no other library functions call these functions.

7.27.3.1 The asctime functions

Synopsis

1
```
#include <time.h>
~~char *asctime(const struct tm *timeptr);~~
const char *asctime(const struct tm timeptr[static 1]);
char *asctime_r(const struct tm timeptr[static restrict 1],
          char buf[static restrict 26]);
```

Description

2 The **asctime** functions ~~converts~~ convert the broken-down time in the structure pointed to by timeptr into a string in the form

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm.

```
~~char *asctime(const struct tm *timeptr)~~
char *asctime_r(const struct tm timeptr[static restrict 1],
          char buf[static restrict 26]);
{
          static const char wday_name[7][3] = {
                    "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
          };
          static const char mon_name[12][3] = {
                    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
          };
~~          static char result[26];~~

~~          sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",~~
          snprintf(buf, 26, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
                    wday_name[timeptr->tm_wday],
```

345) See "future library directions" (7.31.16).

```
            mon_name[timeptr->tm_mon],
            timeptr->tm_mday, timeptr->tm_hour,
            timeptr->tm_min, timeptr->tm_sec,
            1900 + timeptr->tm_year);
      return result;
      return buf;
}
```

3 The return value points to a zero terminated string of length at most 25 and no write beyond the 26th byte occurs. If any of the members of the broken-down time contain values that are outside their normal ranges,[346] ~~the behavior of the is undefined. Likewise,~~ or if the calculated year exceeds four digits or is less than the year 1000, the ~~behavior is undefined~~contents of the returned string is otherwise unspecified.

Returns

4 The **asctime** functions ~~returns~~ return a pointer to the string.

7.27.3.2 The ctime functions

Synopsis

1
```
      #include <time.h>
      char *ctime(const time_t *timer);
      const char *ctime(const time_t timer[static 1]);
      char *ctime_r(const time_t timer[static restrict 1],
            char buf[static restrict 26]);
```

Description

2 The ~~converts the calendar time pointed to by~~ calendar time ~~timer~~ specified by timer[0] shall be convertible to local time. The **ctime** functions convert the specified time to local time in the form of a string. ~~It is~~ They are equivalent to

```
      asctime(localtime(timer))
```

and

```
      asctime_r(localtime_r(timer, (struct tm[1]){ 0 }), buf)
```

Returns

3 The **ctime** functions ~~returns~~ return the pointer returned by the **asctime** ~~function~~ functions with that broken-down time as argument.

Forward references: the **localtime** ~~function (??~~functions (7.27.3.4).

7.27.3.3 The gmtime functions

Synopsis

1
```
      #include <time.h>
      struct tm *gmtime(const time_t *timer);
      const struct tm *gmtime(const time_t timer[static 1]);
      struct tm *gmtime_r(const time_t timer[static 1], struct tm buf[static 1]);
```

Description

2 The **gmtime** functions ~~converts~~ convert the calendar time pointed to by timer into a broken-down time, expressed as UTC.

[346]See 7.27.1.

Returns

3 The **gmtime** functions ~~returns~~ return a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to UTC.

7.27.3.4 The **localtime** functions

Synopsis

1
```
#include <time.h>
struct tm *localtime(const time_t *timer);
const struct tm *localtime(const time_t timer[static 1]);
struct tm *localtime_r(const time_t timer[static 1], struct tm buf[static 1]);
```

Description

2 The **localtime** functions converts the calendar time pointed to by `timer` into a broken-down time, expressed as local time.

Returns

3 The **localtime** functions ~~returns~~ return a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to local time.

7.27.3.5 The **strftime** function

Synopsis

1
```
#include <time.h>
size_t strftime(char * restrict s, size_t maxsize, const char * restrict format,
          const struct tm * restrict timeptr);
size_t strftime(char s[static restrict 1], size_t maxsize,
          const char format[static restrict 3],
          const struct tm timeptr[static restrict 1]);
```

Description

2 The **strftime** function places characters into the array pointed to by `s` as controlled by the string pointed to by `format`. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The `format` string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a `%` character, possibly followed by an `E` or `O` modifier character (described below), followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than `maxsize` characters are placed into the array.

3 Each conversion specifier shall be replaced by appropriate characters as described in the following list. The appropriate characters shall be determined using the **LC_TIME** category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by `timeptr`, as specified in brackets in the description. If any of the specified values is outside the normal range, the characters stored are unspecified.

`%a` is replaced by the locale's abbreviated weekday name. [**tm_wday**]

`%A` is replaced by the locale's full weekday name. [**tm_wday**]

`%b` is replaced by the locale's abbreviated month name. [**tm_mon**]

`%B` is replaced by the locale's full month name. [**tm_mon**]

`%c` is replaced by the locale's appropriate date and time representation. [all specified in 7.27.1]

`%C` is replaced by the year divided by 100 and truncated to an integer, as a decimal number (`00`–`99`). [**tm_year**]

`%d` is replaced by the day of the month as a decimal number (`01`–`31`). [**tm_mday**]

`%D` is equivalent to "`%m/%d/%y`". [**tm_mon**, **tm_mday**, **tm_year**]

7.31.10 Atomics <stdatomic.h>

1 Macros that begin with **ATOMIC_** and an uppercase letter may be added to the macros defined in the <stdatomic.h> header. Typedef names that begin with either **atomic_** or **memory_**, and a lowercase letter may be added to the declarations in the <stdatomic.h> header. Enumeration constants that begin with **memory_order_** and a lowercase letter may be added to the definition of the **memory_order** type in the <stdatomic.h> header. Function names that begin with **atomic_** and a lowercase letter may be added to the declarations in the <stdatomic.h> header.

2 The macro **ATOMIC_VAR_INIT** is an obsolescent feature.

7.31.11 Boolean type and values <stdbool.h>

1 The ability to undefine and perhaps then redefine the macros **bool**, **true**, and **false** is an obsolescent feature.

7.31.12 Integer types <stdint.h>

1 Typedef names beginning with **int** or **uint** and ending with **_t** may be added to the types defined in the <stdint.h> header. Macro names beginning with **INT** or **UINT** and ending with **_MAX**, **_MIN**, **_WIDTH**, or **_C** may be added to the macros defined in the <stdint.h> header.

7.31.13 Input/output <stdio.h>

1 Lowercase letters may be added to the conversion specifiers and length modifiers in **fprintf** and **fscanf**. Other characters may be used in extensions.

2 The use of **ungetc** on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

7.31.14 General utilities <stdlib.h>

1 Function names that begin with **str** or **wcs** and a lowercase letter may be added to the declarations in the <stdlib.h> header.

2 Invoking **realloc** with a size argument equal to zero is an obsolescent feature.

7.31.15 String handling <string.h>

1 Function names that begin with **str**, **mem**, or **wcs** and a lowercase letter may be added to the declarations in the <string.h> header.

7.31.16 Date and time <time.h>

1 Macros beginning with **TIME_** and an uppercase letter may be added to the macros in the <time.h> header. The macros **TIME_REALTIME**, **TIME_MONOTONIC**, **TIME_PROCESS_CPUTIME_ID** and **TIME_THREAD_CPUTIME_ID** may become mandatory in future editions of this standard.

2 The functions **asctime**, **ctime**, **gmtime**, and **localtime** are obsolescent features.

3 The function **clock** and the associated return type **clock_t** and macro **CLOCKS_PER_SEC** are obsolescent features.

7.31.17 Threads <threads.h>

1 Function names, type names, and enumeration constants that begin with either **cnd_**, **mtx_**, **thrd_**, or **tss_**, and a lowercase letter may be added to the declarations in the <threads.h> header.

7.31.18 Extended multibyte and wide character utilities <wchar.h>

1 Function names that begin with **wcs** and a lowercase letter may be added to the declarations in the <wchar.h> header.

2 Lowercase letters may be added to the conversion specifiers and length modifiers in **fwprintf** and **fwscanf**. Other characters may be used in extensions.