# Tutorial 4 – Timers

## Aim

Contiki is an event based operating system that designed to operate on resource scarce and low power embedded system platforms. This tutorial shows you how to use the timer library of contiki.

- **Contiki Concepts**

1. Clock Module
2. Timer
3. Stimer
4. Etimer
5. Ctimer
6. Rtimer

## Timers

Contiki provides a set of timer libraries that are used both by application programs and by the Contiki kernel. The timer libraries contain functionality for checking if a time period has passed, waking up the system from low power mode at scheduled times, and real-time task scheduling. The timers are also used by applications to let the system work with other things or enter low power mode for a time period before resuming execution.

Contiki has one clock module and a set of timer modules: **timer, stimer, ctimer, etimer,** and **rtimer**. The different timer modules have different uses: some provide long-running timers with low granularity, some provide a high granularity but with a short range, some can be used in interrupt contexts (rtimer) others cannot.

## Example code:

The timers example code  consists of 4 processes and 4 timers ((etimer, stimer, ctimer and timer).
Location: 'contiki-examples/timers'

### Clock module

The clock module provides functionality to handle the *system time* and also to block the CPU for short time periods. The timer libraries are implemented with the functionality of the clock module as base.

The API for the Contiki clock module consists of:

`clock_time()` returns the current system time in clock ticks. The number of clock ticks per second is platform dependent and is specified with the constant CLOCK_SECOND.

`clock_seconds()` for getting the system time in seconds as an unsigned long and this

Acknowledgement: Dr. Matt D' Souza (University of Queensland), Dr. Brano Kusy (CSIRO)

time value can become much larger before it wraps around. The system time starts from zero when the Contiki system starts.

`clock_delay()` which blocks the CPU for a specified delay

`clock_wait()` which blocks the CPU for a specified number of clock ticks.

`clock_init()` is called by the system during the boot-up procedure to initialize the clock module. You must **NOT** call `clock_init()` in your program, as it is called by the main function.

```
clock_time_t clock_time(); // Get the system time.
unsigned long clock_seconds(); // Get the system time in seconds.
void clock_delay(unsigned int delay); // Delay the CPU.
void clock_wait(int delay); // Delay the CPU for a number of clock ticks.
void clock_init(void); // Initialize the clock module.
CLOCK_SECOND; // The number of ticks per second.
```

## Timer

The **timer** library provides the simplest form of timers and is used to check if a time period has passed. Timers use **system clock ticks**.

The Contiki timer library provides functions for setting, resetting and restarting timers, and for checking if a timer has expired. An application must "manually" check if its timers have expired; this is not done automatically. The timer library use `clock_time()` in the clock module to get the current system time.

The API for the Contiki timer library is:

`totimer_set()` which sets the timer to expire the specified delay from current time and also stores the time interval in the timer.

`timer_reset()` can then be used to restart the timer from previous expire time.

`timer_restart()` to restart the timer from current time.

`timer_set()` is called by both `timer_reset()` and `timer_restart()`. The difference between these functions is that `timer_reset()` set the timer to expire at exactly the same time interval while `timer_restart()` set the timer to expire some time interval from current time, thus allowing time drift.

`timer_expired()` is used to determine if the timer has expired.

`timer_remaining()` to get an estimate of the remaining time until the timer expires. The return value is undefined if the timer already has expired.

The timer library can safely be used from interrupts. The code example below shows a simple example how a timer can be used to detect timeouts in an interrupt.

*The Timer Library API:*

```
void timer_set(struct timer *t, clock_time_t interval); // Start the timer.
void timer_reset(struct timer *t); // Restart the timer from the previous
expiration time.
void timer_restart(struct timer *t); // Restart the timer from current time.
int timer_expired(struct timer *t); // Check if the timer has expired.
clock_time_t timer_remaining(struct timer *t); // Get the time until the timer
expires.
```

*A pseudo code example showing how a timer can be used to detect timeouts:*

```
#include "sys/timer.h"
static struct timer rxtimer;

void init(void) {

        timer_set(&rxtimer, CLOCK_SECOND / 2);

}

interrupt(UART1RX_VECTOR)  uart1_rx_interrupt(void)  {

        if (timer_expired(&rxtimer)) {      /* Timeout */

                /* ... */
        }

        timer_restart(&rxtimer);

        /* ... */
}
```

## Stimer (Second Timer)

The **stimer** library provides the simplest form of timers and are used to check if a time period has passed. The applications need to ask the timers if they have expired. **Stimers** use **seconds**.

The API for the Contiki stimer library is shown below and it is similar to the timer library. The difference is that times are specified as seconds instead of clock ticks. The stimer library can safely be used from interrupts.

*The stimer Library API:*

```
void stimer_set(struct stimer *t, unsigned long interval); // Start the timer.
void stimer_reset(struct stimer *t); // Restart the stimer from the previous
expiration time.
```

```
void stimer_restart(struct stimer *t); // Restart the stimer from current time.
int stimer_expired(struct stimer *t); // Check if the stimer has expired.
unsigned long stimer_remaining(struct stimer *t); // Get the time until the timer
expires.
```

## Etimer

The **etimer** library provides event timers and are used to schedule events to Contiki processes after a period of time. They are used in Contiki processes to wait for a time period while the rest of the system can work or enter low power mode.

An event timer will post the event PROCESS_EVENT_TIMER to the process that set the timer when the event timer expires. The etimer library use clock_time() in the clock module to get the current system time.

The API for the Contiki etimer library:

etimer_set() which sets the timer to expire the specified delay from current time.

etimer_reset() can then be used to restart the timer from previous expire time andetimer_restart() to restart the timer from current time, both using the same time interval that was originally set by etimer_set(). The difference between etimer_reset() andetimer_restart() is that the former schedules the timer from previous expiration time while the latter schedules the timer from current time thus allowing time drift.

etimer_stop() An event timer can be stopped by a call to  which means it will be immediately expired without posting a timer event.

etimer_expired() is used to determine if the event timer has expired.

Note that the timer event is sent to the Contiki process used to schedule the event timer. If an event timer should be scheduled from a callback function or another Contiki process, PROCESS_CONTEXT_BEGIN() and PROCESS_CONTEXT_END() can be used to temporary change the process context. See the processes tutorial for more information about process management.

Below is a simple example how an etimer can be used to schedule a process to run once per second.

*The etimer Library API:*

```
void etimer_set(struct etimer *t, clock_time_t interval); // Start the timer.
void etimer_reset(struct etimer *t); // Restart the timer from the previous
expiration time.
void etimer_restart(struct etimer *t); // Restart the timer from current time.
void etimer_stop(struct etimer *t); // Stop the timer.
int etimer_expired(struct etimer *t); // Check if the timer has expired.
int etimer_pending(); // Check if there are any non-expired event timers.
clock_time_t etimer_next_expiration_time(); // Get the next event timer expiration
time.
void etimer_request_poll(); // Inform the etimer library that the system clock has
changed.
```

*Setup an event timer to let a process execute once per second.*

```
#include "sys/etimer.h"    PROCESS_THREAD(example_process, ev, data)  {    static
struct etimer et;    PROCESS_BEGIN();      /* Delay 1 second */    etimer_set(&et,
CLOCK_SECOND);      while(1) {      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
/* Reset the etimer to trig again in 1 second */      etimer_reset(&et);      /* ...
*/    }    PROCESS_END();  }
```

## Ctimer

The **ctimer** library provides callback timers and are used to schedule calls to callback functions after a period of time. Like event timers, they are used to wait for some time while the rest of the system can work or enter low power mode. Since the callback timers call a function when a timer expires, they are especially useful in any code that do not have an explicit Contiki process such as protocol implementations. The callback timers are, among other things, used in communication stacks to handle timeouts.

The API for the Contiki ctimer library is similar to the etimer. The difference is that `ctimer_set()` takes a callback function pointer and a data pointer as arguments. When a ctimer expires, it will call the callback function with the data pointer as argument. The code example below shows how a ctimer can be used to schedule a callback to a function once per second.

Note that although the callback timers are calling a specified callback function, the process context for the callback is set to the process used to schedule the ctimer. Do not assume any specific process context in the callback unless you are sure about how the callback timers are scheduled.

The ctimer library can not safely be used from interrupts.

*The etimer Library API:*

```
void ctimer_set(struct ctimer *c, clock_time_t t, void(*f)(void *), void *ptr); //
Start the timer.
void ctimer_reset(struct ctimer *t); // Restart the timer from the previous
expiration time.
void ctimer_restart(struct ctimer *t); // Restart the timer from current time.
void ctimer_stop(struct ctimer *t); // Stop the timer.
```

```
int ctimer_expired(struct ctimer *t); // Check if the timer has expired.
```

*Setup a ctimer to call a function once per second:*

```
#include "sys/ctimer.h"

static struct ctimer timer;

static void  callback(void *ptr)  {
        ctimer_reset(&timer);
        /* ... */
}

void  init(void)  {
        ctimer_set(&timer, CLOCK_SECOND, callback, NULL);
}
```

### Rtimer

The **rtimer** library provides scheduling of real-time tasks. The **rtimer** library pre-empt any running Contiki process in order to let the real-time tasks execute at the scheduled time. The real-time tasks are used in time critical code such as the X-MAC wireless communication protocol implementation where the radio needs to be turned on or off at scheduled times without delay.

Rtimer is only used for low level critical sections of contiki driver code. You should **NOT** use rtimer in your contiki process code.

**Example**: contiki-examples/timers/all-timers.c

## Conclusion

This lesson has introduced the timer libraries of the Contiki operating system. Refer to ex1_timer example and to the Contiki documentation for more information.

## Related Documentation

- Contiki Wiki
- Contiki Timers Wiki

Acknowledgement: Dr. Matt D' Souza (University of Queensland), Dr. Brano Kusy (CSIRO)