Tick-tock - Understanding the Neutrino micro kernel's concept of time, Part II

Authored by: **Mario Charest, with Brian Stecher's assistance**
Updated by: **Thomas Fletcher**

This discussion is a follow-up to the "Tick-tock: Understanding the Neutrino micro kernel's concept of time" discussion. Make sure you read it first - to keep in the right frame of mind.

The hardware timer of the PC has another side effect when it comes to dealing with timers. Brian's article explains the behavior of "sleep" related functions. Timers are similarly affected by the design of the PC hardware.

Let's jump into the heart of the matter with some C code. The following is a working sample for QNX Neutrino. The same principles apply to QNX4 as well.

```
// --- INCLUDE FILE

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/neutrino.h>
#include <sys/netmgr.h>
#include <sys/syspage.h>

// --- FUNCTION (from Bottom to Top, and/or sorted)

int main( int argc, char *argv[] )
{
    int pid;
    int chid;
    int pulse_id;
    timer_t timer_id;
    struct sigevent event;
    struct itimerspec timer;
    struct _clockperiod clkper;
    struct _pulse pulse;
    _uint64 last_cycles=-1;
    _uint64 current_cycles;
    float cpu_freq;
    time_t start;

    // --- Get CPU frequency in order to do precise time calculation
    cpu_freq =  SYSPAGE_ENTRY( qtime )->cycles_per_sec;

    // --- Set priority to max so we don't get disrupted but anything
    // --- else then interrupts
    {
        struct sched_param param;
        int ret;
        param.sched_priority = sched_get_priority_max( SCHED_RR );
        ret = sched_setscheduler( 0, SCHED_RR, &param);
        assert ( ret != -1 );
    }
```

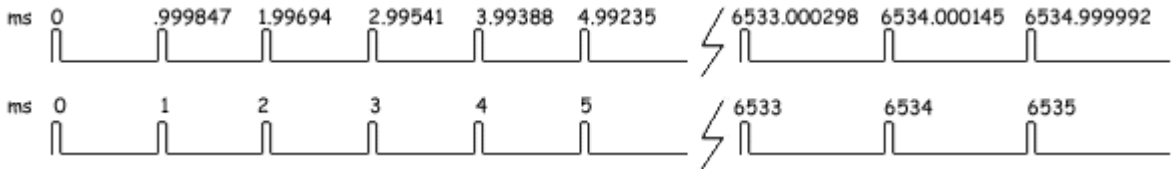
```
    // --- Create channel to receive timer event
    chid = ChannelCreate( 0 );
    assert ( chid != -1 );

    // --- setup timer and timer event
    event.sigev_notify           = SIGEV_PULSE;
    event.sigev_coid             = ConnectAttach ( ND_LOCAL_NODE, 0,
chid, 0, 0 );
    event.sigev_priority         = getprio(0);
    event.sigev_code             = 1023;
    event.sigev_value.sival_ptr  = (void*)pulse_id;

    assert ( event.sigev_coid != -1 );

    if ( timer_create(  CLOCK_REALTIME, &event, &timer_id ) == -1 )
    {
        perror ( "can't create timer" );
        exit( EXIT_FAILURE );
    }

    // --- change timer request to alter behavior
#if 1
      timer.it_value.tv_sec        = 0;
      timer.it_value.tv_nsec       = 1000000;
      timer.it_interval.tv_sec     = 0;
      timer.it_interval.tv_nsec    = 1000000;
#else
      timer.it_value.tv_sec        = 0;
      timer.it_value.tv_nsec       = 999847;
      timer.it_interval.tv_sec     = 0;
      timer.it_interval.tv_nsec    = 999847;
#endif

    // --- start timer
    if ( timer_settime( timer_id, 0, &timer, NULL ) == -1 )
    {
        perror("Can't start timern");
        exit( EXIT_FAILURE );
    }

    // --- set tick to 1ms otherwise if left to 10 ms default it
    // --- would take 65 seconds to demonstrate ;-)
    clkper.nsec       = 1000000;
    clkper.fract      = 0;
    ClockPeriod ( CLOCK_REALTIME, &clkper, NULL, 0  );   // 1ms

    // --- keep track of time
    start = time(NULL);
    for( ;; )
    {
        // --- wait for pulse
        pid = MsgReceivePulse ( chid, &pulse, sizeof( pulse ), NULL );

        // --- should put pulse validation here ...
        current_cycles = ClockCycles();
```

```
        if ( last_cycles != -1 )    // --- don't print first iteration
        {
            // --- could get rid of timer by using more
            // --- clever timer setup
            float elapse = (current_cycles - last_cycles) / cpu_freq;

            // --- printf if request is 50us longer then requested...
            if ( elapse > .00105 )
            {
                printf("Elapse %f at %dn", elapse, time(NULL)-start );
            }
        }

        last_cycles = current_cycles;
    }
}
```

The program checks to see if the time between two timer events is greater then 1.05 milliseconds (ms). Most people expect that given QNX Neutrino's great real-time behavior, such a condition will never occur. But, surprise! It will; not because of an ill behaving kernel, rather because of the limitation in the PC hardware. It's impossible for the OS to generate a timer event at exactly 1.0 ms. It will be .99847 ms!!! This has unexpected side effects.

Where's the catch?

There is a 153 nanosecond (ns) discrepancy between the request and what the hardware can do. The kernel timer manager is invoked every .999847 ms. Every time a timer fires, the kernel checks to see if the timer is periodic and, if so, adds the number of nanoseconds to the expected timer expiring point, no matter what the current time is. This phenomenon is illustrated in the following diagram:



The first line illustrated the real time at which timer management occurs. The second line is the time at which the kernel expects the timer to be fired.

Note what happens at 6534! The next value appears to not have incremented by 1 ms, thus the event 6535 will NOT be fired!

For anyone that knows a bit about signal frequency, this phenomenon is called beat. When two signals of various frequencies are "added" a third frequency is generated. This is often seen if you use your camcorder to record a TV image. Because a TV is updated at 60 Hz and camcorders usually operate on different frequency, at playback, one can often see a white line that scrolls in the TV image. The speed of that line is related to the difference in frequency between the camcorder and the TV.

In this case we have two frequencies, one is 1000 Hz and the other is 1005.495 Hz. Thus, the beat frequency is 1.5 micro Hz, or one blip every 6535 milliseconds.

This behavior has the benefit of giving you the expected number of fired timers, on average. In the example above, after 1 minute, the program would have received 60000 fired timer event (1000 events /sec * 60 sec). If your design requires very precise timing, you have no other choice then to request a timer event of .999847 ms and not 1 ms. This can make the difference between a robot moving very smoothly or scratching your car.

Now, why don't you try to figure out what would happen if the tick size is set to 10 ms and a timer event is requested every 23 ms - and post your answer in the Neutrino Operating System forums!