NAME

clock_getres, clock_gettime, clock_settime − clock and time functions

LIBRARY

Standard C library (*libc*, *−lc*), since glibc 2.17

Before glibc 2.17, Real-time library (*librt*, *−lrt*)

SYNOPSIS

```
#include <time.h>

int clock_getres(clockid_t clockid, struct timespec *_Nullable res);

int clock_gettime(clockid_t clockid, struct timespec *tp);
int clock_settime(clockid_t clockid, const struct timespec *tp);
```

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

clock_getres(), **clock_gettime**(), **clock_settime**():
_POSIX_C_SOURCE >= 199309L

DESCRIPTION

The function **clock_getres**() finds the resolution (precision) of the specified clock *clockid*, and, if *res* is non-NULL, stores it in the *struct timespec* pointed to by *res*. The resolution of clocks depends on the implementation and cannot be configured by a particular process. If the time value pointed to by the argument *tp* of **clock_settime**() is not a multiple of *res*, then it is truncated to a multiple of *res*.

The functions **clock_gettime**() and **clock_settime**() retrieve and set the time of the specified clock *clockid*.

The *res* and *tp* arguments are **timespec**(3) structures.

The *clockid* argument is the identifier of the particular clock on which to act. A clock may be system-wide and hence visible for all processes, or per-process if it measures time only within a single process.

All implementations support the system-wide real-time clock, which is identified by **CLOCK_REALTIME**. Its time represents seconds and nanoseconds since the Epoch. When its time is changed, timers for a relative interval are unaffected, but timers for an absolute point in time are affected.

More clocks may be implemented. The interpretation of the corresponding time values and the effect on timers is unspecified.

Sufficiently recent versions of glibc and the Linux kernel support the following clocks:

CLOCK_REALTIME
A settable system-wide clock that measures real (i.e., wall-clock) time. Setting this clock requires appropriate privileges. This clock is affected by discontinuous jumps in the system time (e.g., if the system administrator manually changes the clock), and by the incremental adjustments performed by **adjtime**(3) and NTP.

CLOCK_REALTIME_ALARM (since Linux 3.0; Linux-specific)
Like **CLOCK_REALTIME**, but not settable. See **timer_create**(2) for further details.

CLOCK_REALTIME_COARSE (since Linux 2.6.32; Linux-specific)
A faster but less precise version of **CLOCK_REALTIME**. This clock is not settable. Use when you need very fast, but not fine-grained timestamps. Requires per-architecture support, and probably also architecture support for this flag in the **vdso**(7).

CLOCK_TAI (since Linux 3.10; Linux-specific)
A nonsettable system-wide clock derived from wall-clock time but ignoring leap seconds. This clock does not experience discontinuities and backwards jumps caused by NTP inserting leap seconds as **CLOCK_REALTIME** does.

The acronym TAI refers to International Atomic Time.

CLOCK_MONOTONIC
A nonsettable system-wide clock that represents monotonic time since—as described by POSIX—"some unspecified point in the past". On Linux, that point corresponds to the number of seconds that the system has been running since it was booted.

The **CLOCK_MONOTONIC** clock is not affected by discontinuous jumps in the system time (e.g., if the system administrator manually changes the clock), but is affected by the incremental adjustments performed by **adjtime**(3) and NTP. This clock does not count time that the system is suspended. All**CLOCK_MONO TONIC** variants guarantee that the time returned by consecutive calls will not go backwards, but successive calls may—depending on the architecture—return identical (not-increased) time values.

CLOCK_MONOTONIC_COARSE (since Linux 2.6.32; Linux-specific)
A faster but less precise version of **CLOCK_MONOTONIC**. Use when you need very fast, but not fine-grained timestamps. Requires per-architecture support, and probably also architecture support for this flag in the **vdso**(7).

CLOCK_MONOTONIC_RAW (since Linux 2.6.28; Linux-specific)
Similar to **CLOCK_MONOTONIC**, but provides access to a raw hardware-based time that is not subject to NTP adjustments or the incremental adjustments performed by **adjtime**(3). This clock does not count time that the system is suspended.

CLOCK_BOOTTIME (since Linux 2.6.39; Linux-specific)
A nonsettable system-wide clock that is identical to **CLOCK_MONOTONIC**, except that it also includes any time that the system is suspended. This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of **CLOCK_REALTIME**, which may have discontinuities if the time is changed using **settimeofday**(2) or similar.

CLOCK_BOOTTIME_ALARM (since Linux 3.0; Linux-specific)
Like **CLOCK_BOOTTIME**. See**timer_cr eate**(2) for further details.

CLOCK_PROCESS_CPUTIME_ID (since Linux 2.6.12)
This is a clock that measures CPU time consumed by this process (i.e., CPU time consumed by all threads in the process). On Linux, this clock is not settable.

CLOCK_THREAD_CPUTIME_ID (since Linux 2.6.12)
This is a clock that measures CPU time consumed by this thread. On Linux, this clock is not settable.

Linux also implements dynamic clock instances as described below.

Dynamic clocks
In addition to the hard-coded System-V style clock IDs described above, Linux also supports POSIX clock operations on certain character devices. Such devices are called "dynamic" clocks, and are supported since Linux 2.6.39.

Using the appropriate macros, open file descriptors may be converted into clock IDs and passed to **clock_gettime**(), **clock_settime**(), and **clock_adjtime**(2). The following example shows how to convert a file descriptor into a dynamic clock ID.

```
#define CLOCKFD 3
#define FD_TO_CLOCKID(fd)   ((~(clockid_t) (fd) << 3) | CLOCKFD)
#define CLOCKID_TO_FD(clk)  ((unsigned int) ~((clk) >> 3))

struct timespec ts;
clockid_t clkid;
int fd;

fd = open("/dev/ptp0", O_RDWR);
clkid = FD_TO_CLOCKID(fd);
clock_gettime(clkid, &ts);
```

RETURN VALUE

clock_gettime(), **clock_settime**(), and **clock_getres**() return 0 for success. On error, −1 is returned and *errno* is set to indicate the error.

ERRORS

EACCES

clock_settime() does not have write permission for the dynamic POSIX clock device indicated.

EFAULT

tp points outside the accessible address space.

EINVAL

The *clockid* specified is invalid for one of two reasons. Either the System-V style hard coded positive value is out of range, or the dynamic clock ID does not refer to a valid instance of a clock object.

EINVAL

(**clock_settime**()): *tp.tv_sec* is negative or *tp.tv_nsec* is outside the range [0, 999,999,999].

EINVAL

The *clockid* specified in a call to **clock_settime**() is not a settable clock.

EINVAL (since Linux 4.3)

A call to **clock_settime**() with a *clockid* of **CLOCK_REALTIME** attempted to set the time to a value less than the current value of the **CLOCK_MONOTONIC** clock.

ENODEV

The hot-pluggable device (like USB for example) represented by a dynamic *clk_id* has disappeared after its character device was opened.

ENOTSUP

The operation is not supported by the dynamic POSIX clock device specified.

EPERM

clock_settime() does not have permission to set the clock indicated.

VERSIONS

These system calls first appeared in Linux 2.6.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
clock_getres(), **clock_gettime**(), **clock_settime**()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2001, POSIX.1-2008, SUSv2.

On POSIX systems on which these functions are available, the symbol **_POSIX_TIMERS** is defined in *<unistd.h>* to a value greater than 0. The symbols **_POSIX_MONOTONIC_CLOCK**, **_POSIX_CPUTIME**, **_POSIX_THREAD_CPUTIME** indicate that **CLOCK_MONOTONIC**, **CLOCK_PROCESS_CPUTIME_ID**, **CLOCK_THREAD_CPUTIME_ID** are available. (See also **sysconf**(3).)

NOTES

POSIX.1 specifies the following:

> Setting the value of the **CLOCK_REALTIME** clock via **clock_settime**() shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the **nanosleep**() function; nor on the expiration of relative timers based upon this clock. Consequently, these time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.

According to POSIX.1-2001, a process with "appropriate privileges" may set the **CLOCK_PROCESS_CPUTIME_ID** and **CLOCK_THREAD_CPUTIME_ID** clocks using **clock_settime**(). On Linux, these clocks are not settable (i.e., no process has "appropriate privileges").

C library/kernel differences

On some architectures, an implementation of **clock_gettime**() is provided in the **vdso**(7).

Historical note for SMP systems

Before Linux added kernel support for **CLOCK_PROCESS_CPUTIME_ID** and **CLOCK_THREAD_CPUTIME_ID**, glibc implemented these clocks on many platforms using timer registers from the CPUs (TSC on i386, AR.ITC on Itanium). These registers may differ between CPUs and as a consequence these clocks may return **bogus results** if a process is migrated to another CPU.

If the CPUs in an SMP system have different clock sources, then there is no way to maintain a correlation between the timer registers since each CPU will run at a slightly different frequency. If that is the case, then *clock_getcpuclockid(0)* will return **ENOENT** to signify this condition. The two clocks will then be useful only if it can be ensured that a process stays on a certain CPU.

The processors in an SMP system do not start all at exactly the same time and therefore the timer registers are typically running at an offset. Some architectures include code that attempts to limit these offsets on bootup. However, the code cannot guarantee to accurately tune the offsets. glibc contains no provisions to deal with these offsets (unlike the Linux Kernel). Typically these offsets are small and therefore the effects may be negligible in most cases.

Since glibc 2.4, the wrapper functions for the system calls described in this page avoid the abovementioned problems by employing the kernel implementation of **CLOCK_PROCESS_CPUTIME_ID** and **CLOCK_THREAD_CPUTIME_ID**, on systems that provide such an implementation (i.e., Linux 2.6.12 and later).

EXAMPLES

The program below demonstrates the use of **clock_gettime**() and **clock_getres**() with various clocks. This is an example of what we might see when running the program:

```
$ ./clock_times x
CLOCK_REALTIME : 1585985459.446 (18356 days +  7h 30m 59s)
     resolution:          0.000000001
CLOCK_TAI      : 1585985496.447 (18356 days +  7h 31m 36s)
     resolution:          0.000000001
CLOCK_MONOTONIC:      52395.722 (14h 33m 15s)
     resolution:          0.000000001
CLOCK_BOOTTIME :      72691.019 (20h 11m 31s)
     resolution:          0.000000001
```

Program source

```
/* clock_times.c

   Licensed under GNU General Public License v2 or later.
*/
#define _XOPEN_SOURCE 600
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SECS_IN_DAY (24 * 60 * 60)

static void
```

```
displayClock(clockid_t clock, const char *name, bool showRes)
{
    long            days;
    struct timespec  ts;

    if (clock_gettime(clock, &ts) == -1) {
        perror("clock_gettime");
        exit(EXIT_FAILURE);
    }

    printf("%-15s: %10jd.%03ld (", name,
           (intmax_t) ts.tv_sec, ts.tv_nsec / 1000000);

    days = ts.tv_sec / SECS_IN_DAY;
    if (days > 0)
        printf("%ld days + ", days);

    printf("%2dh %2dm %2ds",
           (int) (ts.tv_sec % SECS_IN_DAY) / 3600,
           (int) (ts.tv_sec % 3600) / 60,
           (int) ts.tv_sec % 60);
    printf(")\n");

    if (clock_getres(clock, &ts) == -1) {
        perror("clock_getres");
        exit(EXIT_FAILURE);
    }

    if (showRes)
        printf("     resolution: %10jd.%09ld\n",
               (intmax_t) ts.tv_sec, ts.tv_nsec);
}

int
main(int argc, char *argv[])
{
    bool showRes = argc > 1;

    displayClock(CLOCK_REALTIME, "CLOCK_REALTIME", showRes);
#ifdef CLOCK_TAI
    displayClock(CLOCK_TAI, "CLOCK_TAI", showRes);
#endif
    displayClock(CLOCK_MONOTONIC, "CLOCK_MONOTONIC", showRes);
#ifdef CLOCK_BOOTTIME
    displayClock(CLOCK_BOOTTIME, "CLOCK_BOOTTIME", showRes);
#endif
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

date(1), **gettimeofday**(2), **settimeofday**(2), **time**(2), **adjtime**(3), **clock_getcpuclockid**(3), **ctime**(3), **ftime**(3), **pthread_getcpuclockid**(3), **sysconf**(3), **timespec**(3), **time**(7), **time_namespaces**(7), **vdso**(7), **hwclock**(8)