Timekeeping

UNIX timekeeping is an untidy area, made more confusing by national and international laws and customs. Broadly, there are two kinds of functions: one group is concerned with getting and setting system times, and the other group is concerned with converting time representations between a bewildering number of formats.

Before we start, we'll define some terms:

- A *time zone* is a definition of the time at a particular location relative to the time at other locations. Most time zones are bound to political borders, and vary from one another in steps of one hour, although there are still a number of time zones that are offset from adjacent time zones by 30 minutes. Time zones tend to have three-letter abbreviations (*TLA*s) such as *PST* (*Pacific Standard Time*), *EDT* (*Eastern Daylight Time*), *BST* (*British Summer Time*), *AET* (*Australian Eastern Time*), *MEZ* (*Mitteleuropäische Zeit*). As the example shows, you should not rely on the combination *ST* to represent *Standard Time*.
- *UTC* is the international base time zone, and has the distinction of being one of those abbreviations which nobody can expand. It means *Universal Coordinated Time*, despite the initials. It obviously doesn't stand for the French *Temps Universel Coordonné* either. It corresponds very closely, but not exactly, to Greenwich Mean Time (GMT), the local time in England in the winter, and is the basis of all UNIX timestamps. The result is that for most of us, UTC is *not* the current local time, though it might be close enough to be confusing or far enough away to be annoying.
- From the standpoint of UNIX, you can consider the *Epoch* to be the beginning of recorded history: it's 00:00:00 UTC, 1 January 1970. All system internal dates are relative to the Epoch.
- Daylight Savings Time is a method of making the days appear longer in summer by setting the clocks forward, usually by one hour. Thus in summer, the sun appears to set one hour later than would otherwise be the case.

Even after clarifying these definitions, timekeeping remains a pain. We'll look at the main problems in the following sections:

Difficult to use

The time functions are not easy to use: to get them to do anything useful requires a lot of work. You'd think that UNIX would supply a primitive call upon which you could easily build, but unfortunately there isn't any such call, and the ones that are available do not operate in an intuitively obvious way. For example, there is no standard function for returning the current time in a useful format.

Implementations differ

There is no single system call that is supported across all platforms. Functions are implemented as system calls in some systems and as library functions in others. As a result, it doesn't make sense to maintain our distinction between system calls and library functions when it comes to timekeeping. In our discussion of the individual functions, we'll note which systems implement them as system calls and which as library calls.

Differing time formats

There are at least four different time formats:

- The system uses the *time_t* format, which represents the number of seconds since the Epoch. This format is not subject to time zones or daylight savings time, but it is accurate only to one second, which is not accurate enough for many applications.
- The *struct timeval* format is something like an extended `time_t` with a resolution of 1 microsecond:

```
#include <sys/time.h>

struct timeval
  {
  long    tv_sec;                     /* seconds since Jan. 1, 1970 */
  long    tv_usec;                    /* and microseconds */
  };
```

  It is used for a number of newer functions, such as `gettimeofday` and `setitimer`.
- Many library routines represent the calendar time as a struct *tm*. It is usually defined in */usr/include/time.h*:

```
struct tm
  {
  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 Savings Time flag */
```

```
long  tm_gmtoff;                    /* offset from UTC in seconds */
char  *tm_zone;                     /* timezone abbreviation */
};
```

Unlike `time_t`, a *struct tm* does not uniquely define the time: it may be a UTC time, or it may be local time, depending on the time zone information for the system.

- Dates as a text string are frequently represented in a strange manner, for example `Sat Sep 17 14:28:03 1994\n`. This format includes a `\n` character, which is seldom needed—often you will have to chop it off again.

Daylight Savings Time

The support for Daylight Savings Time was rudimentary in the Seventh Edition, and the solutions that have arisen since then are not completely compatible. In particular, System V handles Daylight Savings Time via environment variables, so one user's view of time could be different from the next. Recent versions of BSD handle this via a database that keeps track of local regulations.

National time formats

Printable representations of dates and times are very much a matter of local customs. For example, the date *9/4/94* (in the USA) would be written as *4/9/94* in Great Britain and *04.09.94* in Germany. The time written as *4:23 pm* in the USA would be written *16.23* in France. Things get even worse if you want to have the names of the days and months. As a result, many timekeeping functions refer to the *locale* kept by ANSI C. The locale describes country-specific information. Since it does not vary from one system to the next, we won't look at it in more detail—see *POSIX Programmer's Guide*, by Donald Lewine, for more information.

Global timekeeping variables

A number of global variables define various aspects of timekeeping:

- The variable *timezone*, which is used in System V and XENIX, specifies the number of minutes that the standard time zone is west of Greenwich. It is set from the environment variable `TZ`, which has a rather bizarre syntax. For example, in Germany daylight savings time starts on the last Sunday of March and ends on the last Sunday of September (not October as in some other countries, including the USA). To tell the system about this, you would use the `TZ` string

  ```
  MEZ-1MSZ-2;M3.5,M9.5
  ```

  This states that the standard time zone is called *MEZ*, and that it is one hour ahead of UTC, that the summer time zone is called *MSZ*, and that it is two hours ahead of UTC. Summer time begins on the (implied Sunday of the) fifth week in March and ends in the fifth week of September.

The punctuation varies: this example comes from System V.3, which requires a semicolon in the indicated position. Other systems allow a comma here, which works until you try to move the information to System V.3.

- The variable `altzone`, used in SVR4 and XENIX, specifies the number of minutes that the Daylight Savings Time zone is west of Greenwich.
- The variable `daylight`, used in SVR4 and XENIX, indicates that Daylight Savings Time is currently in effect.
- The variable *tzname*, used in BSD, SVR4 and XENIX, is a pointer to two strings, specifying the name of the standard time zone and the Daylight Savings Time zone respectively.

In the following sections we'll look at how to get the current time, how to set the current time, how to convert time values, and how to suspend process execution for a period of time.

Getting the current time

The system supplies the current time via the system calls `time` or `gettimeofday`—only one of these is a system call, but the system determines which one it is.

time

```
#include <sys/types.h>
#include <time.h>

time_t time (time_t *tloc);
```

`time` returns the current time in `time_t` form, both as a return value and at *tloc* if this is not `NULL`. `time` is implemented as a system call in System V and as a library function (which calls `gettimeofday`) in BSD. Since it returns a scalar value, a call to `time` can be used as a parameter to functions like `localtime` or `ctime`.

ftime

`ftime` is a variant of `time` that returns time information with a resolution of one millisecond. It originally came from 4.2BSD, but is now considered obsolete.

```
#include <sys/types.h>
#include <sys/timeb.h>

typedef long time_t;                    /* (typically) */

struct timeb
  {
  time_t   time;                        /* the same time returned by time */
  unsigned short millitm;               /* Milliseconds */
  short    timezone;                    /* System default time zone */
  short    dstflag;                     /* set during daylight savings time */
```

```
  };

struct timeb *ftime (struct timeb *tp);
```

The timezone returned is the system default, possibly not what you want. System V.4 deprecates* the use of this variable as a result. Depending on which parameters are actually used, there are a number of alternatives to `ftime`. In many cases, `time` supplies all you need. However, `time` is accurate only to one second.

On some systems, you may be able to define `ftime` in terms of `gettimeofday`, which returns the time of the day with a 1 microsecond resolution—see the next section. On other systems, unfortunately, the system clock does not have a finer resolution than one second, and you are stuck with `time`.

gettimeofday

```
#include <sys/time.h>

struct timeval
  {
  long    tv_sec;                     /* seconds since Jan. 1, 1970 */
  long    tv_usec;                    /* and microseconds */
  };

int gettimeofday (struct timeval *tp,
                  struct timezone *tzp); /* (BSD) */
int gettimeofday (struct timeval *tp);  /* (System V.4) */
```

`gettimeofday` returns the current system time, with a resolution of 1 microsecond, to `tp`. The name is misleading, since the `struct timeval` representation does not relate to the time of day. Many implementations ignore `tzp`, but others, such as SunOS 4, return time zone information there.

In BSD, `gettimeofday` is a system call. In some versions of System V.4 it is emulated as a library function defined in terms of `time`, which limits its resolution to 1 second. Other versions of System V appear to have implemented it as a system call, though this is not documented.

---

* The term *deprecate* is a religious term meaning "to seek to avert by prayer". Nowadays used to indicate functionality that the implementors or maintainers wish would go away. This term seems to have come from Berkeley. To quote the "New Hackers Dictionary":

> **:deprecated:** adj. Said of a program or feature that is considered obsolescent and in the process of being phased out, usually in favor of a specified replacement. Deprecated features can, unfortunately, linger on for many years. This term appears with distressing frequency in standards documents when the committees writing the documents realize that large amounts of extant (and presumably happily working) code depend on the feature(s) that have passed out of favor. See also {dusty deck}.

Setting the current time

Setting the system time is similar to getting it, except that for security reasons only the superuser (*root*) is allowed to perform the function. It is normally executed by the *date* program.

adjtime

```
#include <sys/time.h>

int adjtime (struct timeval *delta, struct timeval *olddelta);
```

`adjtime` makes small adjustments to the system time, and is intended to help synchronize time in a network. The adjustment is made gradually—the system slows down or speeds up the passage of time by a fraction of a percent until it has made the correction, in order not to confuse programs like *cron* which are watching the time. As a result, if you call `adjtime` again, the previous adjustment might still not be complete; in this case, the remaining adjustment is returned in `olddelta`. `adjtime` was introduced in 4.3BSD and is also supported by System V. It is implemented as a system call in all systems.

settimeofday

```
#include <sys/time.h>

int gettimeofday (struct timeval *tp, struct timezone *tzp);
int settimeofday (struct timeval *tp, struct timezone *tzp);
```

`settimeofday` is a BSD system call that is emulated as a library function in System V.4. It sets the current system time to the value of `tp`. The value of `tzp` is no longer used. In System V, this call is implemented in terms of the `stime` system call, which sets the time only to the nearest second. If you really need to set the time more accurately in System V.4, you can use `adjtime`.

stime

```
#include <unistd.h>

int stime (const time_t *tp);
```

`stime` sets the system time and date. This is the original Seventh Edition function that is still available in System V. It is *not* supported in BSD—use `settimeofday` instead on BSD systems.

Converting time values

As advertised, there are a large number of time conversion functions, made all the more complicated because many are supported only on specific platforms. All are library functions. Many return pointers to static data areas that are overwritten by the next call. Solaris attempts to solve this problem with versions of the functions with the characters `_r` (for *reentrant*)

appended to their names. These functions use a user-supplied buffer to store the data they return.

strftime

```
#include <sys/types.h>
#include <time.h>
#include <string.h>

size_t strftime (char *s, size_t maxsize, char *format, struct tm *tm);
```

`strftime` converts the time at `tm` into a formatted string at `s`. `format` specifies the format of the resultant string, which can be no longer than `maxsize` characters. `format` is similar to the format strings used by `printf`, but contains strings related to dates. `strftime` has a rather strange return value: if the complete date string, including the terminating NUL character, fits into the space provided, it returns the length of the string—otherwise it returns 0, which implies that the date string has been truncated.

`strftime` is available on all platforms and is implemented as a library function. System V.4 considers `ascftime` and `cftime` to be obsolete. The man pages state that `strftime` should be used instead.

strptime

```
#include <time.h>

char *strptime (char *buf, char *fmt, struct tm *tm);
```

`strptime` is a library function supplied with SunOS 4. It converts the date and time string `buf` into a `struct tm` value at `tm`. This call bears the same relationship to `scanf` that `strftime` bears to `printf`.

ascftime

```
#include <sys/types.h>
#include <time.h>

int ascftime (char *buf, char *fmt, tm *tm);
```

`ascftime` converts the time at `tm` into a formatted string at `buf`. `format` specifies the format of the resultant string. This is effectively the same function as `strftime`, except that there is no provision to supply the maximum length of `buf`. `ascftime` is available on all platforms and is implemented as a library function.

asctime and asctime_r

```
#include <sys/types.h>
#include <time.h>

char *asctime (const struct tm *tm);
```

```
char *asctime_r (const struct tm *tm, char *buf, int buflen);
```

`asctime` converts a time in `struct tm*` format into the same kind of string that is returned by `ctime`. `asctime` is available on all platforms and is always a library function.

`asctime_r` is a version of `asctime` that returns the string to the user-provided buffer `res`, which must be at least `buflen` characters long. It returns the address of `res`. It is supplied as a library function on Solaris systems.

cftime

```
#include <sys/types.h>
#include <time.h>

int cftime (char *buf, char *fmt, time_t *clock);
```

`cftime` converts the time at `clock` into a formatted string at `buf`. `format` specifies the format of the resultant string. This is effectively the same function as `strftime`, except that there is no provision to supply the maximum length of `buf`, and the time is supplied in `time_t` format. `cftime` is available on all platforms and is implemented as a library function.

ctime and ctime_r

```
#include <sys/types.h>
#include <time.h>
extern char *tzname[2];

char *ctime (const time_t *clock);
char *ctime_r (const time_t *clock, char *buf, int buflen);
```

`ctime` converts the time `clock` into a string in the form `Sat Sep 17 14:28:03 1994\n`, which has the advantage of consistency: it is not a normal representation anywhere in the world, and immediately brands any printed output with the word *UNIX*. It uses the environment variable `TZ` to determine the current time zone. You can rely on the string to be exactly 26 characters long, including the final `\0`, and to contain that irritating `\n` at the end. `ctime` is available on all platforms and is always a library function.

`ctime_r` is a version of `ctime` that returns its result in the buffer pointed to by `buf`. The length is limited to `buflen` bytes. `ctime_r` is available on Solaris platforms as a library function.

dysize

```
#include <time.h>

int dysize (int year);
```

`dysize` return the number of days in `year`. It is supplied as a library function in SunOS 4.

gmtime and gmtime_r

```
#include <time.h>

struct tm *gmtime (const time_t *clock);
struct tm *gmtime_r (const time_t *clock, struct tm *res);
```

`gmtime` converts a time in `time_t` format into `struct tm*` format, like `localtime`. As the name suggests, however, it does not account for local timezones—it returns a UTC time (this was formerly called Greenwich Mean Time, thus the name of the function). `gmtime` is available on all platforms and is always a library function.

`gmtime_r` is a version of `gmtime` that returns the string to the user-provided buffer `res`. It returns the address of `res`. It is supplied as a library function on Solaris systems.

localtime and localtime_r

```
#include <time.h>

struct tm *localtime (const time_t *clock);
struct tm *localtime_r (const time_t *clock, struct tm *res);
```

`localtime` converts a time in `time_t` format into `struct tm*` format. Like `ctime`, it uses the time zone information in `tzname` to convert to local time. `localtime` is available on all platforms and is always a library function.

`localtime_r` is a version of `localtime` that returns the string to the user-provided buffer `res`. It returns the address of `res`. It is supplied as a library function on Solaris systems.

mktime

```
#include <sys/types.h>
#include <time.h>
time_t mktime (struct tm *tm);
```

`mktime` converts a local time in `struct tm` format into a time in `time_t` format. It does not use `tzname` in the conversion—it uses the information at `tm->tm_zone` instead. In addition to converting the time, `mktime` also sets the members `wday` (day of week) and `yday` (day of year) of the *input* struct tm to agree with day, month and year. `tm->tm_isdst` determines whether Daylight Savings Time is applicable:

- if it is > 0, `mktime` assumes Daylight Savings Time is in effect.
- If it is 0, it assumes that no Daylight Savings Time is in effect.
- If it is < 0, `mktime` tries to determine whether Daylight Savings Time is in effect or not. It is often wrong.

`mktime` is available on all platforms and is always a library function.

timegm

```
#include <time.h>

time_t timegm (struct tm *tm);
```

`timegm` converts a `struct tm` time, assumed to be UTC, to the corresponding `time_t` value. This is effectively the same thing as `mktime` with the time zone set to UTC, and is the converse of `gmtime`. `timegm` is a library function supplied with SunOS 4.

timelocal

```
#include <time.h>

time_t timelocal (struct tm *tm);
```

`timelocal` converts a `struct tm` time, assumed to be local time, to the corresponding `time_t` value. This is similar to `mktime`, but it uses the local time zone information instead of the information in `tm`. It is also the converse of `localtime`. `timelocal` is a library function supplied with SunOS 4.

difftime

```
#include <sys/types.h>
#include <time.h>

double difftime (time_t time1, time_t time0);
```

`difftime` returns the difference in seconds between two `time_t` values. This is effectively the same thing as `(int) time1 - (int) time0`. `difftime` is a library function available on all platforms.

timezone

```
#include <time.h>

char *timezone (int zone, int dst);
```

`timezone` returns the name of the timezone that is `zone` minutes west of Greenwich. If `dst` is non-0, the name of the Daylight Savings Time zone is returned instead. This call is obsolete—it was used at a time when time zone information was stored as the number of minutes west of Greenwich. Nowadays the information is stored with a time zone name, so there should be no need for this function.

tzset

```
#include <time.h>

void tzset ();
```

`tzset` sets the value of the internal variables used by `localtime` to the values specified in the environment variable `TZ`. It is called by `asctime`. In System V, it sets the value of the global variable `daylight`. `tzset` is a library function supplied with BSD and System V.4.

tzsetwall

```
#include <time.h>

void tzsetwall ();
```

`tzsetwall` sets the value of the internal variables used by `localtime` to the default values for the site. `tzsetwall` is a library function supplied with BSD and System V.4.

Suspending process execution

Occasionally you want to suspend process execution for a short period of time. For example, the *tail* program with the `-f` flag waits until a file has grown longer, so it needs to relinquish the processor for a second or two between checks on the file status.

Typically, this is done with `sleep`. However, some applications need to specify the length of time more accurately than `sleep` allows, so a couple of alternatives have arisen: `nap` suspends execution for a number of milliseconds, and `usleep` suspends it for a number of microseconds.

nap

`nap` is a XENIX variant of `sleep` with finer resolution:

```
#include <time.h>

long nap (long millisecs);
```

`nap` suspends process execution for at least `millisecs` milliseconds. In practice, the XENIX clock counts in intervals of 20 ms, so this is the maximum accuracy with which you can specify `millisecs`. You can simulate this function with `usleep` (see page 281 for more details).

setitimer

BSD systems and derivatives maintain three (possibly four) interval timers:

- A *real time* timer, `ITIMER_REAL`, which keeps track of real elapsed time.
- A *virtual* timer, `ITIMER_VIRTUAL`, which keeps track of process execution time, in other words the amount of CPU time that the process has used.
- A *profiler* timer, `ITIMER_PROF`, which keeps track of both process execution time and time spent in the kernel on behalf of the process. As the name suggests, it is used to implement profiling tools.

- A *real time profiler* timer, ITIMER_REALPROF, used for profiling Solaris 2.X multi-threaded processes.

These timers are manipulated with the system calls getitimer and setitimer:

```
#include <sys/time.h>

struct timeval
  {
  long  tv_sec;                         /* seconds */
  long  tv_usec;                        /* and microseconds */
  };

struct itimerval
  {
  struct  timeval it_interval;          /* timer interval */
  struct  timeval it_value;             /* current value */
  };

int getitimer (int which, struct itimerval *value);
int setitimer (int which, struct itimerval *value, struct itimerval *ovalue);
```

setitimer sets the value of a specific timer which to value, and optionally returns the previous value in ovalue if this is not a NULL pointer. getitimer just returns the current value of the timer to value. The resolution is specified to an accuracy of 1 microsecond, but it is really limited to the accuracy of the system clock, which is more typically in the order of 10 milliseconds. In addition, as with all timing functions, there is no guarantee that the process will be able to run immediately when the timer expires.

In the struct itimerval, it_value is the current value of the timer, which is decremented depending on type as described above. When it_value is decremented to 0, two things happen: a signal is generated, and it_value is reloaded from it_interval. If the result is 0, no further action occurs; otherwise the system continues decrementing the counter. In this way, one call to setitimer can cause the system to generate continuous signals at a specified interval.

The signal that is generated depends on the timer. Here's an overview:

Table 16−1: setitimer signals

Timer	Signal
Real time	SIGALRM
Virtual	SIGVTALRM
Profiler	SIGPROF
Real-time profiler[1]	SIGPROF

[1] Only Solaris 2.x

The only timer you're likely to see is the real time timer. If you don't have it, you can fake it with alarm. In System V.4, setitimer is implemented as a library function that calls an undocumented system call. See *The Magic Garden explained: The Internals of UNIX System*

V Release 4, by Berny Goodheart and James Cox, for more details.
`setitimer` is used to implement the library routine `usleep`.

sleep

```
#include <unistd.h>

unsigned sleep (u_int seconds);
```

The library routine `sleep` suspends program execution for approximately `seconds` seconds. It is available on all UNIX platforms.

usleep

`usleep` is a variant of `sleep` that suspends program execution for a very short time:

```
#include <unistd.h>
void usleep (u_int microseconds);
```

`usleep` sleeps for at least `microseconds` microseconds. It is supplied on BSD and System V.4 systems as a library function that uses the *setitimer* system call.

select and poll

If your system doesn't supply any timing function with a resolution of less than one second, you might be able to fake it with the functions `select` or `poll`. `select` can wait for nothing if you ask it to, and since the timeout is specified as a `struct timeval` (see page 270), you can specify times down to microsecond accuracy. You can use `poll` in the same way, except that you specifies its timeout value in milliseconds.

For example,

```
void usleep (int microseconds)
  {
  struct timeval timeout;
  timeout.tv_usec = microseconds % 1000000;
  timeout.tv_sec = microseconds / 1000000;
  select (0, NULL, NULL, NULL, &timeout);
  }

or
void usleep (int microseconds)
  {
  poll (0, NULL, microseconds / 1000);
  }
```