6 Clocks and Timers

Realtime applications must be able to operate on data within strict timing constraints in order to schedule application or system events. Timing requirements can be in response to the need for either high system throughput or fast response time. Applications requiring high throughput may process large amounts of data and use a continuous stream of data points equally spaced in time. For example, electrocardiogram research uses a continuous stream of data for qualitative and quantitative analysis.

Applications requiring a fast response to asynchronous external events must capture data as it comes in and perform decision-making operations or generate new output data within a given time frame. For example, flight simulator applications may acquire several hundred input parameters from the cockpit controls and visual display subsystem with calculations to be completed within a 5 millisecond time frame.

DIGITAL UNIX P1003.1b timing facilities allow applications to use relative or absolute time and to schedule events on a one-shot or periodic basis. Applications can create multiple timers for each process.

This chapter includes the following sections:

- Clock Functions, Section 6.1
- Types of Timers, Section 6.2
- Timers and Signals, Section 6.3
- Data Structures Associated with Timing Facilities, Section 6.4
- Timer Functions, Section 6.5
- High-Resolution Sleep, Section 6.6
- Clocks and Timers Example, Section 6.7

The correctness of realtime applications often depends on satisfying timing constraints. A systemwide clock is the primary source for synchronization and high-resolution timers to support realtime requirements for scheduling events. The P1003.1b timing functions perform the following tasks:

- Set a systemwide clock and obtain the current value of the clock
- Set per-process timers to expire once or multiple times (arm the timers)
- Use asynchronous signals on timer expiration
- Retrieve the resolution of the systemwide clock
- Permit the calling thread or process to suspend execution for a period of time or until a signal is delivered

Timing facilities are most useful when combined with other synchronization techniques.

Although non-POSIX functions are available for creating timers, application programmers striving for standards conformance, portability, and use of multiple per-process timers should use the P1003.1b timing facilities described in this chapter.

6.1 Clock Functions

The supported time-of-day clock is the CLOCK_REALTIME clock, defined in the `time.h` header file. The CLOCK_REALTIME clock is a systemwide clock, visible to all processes running on the system. If all processes could read the clock at the same time, each process would see the same value.

The CLOCK_REALTIME clock measures the amount of time that has elapsed since 00:00:00 January 1, 1970 Greenwich Mean Time (GMT). [Footnote 1]

The CLOCK_REALTIME clock measures time in nanoseconds; clock resolution does not reflect fractions of nanoseconds. For example, when the resolution for CLOCK_REALTIME is calculated at 1 sec / 1024 Hz, the result is 976562.5 nanoseconds. The clock resolution returned by the call to `clock_getres` for CLOCK_REALTIME is 976562. The fractional nanoseconds are ignored. The system self-corrects at the end of every second and adjusts time to correct for disparities. See Section 6.1.4 for more information about system clock resolution.

The P1003.1b timing functions for a specified clock are as follows:

Function	Description
`clock_getres`	Returns the resolution of the specified clock
`clock_gettime`	Returns the current value for the specified clock
`clock_settime`	Sets the specified clock to the specified value

Use the name CLOCK_REALTIME as the *clock_id* argument in all P1003.1b clock functions.

The `clock_getres` function returns the clock resolution. Note that you cannot set the resolution of the specified clock, although you can specify a high-resolution option that gives the appearance of higher resolution (see Section 6.1.5).

The values returned by the `clock_gettime` function can be used to determine values for the creation of realtime timers.

When the `clock_settime` function is called, the *time* argument is truncated to a multiple of the clock resolution, if it is not already a multiple of the clock resolution. Similarly, the clock resolution is used when setting interval timers.

The following example calls the `clock_getres` function to determine clock resolution:

```
#include <unistd.h>
#include <time.h>

main()
{
struct timespec     clock_resolution;
int stat;

stat = clock_getres(CLOCK_REALTIME, &clock_resolution);

printf("Clock resolution is %d seconds, %ld nanoseconds\n",
     clock_resolution.tv_sec, clock_resolution.tv_nsec);
}
```

6.1.1 Retrieving System Time

Both the `time` and `clock_gettime` functions return the value of the systemwide clock as the number of elapsed seconds since the Epoch. The `timespec` data structure (used for the `clock_gettime` function) also contains a member to hold the value of the number of elapsed nanoseconds not comprising a full second.

Example 6-1 shows the difference between the time as returned by the `time` and `clock_gettime` functions.

Example 6-1: Returning Time

```
#include <unistd.h>
#include <time.h>

main()
{
struct timespec ts;

      /* Call time */
printf("time returns %d seconds\n", time(NULL);)
       /* Call clock_gettime */

clock_gettime(CLOCK_REALTIME, &ts);
printf("clock_gettime returns:\n");
printf("%d seconds and %ld nanoseconds\n", ts.tv_sec, ts.tv_nsec);
}
```

In Example 6-1, 876,764,530 seconds is returned from the `time` function, and 876,764,530 seconds and 000,0674,633 nanoseconds is returned from the `clock_gettime` function.

The `time` function returns a long integer containing the number of seconds that have elapsed since the Epoch. The `clock_gettime` function receives a pointer to the `timespec` structure and returns the values in the *tv_sec* and *tv_nsec* members.

If you plan to write the current time to a device or file, you may want to convert the time format returned by the `clock_gettime` function.

6.1.2 Setting the Clock

The `clock_settime` function lets you set the time for the specified clock. If you have an application that monitors time over the network use the `clock_settime` function to synchronize with other systems. However, under normal circumstances you would not need to call the `clock_settime` function.

If timers are pending execution, use the `adjtime` function to adjust the clock slowly; armed timers are not affected by this function. Refer to the reference page for `adjtime` for complete information about this function.

You must have `superuser` privileges to use the `clock_settime` and `adjtime` functions.

6.1.3 Converting Time Values

Realtime clock and timer functions use the number of seconds and nanoseconds since the Epoch. Although this method is precise and suitable for the machine, it is not meaningful for application users. If your application prints or receives time information from users, you will want to convert time data to a more readable format.

If you use the `time` function to retrieve system time, the input and return values are expressed in elapsed seconds since the Epoch. Your application should define the format for both user input and output and then convert these time values for use by the program. Applications can store the converted time values for future use.

The C language provides a number of functions to convert and store time in both a `tm` structure and an ASCII format. Note that although these C routines use seconds as the smallest unit of time, they provide users with a readable format.

When you pass the time in seconds to these functions, some functions return a pointer to a `tm` structure. This structure breaks down time into units such as hours, minutes, and seconds, and stores the data in the appropriate fields.

DIGITAL UNIX provides date and time functions that deal with these time units and calendar time, making conversions as necessary. The date and time conversion functions are as follows:

Function	**Description**
`asctime`	Converts time units (hours, minutes, and seconds) into a 26-character string
`ctime`	Converts a time in seconds since the Epoch to an ASCII string in the form generated by `asctime`
`difftime`	Computes the difference between two calendar times (time1-time0) and returns the difference expressed in seconds
`gmtime`	Converts a calendar time into time units, expressed as GMT
`localtime`	Converts a time in seconds since the Epoch into time units
`mktime`	Converts the time units in the `tm` structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by `time`
`tzset`	Sets the external variable *tzname*, which contains current time zone names

To select the most appropriate time conversion function for your application, refer to the reference pages for each of these functions.

The converted time values for the date and time conversion functions are placed in a time structure (`tm`) defined in the `time.h` header file, as follows:

```
struct tm  {
       int tm_sec,                 /* Time in seconds (0-59)          */
           tm_min,                 /* Time in minutes (0-59)          */
           tm_hour,                /* Time in hours  (0-23)           */
           tm_mday,                /* Day of the month (1 to 31)      */
           tm_mon,                 /* Month (0 to 11)                 */
           tm_year,                /* Year (last 2 digits)            */
           tm_wday,                /* Day of the week (Sunday=0)      */
           tm_yday,                /* Day of the year (0 to 365)      */
           tm_isdst;               /* Daylight savings time (always 0) */
      long tm_gmtoff;              /* Offset from GMT in seconds      */
      char *tm_zone                /* Time zone                       */
     };
```

6.1.4 System Clock Resolution

System clock resolution on DIGITAL Alpha systems is 1/1024 second, or roughly 976 microseconds. The system maintains time by adding 976 microseconds at every clock interrupt. The actual time period between clock ticks is exactly 1/1024 second = 976.5625 microseconds.

The missing 576 microseconds (1024 * .5625) are added at the end of the 1024th tick (that is, every second), to make sure that the system time matches with the observed wall-clock time.

This implies that each clock tick increments the system time by 976 microseconds except the 1024th one, which advances the time by 1552 microseconds (976 + 576). Thus there is a spike in the time as maintained by DIGITAL UNIX.

The POSIX 1003.1a specification mandates that the system quantize all timer values passed by a program to the next multiple of the clock tick. If an application program requests a timer value that is not an exact multiple of the system clock resolution (an exact multiple of 976.5625 microseconds), the actual time period counted down by the system will be slightly larger than the requested time period.

A program that asks for a periodic timer of 50 milliseconds will actually get a time period of 50.78 milliseconds (.976562 * 52). Unless accounted for, the additional .78 milliseconds every 50 milliseconds will result in a wrong calculation of the elapsed time as calculated by the program.

Possible solutions to the above anomaly are to either always ask for time periods that are integral multiples of the system clock resolution, or to not use the periodic timer for the purpose of time keeping.

6.1.5 High-Resolution Clock

Version 4.0 of DIGITAL UNIX added the capability of an optional high-resolution clock. To enable the high-resolution clock, add the following line to the kernel configuration file and rebuild the kernel:

```
options MICRO_TIME
```

The system clock (CLOCK_REALTIME) resolution as returned by `clock_getres(3)` will not change; timer resolution remains the same. However, time as returned by the `clock_gettime(3)` routine will now be extrapolated between the clock ticks. The granularity of the time returned will now be in microseconds. The time values returned are SMP safe, monotonically increasing, and have 1 microsecond as the apparent resolution.

The high-resolution clock can be used for time stamping and for measuring events which are of the order of microseconds, such as time spent in some critical code path.

6.2 Types of Timers

Two types of timers are provided to support realtime timing facilities: one-shot timers and periodic timers. Timers can be set up to expire only once (one-shot) or on a repetitive (periodic) schedule. A one-shot timer is armed with an initial expiration time, expires only once, and then is disarmed. A timer becomes a periodic timer with the addition of a repetition value. The timer expires, then loads the repetition interval, rearming the timer to expire after the repetition interval has elapsed.

The initial expiration value can be relative to the current time or an absolute time value. A relative timer has an initial expiration time based on the amount of time elapsed, such as 30 seconds from the start of the application or 0.5 seconds from the last timer expiration. An absolute timer expires at a calendar date and time.

Often, a timer uses both concepts of absolute and relative timers. You can establish a timer to fire as an absolute timer when it first expires, and set subsequent timer expirations relative to the first expiration. For example, an

application may need to collect data between midnight and 3:00 AM. Data collection during this three-hour period may be staged in 12-minute intervals. In this case, absolute times are used to start and stop the data collection processes at midnight and 3:00 AM respectively. Relative time is used to initiate data collection at 12-minute intervals.

The values specified in the arguments to the `timer_settime` function determine whether the timer is a one-shot or periodic and absolute or relative type. Refer to Section 6.5.2 for more information on the `timer_settime` function.

6.3 Timers and Signals

You create a timer with the `timer_create` function, which is associated with a `sigevent` structure. When using timers, you specify an initial expiration value and an interval value. When the timer expires, the system sends the specified signal to the process that created the timer. Therefore, you should set up a signal handler to catch the signal after it is sent to the calling process.

To use signals with timers, include the following steps in your application:

1. Create and declare a signal handler.
2. Set the `sigevent` structure to specify the signal you want sent on timer expiration.
3. Establish a signal handler with the `sigaction` function.
4. Create the timer.

If you do not choose to use realtime signals, then identical signals delivered from multiple timers are compressed into a single signal. In this case, you may need to specify a different signal for each timer. If you use realtime signals, identical signals are queued to the calling process. Refer to Chapter 5 for more information on signals and signal handling.

6.4 Data Structures Associated with Timing Facilities

The `timespec` and `itimerspec` data structures in the `timers.h` header file are used in many of the P1003.1b realtime clock and timer functions. The `timespec` data structure contains members for both second and nanosecond values. This data structure sets up a single time value and is used by many P1003.1b functions that accept or return time value specifications. The `itimerspec` data structure contains two `timespec` data structures. This data structure sets up an initial timer and repetition value used by P1003.1b timer functions.

The `signal.h` header file contains a `sigevent` structure for specifying the signal to be sent on timer expiration.

6.4.1 Using the timespec Data Structure

The `timespec` data structure consists of two members, *tv_sec* and *tv_nsec*, and takes the following form:

```
typedef struct timespec {
     time_t tv_sec;                /* Seconds     */
     long   tv_nsec;               /* Nanoseconds */
} timespec_t;
```

The *tv_nsec* member is valid only if its value is greater than zero and less than the number of nanoseconds in a second. The time interval described by the `timespec` structure is (*tv_sec* * 10^9) + *tv_nsec* nanoseconds. (The minimum possible time interval is limited by the resolution of the specified clock.)

The `timespec` structure is used in P1003.1b functions to set and return the specified clock, return the resolution of the clock, set and return timer values, and specify `nanosleep` values.

6.4.2 Using the itimerspec Data Structure

The `itimerspec` data structure consists of two `timespec` structures and takes the following form:

```
struct itimerspec {
       struct timespec it_interval;   /* Timer interval     */
       struct timespec it_value;      /* Initial expiration */
};
```

The two `timespec` structures specify an interval value and an initial expiration value, both of which are used in all timer functions related to setting up timers. The values specified for the member structures identify the timer as one-shot or periodic. Table 6-1 summarizes the ways that values for the two members of the `itimerspec` structure are used to specify timers.

Table 6-1: Values Used in Setting Timers

Member	Zero	Non-Zero
it_value	No expiration value Disarm the timer	Expiration value Arm the timer
it_interval	No reload value Use as a one-shot timer	Interval reload value Use as a periodic timer

The *it_value* specifies the initial amount of time before the timer expires. A nonzero value for the *it_value* member indicates the amount of time until the timer's first expiration.

TIMER_ABSTIME is a flag which, when set, makes the timer an absolute timer. The time until the next timer expiration is specified in seconds and nanoseconds since the Epoch and is the difference between the absolute time specified by the *it_value* member and the current clock value.

If the TIMER_ABSTIME flag is not set, the time until the next timer expiration is set equal to the interval specified by the *it_value* member, and the timer is a relative timer.

A zero value for the *it_value* member disarms the timer.

Once the timer expires for the first time, the *it_interval* member specifies the interval after which the timer will expire again. That is, the value of the *it_interval* member is reloaded when the timer expires and timing continues. A nonzero value for the *it_interval* member specifies a periodic timer. A zero value for the *it_interval* member causes the timer to expire only once; after the first expiration the *it_value* member is set to zero and the timer is disarmed.

For example, to specify a timer that executes only once, 5.25 seconds from now, specify the following values for the members of the `itimerspec` structure:

```
mytimer.it_value.tv_sec = 5;
mytimer.it_value.tv_nsec = 250000000;
```

```
mytimer.it_interval.tv_sec = 0;
mytimer.it_interval.tv_nsec = 0;
```

To arm a timer to execute 15 seconds from now and then at 0.5 second intervals, specify the following values:

```
mytimer.it_value.tv_sec = 15;
mytimer.it_value.tv_nsec = 0;
mytimer.it_interval.tv_sec = 0;
mytimer.it_interval.tv_nsec = 500000000;
```

In the preceding examples, the timer is armed relative to the current time. To set up a timer with an absolute initial expiration time, such as 10:00 AM, convert the absolute initial expiration value (in seconds and nanoseconds) to the correct offset from the current time.

Because the value of the *tv_nsec* member is expressed in nanoseconds, it may be somewhat cumbersome. To simplify specifying values for the *tv_nsec* member as fractions of a second, you could define a symbolic constant:

```
#define NSECS_PER_SEC 1000000000;
```

After defining this constant, you could specify 1/4 second as follows:

```
mytimer.it_value.tv_nsec = NSECS_PER_SEC/4;
```

See Section 6.5 for more information on relative and absolute timers.

6.4.3 Using the sigevent Data Structure

The `sigevent` structure delivers the signal on timer expiration. The *evp* argument of the `timer_create` function points to a *sigevent* structure, which contains the signal to be sent upon expiration of each timer.

The *sigevent* structure is defined in the `signal.h` header file and contains the following members:

```
union sigval   sigev_value;   /* Application-defined value */
int            sigev_signo;   /* Signal to raise */
int            sigev_notify;  /* Notification type */
```

The *sigval* union contains at least the following members:

```
int     sival_int;   /* Used when sigev_value is of type int */
void    *sival_ptr;  /* Used when sigev_value is of type ptr */
```

The *sigev_value* member is an application-defined value to be passed to the signal-catching function at the time of signal delivery.

The *sigev_signo* member specifies the signal number to be sent on completion of the asynchronous I/O operation or on timer expiration. In both instances, you must set up a signal handler to execute when the signal is received. You can use the `sigaction` function to specify the action required. Refer to Chapter 5 for more information about the `sigaction` function.

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. There are two values defined for *sigev_notify* in P1003.1b: SIGEV_NONE and SIGEV_SIGNAL. SIGEV_NONE indicates that no asynchronous notification is delivered when an event occurs. SIGEV_SIGNAL indicates that a queued signal with an application-defined value is delivered when an event occurs.

6.5 Timer Functions

Clocks and timers allow an application to synchronize and coordinate activities according to a user-defined schedule. DIGITAL UNIX P1003.1b timers have the ability to issue periodic timer requests initiated by a single call from the application.

The following P1003.1b timing functions are available for realtime applications:

Function	Description
`timer_create`	Returns a unique timer ID used in subsequent calls to identify a timer based on the systemwide clock
`timer_delete`	Removes a previously allocated, specified timer
`timer_getoverrun`	Returns the timer expiration overrun count for the specified timer
`timer_gettime`	Returns the amount of time before the specified timer is due to expire and the repetition value
`timer_settime`	Sets the value of the specified timer either to an offset from the current clock setting or to an absolute value

Timers do not have global IDs, which means that they are not inherited by a child process after a call to the `fork` or `exec` system calls. You cannot arm a timer, call the `exec` system call, and have the new image receive the signal. The newly created timer structures are inherited across a fork, but any pending timer signals will be delivered only to the parent process.

6.5.1 Creating Timers

The `timer_create` function allocates a timer and returns a timer ID that is unique within the calling process and exists for the life of that timer. The timer is not armed until you make a call to the `timer_settime` function, which sets the values for the specified timer.

The timer functions perform a series of tasks necessary for setting up timers. To create a timer, you must set up appropriate data structures, set up a signal handler to catch the signal when the timer expires, and arm the timer. To use timers in a realtime application, follow these steps:

1. Include `time.h` and `signal.h` in the application source file.
2. Declare the variable names for your `itimerspec` data structure to specify interval and expiration values.
3. Establish a `sigevent` structure containing the signal to be passed to the process on timer expiration.
4. Set up a signal handler in the calling process to catch the signal when the timer expires.
5. Call the `timer_create` function to create a timer and associate it with the specified clock. Specify a signal to be delivered when the timer expires.
6. Initialize the `itimerspec` data structure with the required values.
7. Call the `timer_settime` function to initialize and activate the timer as either an absolute or relative timer.
8. Call the `timer_delete` function when you want to remove the timer.

The number of per-process timers (TIMER_MAX) is defined in the `limits.h` header file.

The `timer_create` function also takes an *evp* argument which, if non-NULL, is a pointer to a `sigevent` structure. This structure defines the signal and value to be sent to the calling process when the timer expires. If the *sigev_notify* member of *evp* is SIGEV_SIGNAL, the structure must contain the signal number and data value to send to the process when the timer expires. If the *sigev_notify* member is SIGEV_NONE, no notification will be sent.

If the *evp* argument is NULL, the default signal SIGALRM is used.

6.5.2 Setting Timer Values

The `timer_settime` function determines whether the timer is an absolute or relative timer. This function sets the initial expiration value for the timer as well as the interval time used to reload the timer once it has reached the initial expiration value. The interval you specify is rounded up to the next integral multiple of the system clock resolution. See Section 6.1.4 for more information about system clock resolution.

The arguments for the `timer_settime` function perform the following functions:

1. The *timerid* argument identifies the timer.
2. The *flags* argument determines whether the timer behaves as an absolute or relative timer.

   If the TIMER_ABSTIME flag is set, the timer is set with a specified starting time (the timer is an absolute timer). If the TIMER_ABSTIME flag is not set, the timer is set relative to the current time (the timer is a relative timer).
3. The *value* argument points to an `itimerspec` structure, which contains the initial expiration value and repetition value for the timer:

   - The *it_value* member of the *value* argument establishes the initial expiration time.

     For absolute timers, the `timer_settime` function interprets the next expiration value as equal to the difference between the absolute time specified by the *it_value* member of the *value* argument and the current value of the specified clock. The timer then expires when the clock reaches the value specified by the *it_value* member of the *value* argument.

     For relative timers, the `timer_settime` function interprets the next expiration value as equal to the interval specified by the *it_value* member of the *value* argument. The timer will expire in *it_value* seconds and nanoseconds from when the call was made. After a timer is started as an absolute or relative timer, its behavior is driven by whether it is a one-shot or periodic timer.

   - The *it_value* member of the *value* argument can disable a timer.

     To disable a periodic timer, call the timer and specify the value zero for the *it_value* member.

   - The *it_interval* member of the *value* argument establishes the repetition value.

     The timer interval is specified as the value of the *it_interval* member of the `itimerspec` structure in the *value* argument. This value determines whether the timer functions as a one-shot or periodic timer.

     After a one-shot timer expires, the expiration value (*it_value* member) is set to zero. This indicates that no next expiration value is specified, which disarms the timer.

A periodic timer is armed with an initial expiration value and a repetition interval. When the initial expiration time is reached, it is reloaded with the repetition interval and the timer starts again. This continues until the application exits. To arm a periodic timer, set the *it_value* member of the *value* argument to the desired expiration value and set the *it_interval* member of the *value* argument to the desired repetition interval.

4. The *ovalue* argument points to an `itimerspec` structure that contains the time remaining on an active timer. If the timer is not armed, the *ovalue* is equal to zero. If you delete an active timer, the *ovalue* will contain the amount of time remaining in the interval.

You can use the `timer_settime` function to reuse an existing timer ID. If a timer is pending and you call the `timer_settime` function to pass in new expiration times, a new expiration time is established.

6.5.3 Retrieving Timer Values

The `timer_gettime` function returns two values: the amount of time before the timer expires and the repetition value set by the last call to the `timer_settime` function. If the timer is disarmed, a call to the timer with the `timer_gettime` function returns a zero for the value of the *it_value* member. To arm the timer again, call the `timer_settime` function for that timer ID and specify a new expiration value for the timer.

6.5.4 Getting the Overrun Count

Under POSIX.1b, timer expiration signals for a specific timer are not queued to the process. If multiple timers are due to expire at the same time, or a periodic timer generates an indeterminate number of signals with each timer request, a number of signals will be sent at essentially the same time. There may be instances where the requesting process can service the signals as fast as they occur, and there may be other situations where there is an overrun of the signals.

The `timer_getoverrun` function helps track whether or not a signal was delivered to the calling process. DIGITAL UNIX P1003.1b timing functions keep a count of timer expiration signals for each timer created. The `timer_getoverrun` function returns the counter value for the specified timer ID. If a signal is sent, the overrun count is incremented, even if the signal was not delivered or if it was compressed with another signal. If the signal cannot be delivered to the calling process or if the signal is delayed for some reason, the overrun count contains the number of extra timer expirations that occurred during the delay. A signal may not be delivered if, for instance, the signal is blocked or the process was not scheduled. Use the `timer_getoverrun` function to track timer expiration and signal delivery as a means of determining the accuracy or reliability of your application.

If the signal is delivered, the overrun count is set to zero and remains at zero until another overrun occurs.

6.5.5 Disabling Timers

When a one-shot timer expires, the timer is disarmed but the timer ID is still valid. The timer ID is still current and can be rearmed with a call to the `timer_settime` function. To remove the timer ID and disable the timer, use the `timer_delete` function.

6.6 High-Resolution Sleep

To suspend process execution temporarily using the P1003.1b timer interface, call the `nanosleep` function. The `nanosleep` function suspends execution for a specified number of nanoseconds, providing a high-resolution

sleep. A call to the `nanosleep` function suspends execution until either the specified time interval expires or a signal is delivered to the calling process.

Only the calling thread sleeps with a call to the `nanosleep` function. In a threaded environment, other threads within the process continue to execute.

The `nanosleep` function has no effect on the delivery or blockage of signals. The action of the signal must be to invoke a signal-catching function or to terminate the process. When a process is awakened prematurely, the *rmtp* argument contains the amount of time remaining in the interval.

6.7 Clocks and Timers Example

Example 6-2 demonstrates the use of P1003.1b realtime timers. The program creates both absolute and relative timers. The example demonstrates concepts using multiple signals to distinguish between timer expirations. The program loops continuously until the program is terminated by a Ctrl/C from the user.

Example 6-2: Using Timers

```
/*
 * The following program demonstrates the use of various types of
 * POSIX 1003.1b Realtime Timers in conjunction with 1003.1 Signals.
 *
 * The program creates a set of timers and then blocks waiting for
 * either timer expiration or program termination via SIGINT.
 * Pressing CTRL/C after a number of seconds terminates the program
 * and prints out the kind and number of signals received.
 *
 * To build:
 *
 * cc -g3 -O -non_shared -o timer_example timer_example.c -L/usr/ccs/lib -lrt
 */

#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/limits.h>
#include <time.h>
#include <sys/signal.h>
#include <sys/errno.h>

/*
 * Constants and Macros
 */

#define FAILURE -1
#define ABS     TIMER_ABSTIME
#define REL     0
#define TIMERS  3

#define MIN(x,y) (((x) < (y)) ? (x) : (y))

sig_handler();
void timeaddval();
struct sigaction sig_act;

/*
 * Control Structure for Timer Examples
 */
struct timer_definitions {
```

```
        int type;                       /* Absolute or Relative Timer */
        struct sigevent evp;            /* Event structure */
        struct itimerspec timeout;      /* Timer interval */
};

/*
 * Initialize timer_definitions array for use in example as follows:
 *
 *   type, { sigev_value, sigev_signo }, { it_iteration, it_value }
 */

struct timer_definitions timer_values[TIMERS] = {
        { ABS, {0,SIGALRM}, {0,0, 3,0} },
        { ABS, {0,SIGUSR1}, {0,500000000, 2,0} },
        { REL, {0,SIGUSR2}, {0,0, 5,0} }
};

timer_t timerid[TIMERS];
int timers_available;                   /* number of timers available */
volatile int alrm, usr1, usr2;
sigset_t mask;

main()
{
        int status, i;
        int clock_id = CLOCK_REALTIME;
        struct timespec current_time;

        /*
         *  Initialize the sigaction structure for the handler.
         */

        sigemptyset(&mask);
        sig_act.sa_handler = (void *)sig_handler;
        sig_act.sa_flags = 0;
        sigemptyset(&sig_act.sa_mask);
        alrm = usr1 = usr2 = 0;

        /*
         *  Determine whether it's possible to create TIMERS timers.
         *  If not, create TIMER_MAX timers.
         */

        timers_available = MIN(sysconf(_SC_TIMER_MAX),TIMERS);

        /*
         * Create "timer_available" timers, using a unique signal
         * type to denote the timer's expiration. Then initialize
         * a signal handler to handle timer expiration for the timer.
         */

        for (i = 0; i < timers_available; i++) {
                status = timer_create(clock_id, &timer_values[i].evp,
                                      &timerid[i]);
                if (status == FAILURE) {
                        perror("timer_create");
                        exit(FAILURE);
                }
                sigaction(timer_values[i].evp.sigev_signo, &sig_act, 0);
        }

        /*
         * Establish a handler to catch CTRL-c and use it for exiting.
         */
```

```
        sigaction(SIGINT, &sig_act, NULL);      /* catch crtl-c */

        /*
         * Queue the following Timers: (see timer_values structure for details)
         *
         *   1.  An absolute one shot timer  (Notification is via SIGALRM).
         *   2.  An absolute periodic timer. (Notification is via SIGUSR1).
         *   3.  A relative one shot timer.  (Notification is via SIGUSR2).
         *
         * (NOTE: The number of TIMERS queued actually depends on
         *  timers_available)
        */

        for (i = 0; i < timers_available; i++) {
                if (timer_values[i].type == ABS) {
                        status = clock_gettime(CLOCK_REALTIME, &current_time);
                        timeaddval(&timer_values[i].timeout.it_value,
                                &current_time);
                }
                status = timer_settime(timerid[i], timer_values[i].type,
                        &timer_values[i].timeout, NULL);
                if (status == FAILURE) {
                        perror("timer_settime failed: ");
                        exit(FAILURE);
                }
        }

        /*
         * Loop forever.  The application will exit in the signal handler
         * when a SIGINT is issued (CRTL/C will do this).
         */

        for(;;) pause();
}

/*
 *  Handle Timer expiration or Program Termination.
 */

sig_handler(signo)
int signo;
{
        int i, status;

        switch (signo) {
            case SIGALRM:
                alrm++;
                break;
            case SIGUSR1:
                usr1++;
                break;
            case SIGUSR2:
                usr2++;
                break;
            case SIGINT:
                for (i = 0; i < timers_available; i++)  /* delete timers */
                        status = timer_delete(timerid[i]);
                printf("ALRM: %d, USR1: %d, USR2: %d\n", alrm, usr1, usr2);
                exit(1);  /* exit if CRTL/C is issued */
        }
        return;
}

/*
 * Add two timevalues: t1 = t1 + t2
```

```
 */

void timeaddval(t1, t2)
struct timespec *t1, *t2;
{
        t1->tv_sec += t2->tv_sec;
        t1->tv_nsec += t2->tv_nsec;
        if (t1->tv_nsec < 0) {
                t1->tv_sec--;
                t1->tv_nsec += 1000000000;
        }
        if (t1->tv_nsec >= 1000000000) {
                t1->tv_sec++;
                t1->tv_nsec -= 1000000000;
        }
}
```
