Time

QNX SOFTWARE SYSTEMS

NOTES:

Introduction

You will learn how:

– QNX Neutrino handles time

– to read and update the system clock

– to use system timers and kernel timeouts

NOTES:

Time

Topics:

→ **Timing Architecture**

Getting and Setting the System Clock

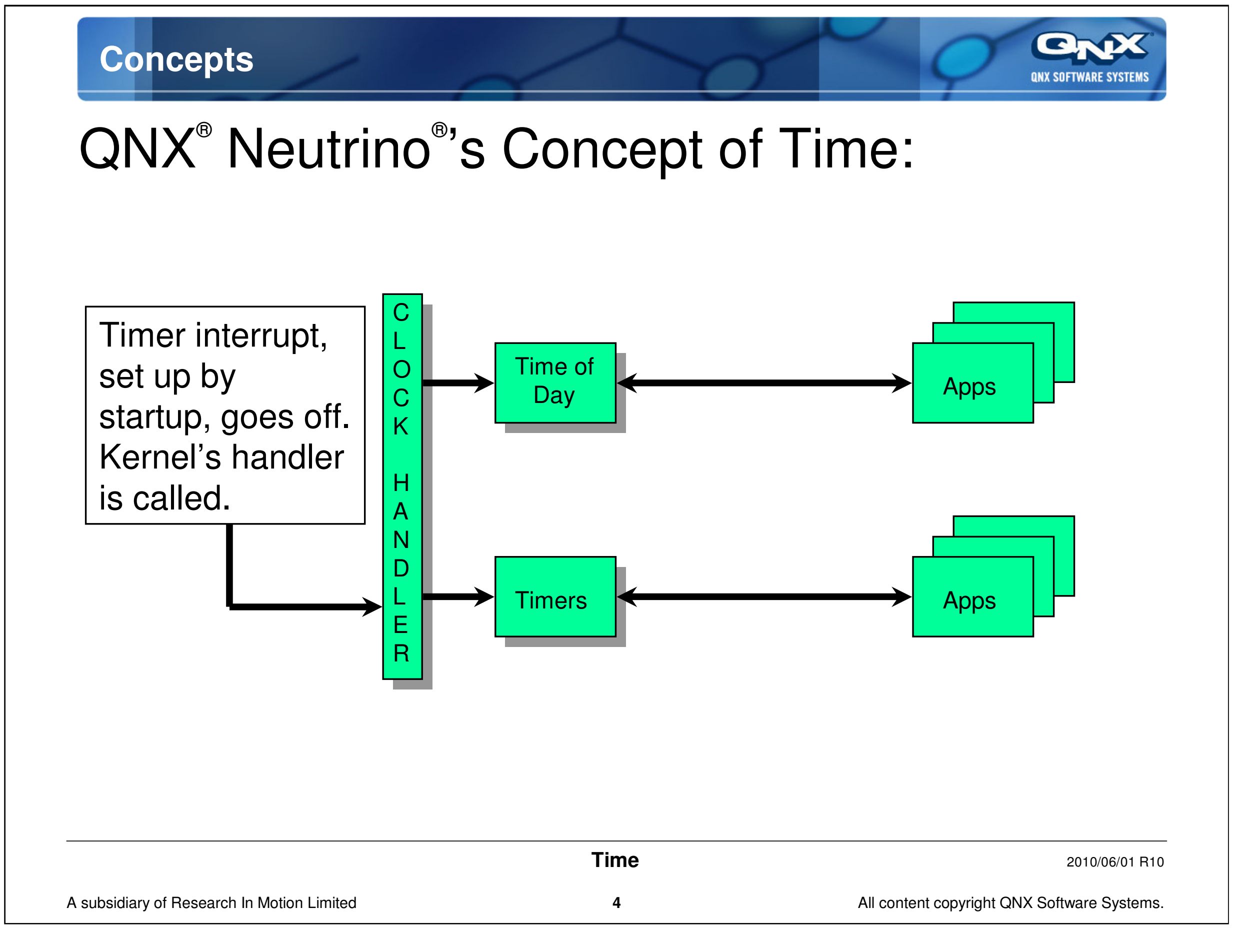
Timers

Design Considerations

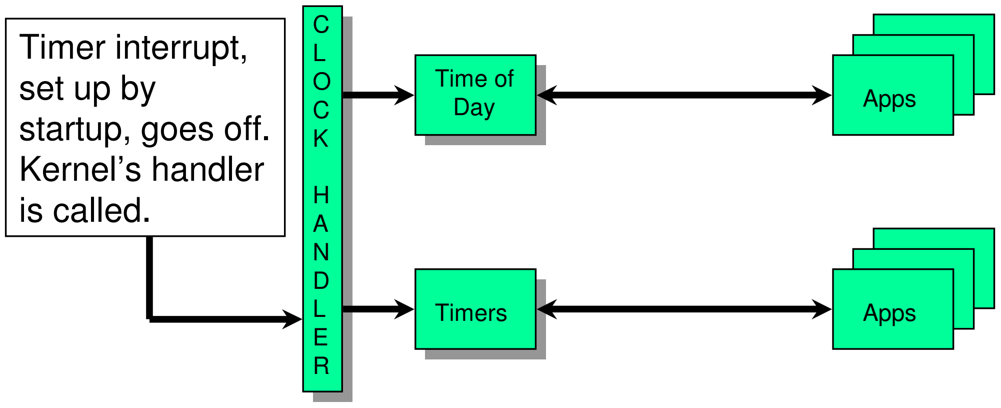
Kernel Timeouts

Conclusion

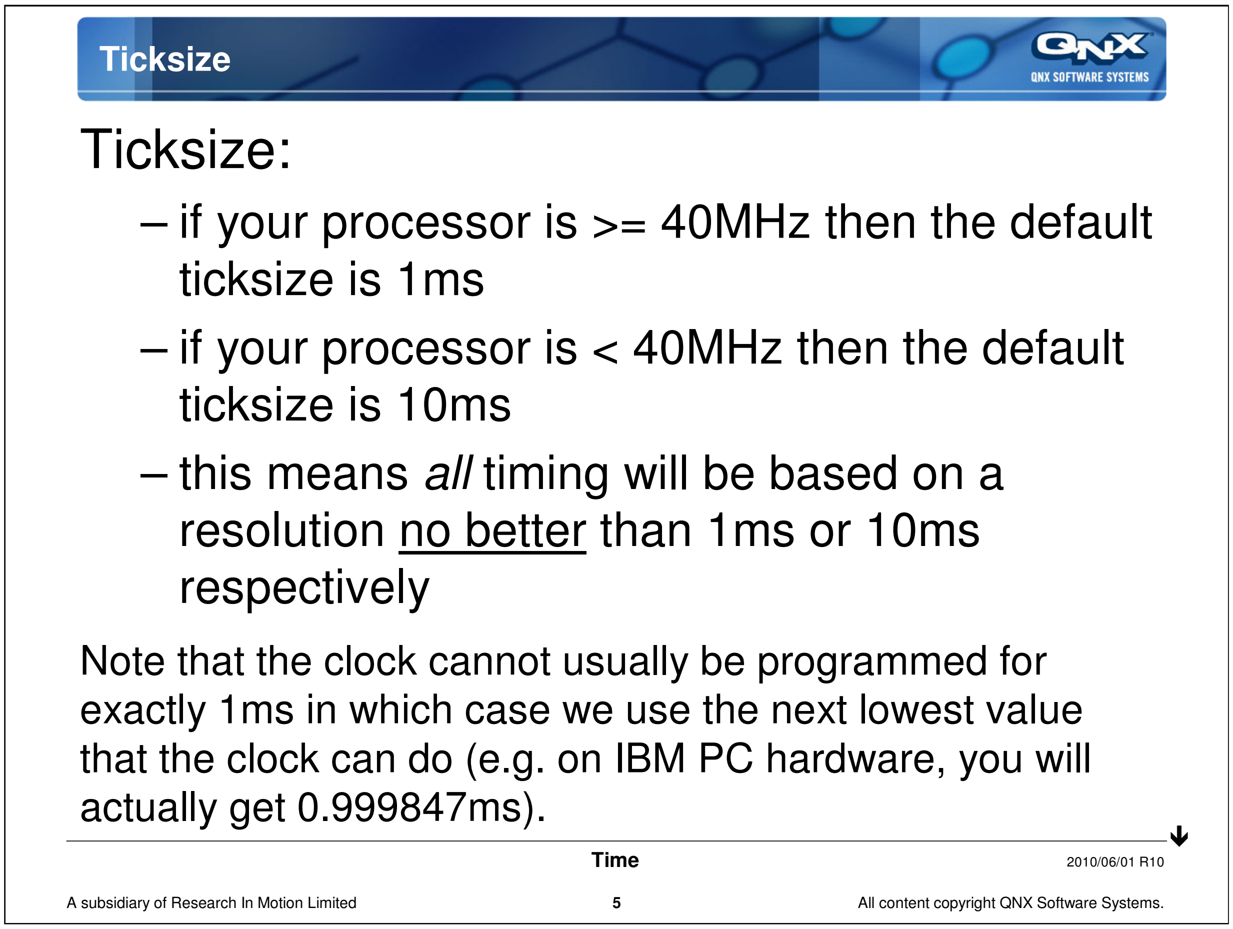
NOTES:



QNX® Neutrino®'s Concept of Time:


Timer interrupt, set up by startup, goes off. Kernel's handler is called.
CLOCK HANDLER
Time of Day
Timers
Apps
Apps

NOTES:



Ticksize

Ticksize:

- if your processor is >= 40MHz then the default ticksize is 1ms
- if your processor is < 40MHz then the default ticksize is 10ms
- this means *all* timing will be based on a resolution no better than 1ms or 10ms respectively

Note that the clock cannot usually be programmed for exactly 1ms in which case we use the next lowest value that the clock can do (e.g. on IBM PC hardware, you will actually get 0.999847ms).

NOTES:

Ticksize has a variety of names -- it can also be referred to as timebase, clockperiod, clock resolution and others.

The kernel will most likely not provide exactly the nominal value because the rate is derived from the crystal oscillator, and is divided by an INTEGER divisor.

Time Slice

The time slice:

- is 4 times the ticksize so it defaults to either 4ms or 40ms
- the multiplier, 4, cannot be changed. If you change the ticksize then the time slice will also change

NOTES:

The timeslice affects two things, round robin scheduling and SMP global rescheduling. The size of the timeslice is usually not an issue, as threads don't usually run for their entire time slice before blocking.

Timing Examples

Let's examine how timing works by looking at some examples:

– we'll have two threads, t1 and t2, both READY at priority 10, both doing round-robin
– t2 will go to sleep for 10.5ms (a relative time)
– when t2 goes to sleep, the kernel figures out when to wake it up using this formula:

```
wake_up_time = now + requested_time + 1 tick
```

(where **`now`** is really the time as of the last tick)

NOTES:

Why does the kernel add 1 tick extra? Because according to POSIX you cannot sleep for less time than you ask for. The kernel has no idea what the time is in between ticks. In the formula above, **`now`** is actually the time as of the last tick. If we didn't add an extra tick then you could end up sleeping for less time than you asked for.

Timing Example 1 - Not so good

Keep in mind the following:

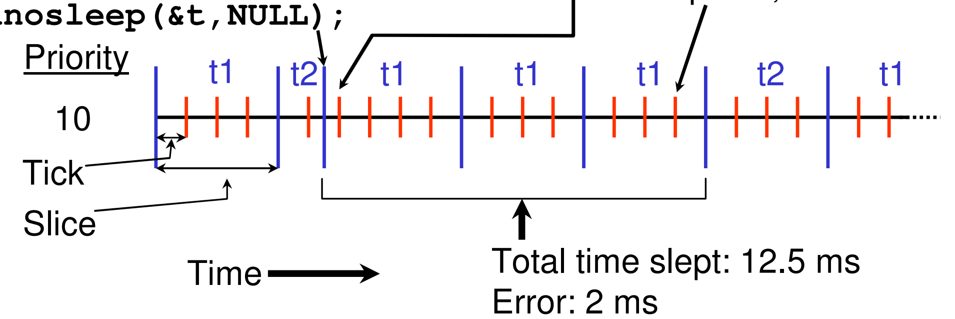
1. At every tick the kernel checks for expired timers.
2. If a thread's timer has expired then the kernel puts that thread at the end of the READY queue for that thread's priority.

For the expiry time, the kernel adds 10.5 ms to whatever time it will be here

t2's timer has expired. t2 is put at the end of priority 10's READY queue, BUT t1 still runs

t2 does:

```
t.tv_nsec = 10500000;
nanosleep(&t,NULL);
```



Total time slept: 12.5 ms
Error: 2 ms
BUT, error can be as bad as 5 ms!

NOTES:

The *nanosleep()* function puts a thread to sleep for some number of milliseconds:

```
    nanosleep( struct timespec *requestedtime,
               struct timespec *timeremaining);
```

Notice that in the example above, we asked to sleep for 10,500,000 nsecs which is 10.5 msecs.

Note that the above is an idealized diagram. On an IBM PC, a 1ms tick would really be 0.999847ms.

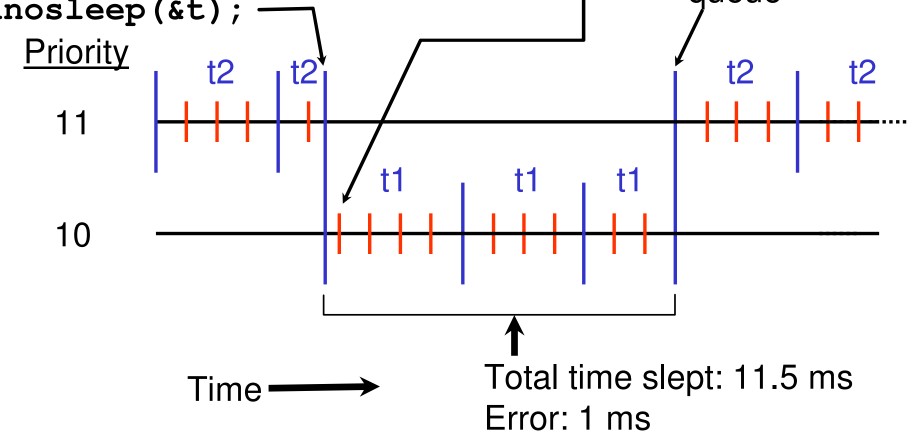
If a timer expires and makes a thread ready and a time slice is expiring, the thread is made ready by the timer first, then the time slice expiry is applied.

Timing Example 2 - Better

For the expiry time, the kernel adds 10.5 ms to whatever time it will be here

t2's timer has expired. t2 is put at the end of priority 11's READY queue

t2 does:
t.tv_sec = 0; t.tv_nsec = 10500000;
`nanosleep(&t);`

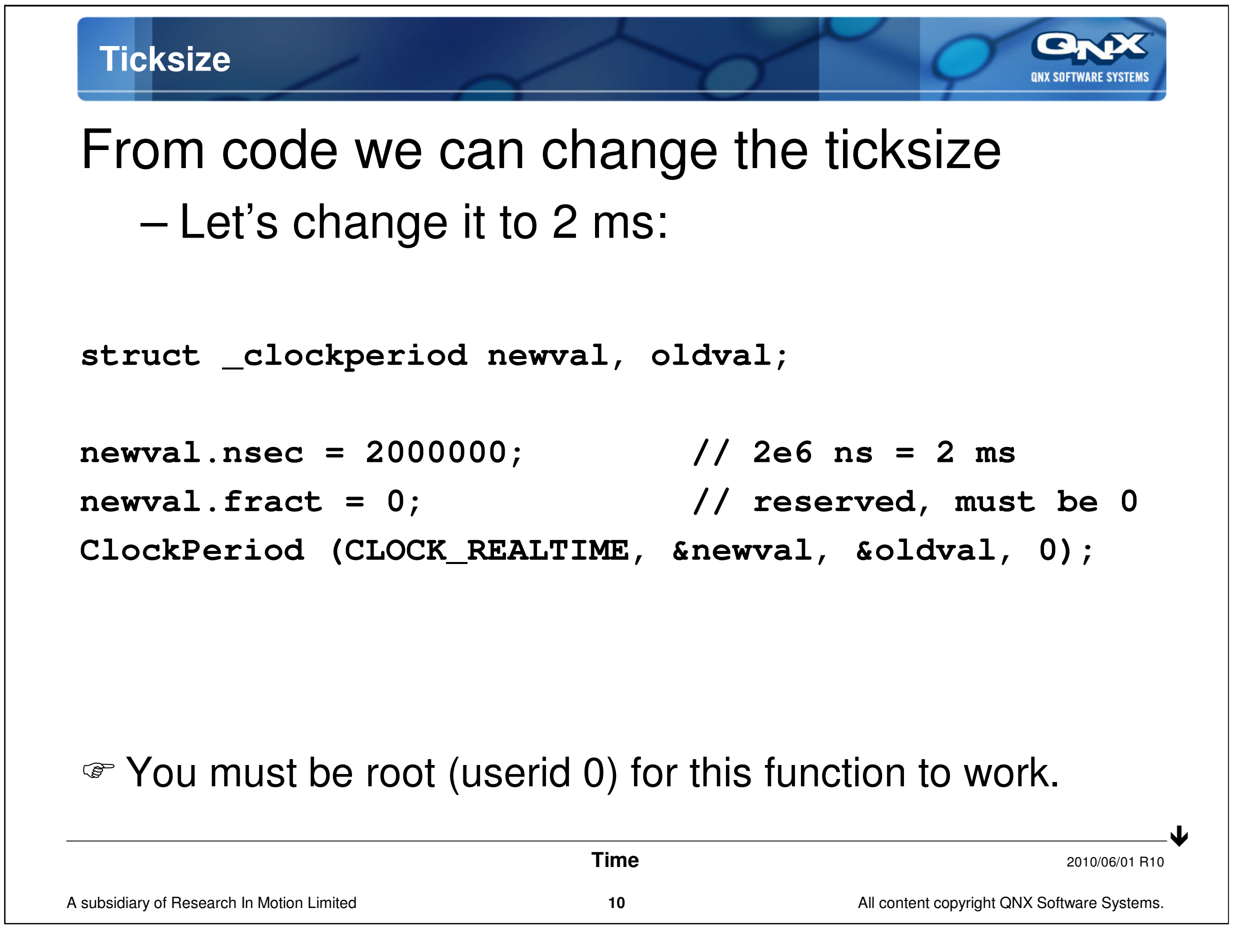


Total time slept: 11.5 ms
Error: 1 ms
Error can be as bad as 2 ms

Don't forget. A higher priority thread can preempt all of this and interrupt handlers can preempt even those.

NOTES:

To improve the timing further, you could make the ticksize smaller. Keep in mind though that the tick really represents the kernel's interrupt handler for the timer interrupt. By decreasing the ticksize you are increasing the frequency of this interrupt and therefore increase interrupt and scheduling latencies and system overhead.



Ticksize

From code we can change the ticksize

– Let’s change it to 2 ms:

```
struct _clockperiod newval, oldval;

newval.nsec = 2000000;          // 2e6 ns = 2 ms
newval.fract = 0;               // reserved, must be 0
ClockPeriod (CLOCK_REALTIME, &newval, &oldval, 0);
```

☞ You must be root (userid 0) for this function to work.

NOTES:

The kernel will most likely not provide exactly 2 ms timing; it will round to the nearest smaller available clock period. To find out what actual ticksize you got, call **`ClockPeriod( CLOCK_REALTIME, NULL, &curval, 0);`** after making your change.

Even on slow machines where we default to 10 ms, a value of 1 ms is always allowed.

The ticksize is a system wide configuration. You would normally set it once as part of your system initialization to whatever you have chosen.

Topics:

Timing Architecture

→ **Getting and Setting the System Clock**

Timers

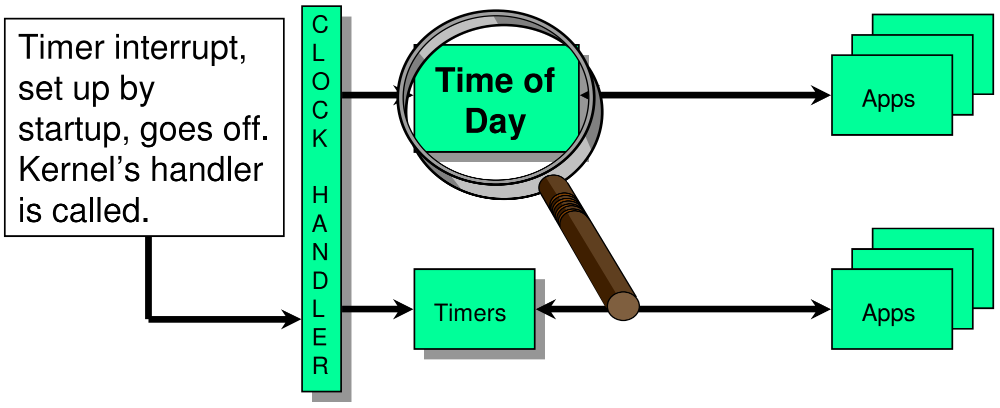
Design Considerations

Kernel Timeouts

Conclusion

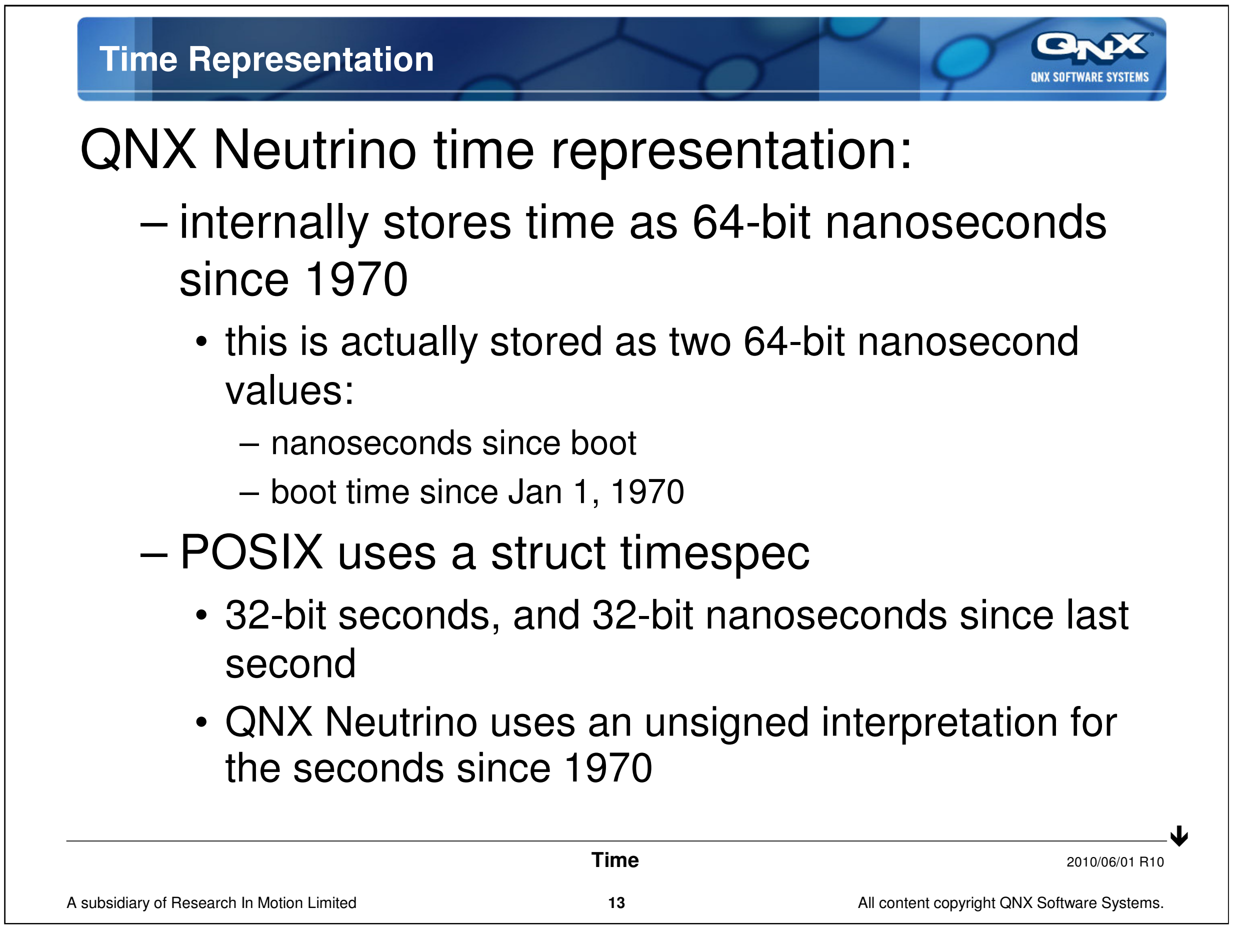
NOTES:

Let’s look at the time of day functions:


Timer interrupt, set up by startup, goes off. Kernel’s handler is called.
CLOCK HANDLER
Time of Day
Timers
Apps
Apps

NOTES:

The system clock is often referred to as the "time of day" clock.



Time Representation

QNX Neutrino time representation:

- internally stores time as 64-bit nanoseconds since 1970
  - this is actually stored as two 64-bit nanosecond values:
    - nanoseconds since boot
    - boot time since Jan 1, 1970
- POSIX uses a struct timespec
  - 32-bit seconds, and 32-bit nanoseconds since last second
  - QNX Neutrino uses an unsigned interpretation for the seconds since 1970

NOTES:

When do these expire? (That is, when do the values wrap to 0/negative).

POSIX 32-bit seconds, if interpreted as a signed value (historically common case), wraps to negative in 2038.

POSIX 32-bit unsigned seconds wraps in 2106.

QNX unsigned 64-bit nanosecond value is valid past 2500.

Keeping Track of Time

At bootup time:

- the kernel is given the current date and time from somewhere (battery backed up clock as on a PC, GPS, NTP, some atomic clock, ...)
- from then on, every tick, the kernel adds the ticksize to the current time
    - e.g. for a 1ms tick, every 1ms the kernel adds 1ms to the current time

NOTES:

Actually, the kernel adds the real (nanosecond) value of the ticksize on each tick, not the "rounded" 1ms value that might have been requested.

NTP stands for Network Time Protocol and is a way of getting time over a network (often from an atomic clock).

Read and/or Set the System Clock:

```
struct timespec tval;

clock_gettime( CLOCK_REALTIME, &tval );
tval.tv_sec += (60*60)*24L; /* add one day */
tval.tv_nsec = 0;
clock_settime( CLOCK_REALTIME, &tval );
```

NOTES:

We can adjust the time:

– Bring it forward by one second:

```
struct _clockadjust new, old;

new.tick_nsec_inc = 10000; // 1e4 ns == 10 µs
new.tick_count = 100000;   // 100k * 10µs = 1s
ClockAdjust (CLOCK_REALTIME, &new, &old);
```

– Bring it backward by one second:

```
new.tick_nsec_inc = -10000;// 1e4 ns == 10 µs
new.tick_count = 100000;    // 100k * 10µs = 1s
ClockAdjust (CLOCK_REALTIME, &new, &old);
```

In both above examples the total adjustment will usually take 100 seconds (100k ticks (for a 1 msec tick size) = 100 seconds.)

NOTES:

Rule of thumb:

Do not adjust the clock by more than 1% of the tick size, otherwise distortion from the “real” world would be too great. Definitely don’t adjust more than the ticksize, especially in the negative direction as this would result in time going backwards.

QNX provides a free running counter:

```
uint64_t count;
count = ClockCycles ();
```

– It returns increments of the **`cycles_per_sec`** available from the system page (see below).
– On a system with supporting hardware we use it
  - e.g. **`rdtsc`** op code on Pentium class machines or the PPC decrementor register.
  - these often increment at processor clock speed, e.g on a 100 MHz processor it returns increments of 10 ns (2 ns on 500 MHz, etc.)
– On a processor that does not have a free running counter, we fake it.

To find out how many cycles per second this clock is running at:

```
#include <sys/syspage.h>
uint64_t  cycles_per_sec;
cycles_per_sec = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
```

NOTES:

While a 64-bit value incremented at 2 GHz would take approximately 272 years to wrap, on other processors there may only be a 32-bit counter available, in which case wrapping is a possibility. This does need to be taken into account.

Accessing the system page is discouraged except in cases where the information cannot be found anywhere else.

Exercise:

- in your **`time`** project
- look at **`calctime1.c`**, and run it
  - what do the first set of delta values represent?
  - what would you do to figure out how long each iteration of the second **`for`** loop takes in microseconds?
- modify **`calctime2.c`** to adjust the ticksize to be 2 milliseconds

NOTES:

Time

Topics:

Timing Architecture

Getting and Setting the System Clock

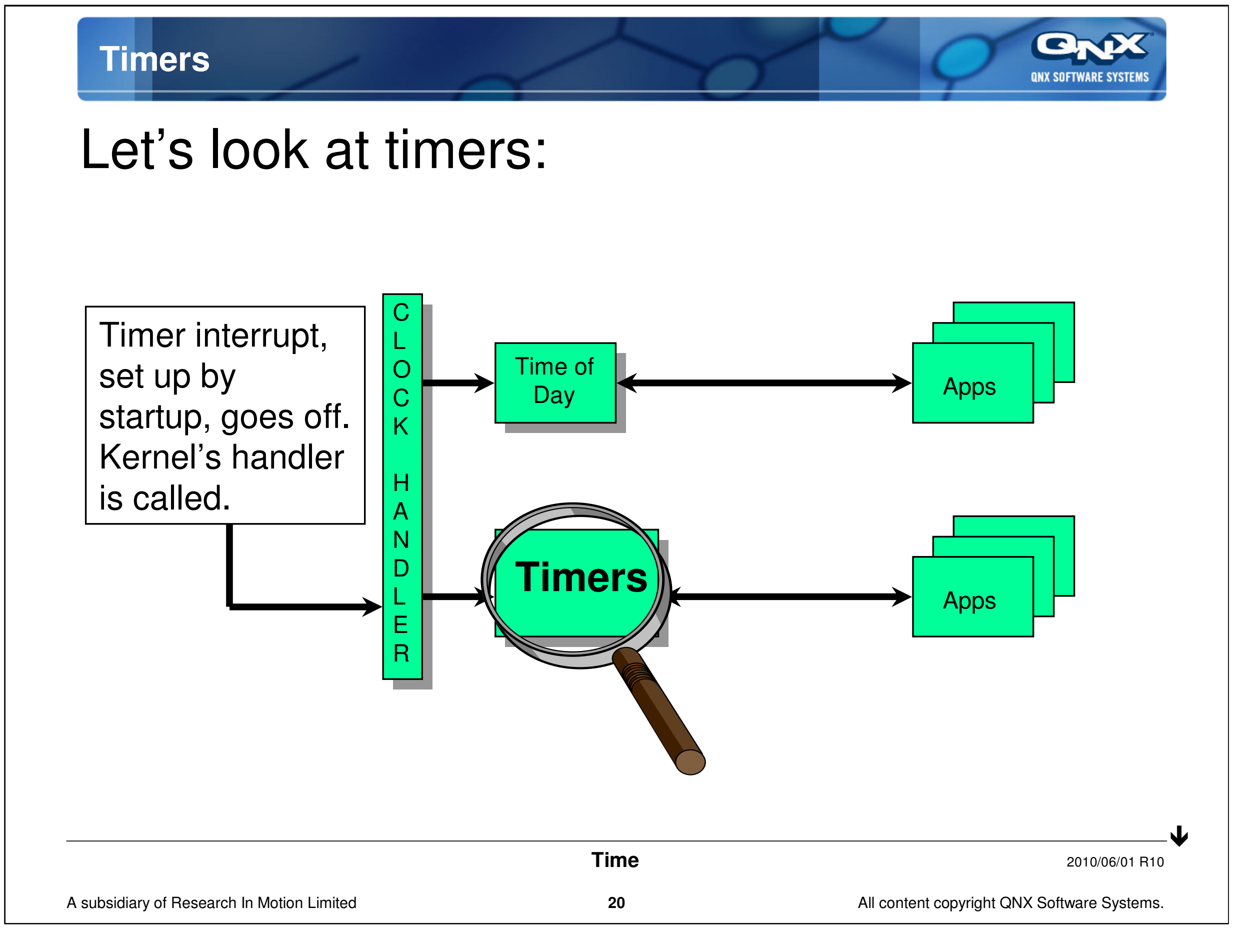
→ Timers

Design Considerations

Kernel Timeouts

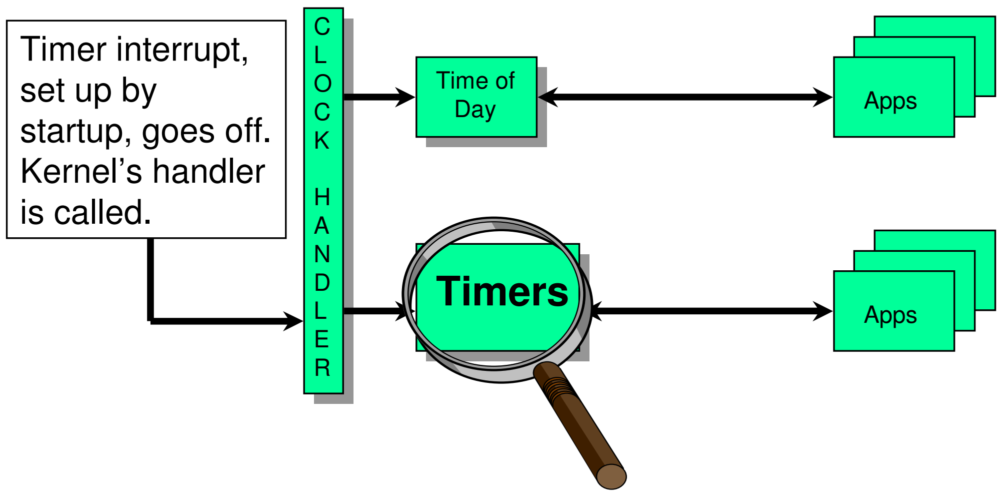
Conclusion

NOTES:



Timers

Let's look at timers:


Timer interrupt, set up by startup, goes off. Kernel's handler is called.
CLOCK HANDLER
Time of Day
Apps
Timers
Apps

Time

2010/06/01 R10

NOTES:

The kernel keeps the list of timers sorted. The timer to expire soonest is at the head of the list. If that timer has not expired then none of the other timers have. So if you have many timers, unless they are all expiring during the same tick, you will not increase the length of time spent by the kernel in its timer interrupt handler.

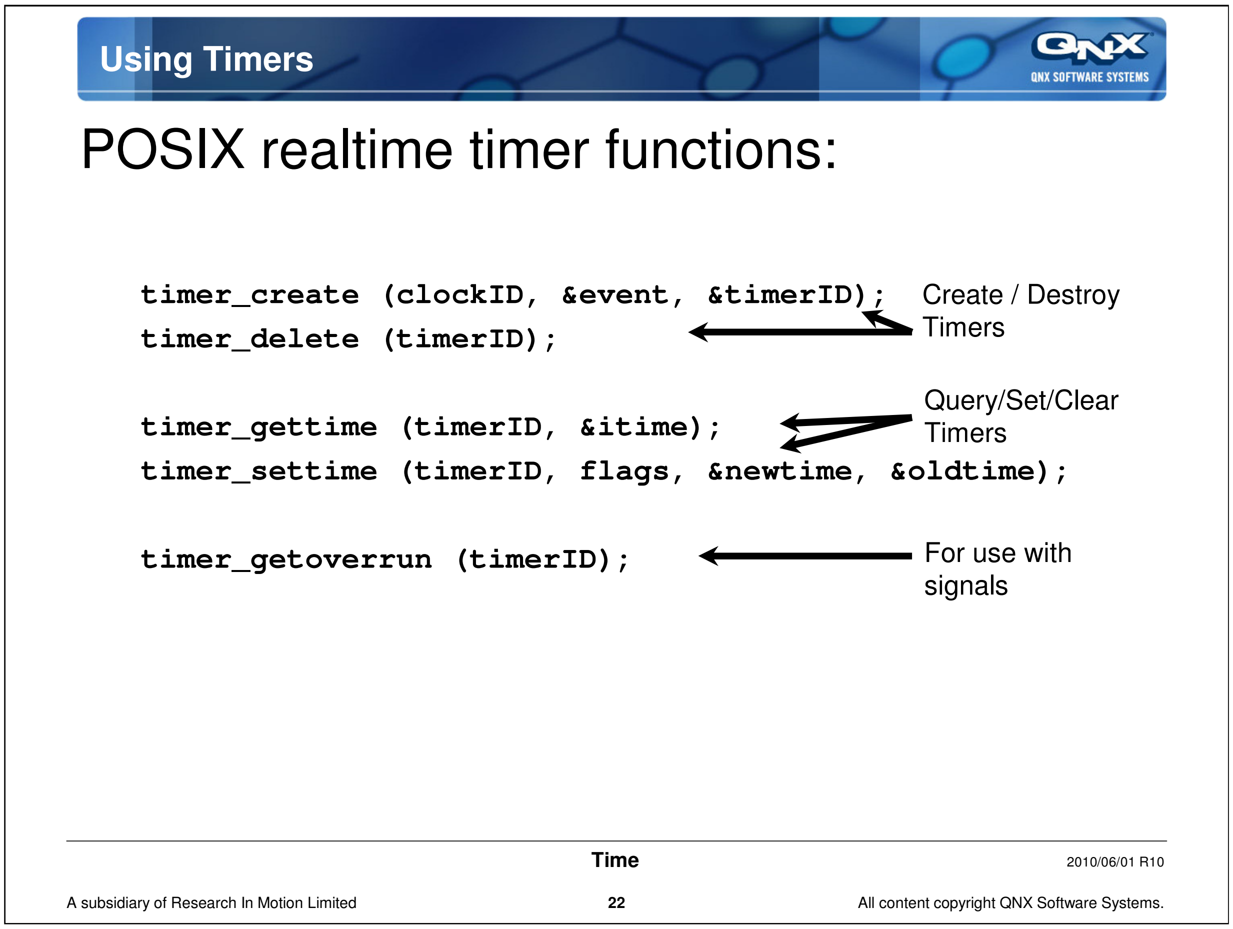
Setting a Timer

To set a timer, the process chooses:

- what kind of timer
  - periodic
  - one shot
- timer anchor
  - absolute
  - relative
- event to deliver upon trigger
  - fill in an EVENT structure

☞ If preparing to use power management, make sure you stop your periodic timers when you don't need them.

NOTES:



Using Timers

QNX SOFTWARE SYSTEMS

POSIX realtime timer functions:

```
timer_create (clockID, &event, &timerID);
timer_delete (timerID);
```

Create / Destroy Timers

```
timer_gettime (timerID, &itime);
timer_settime (timerID, flags, &newtime, &oldtime);
```

Query/Set/Clear Timers

```
timer_getoverrun (timerID);
```

For use with signals

Time

2010/06/01 R10

A subsidiary of Research In Motion Limited

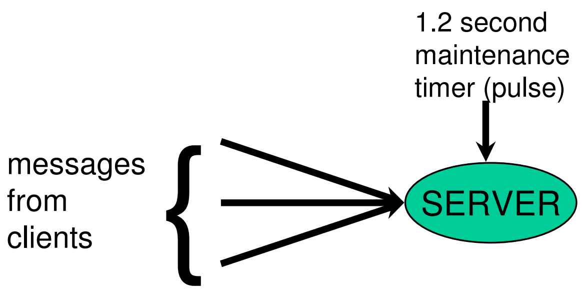
22

NOTES:

Setting a Timer and Receiving Timer Pulses

QNX SOFTWARE SYSTEMS

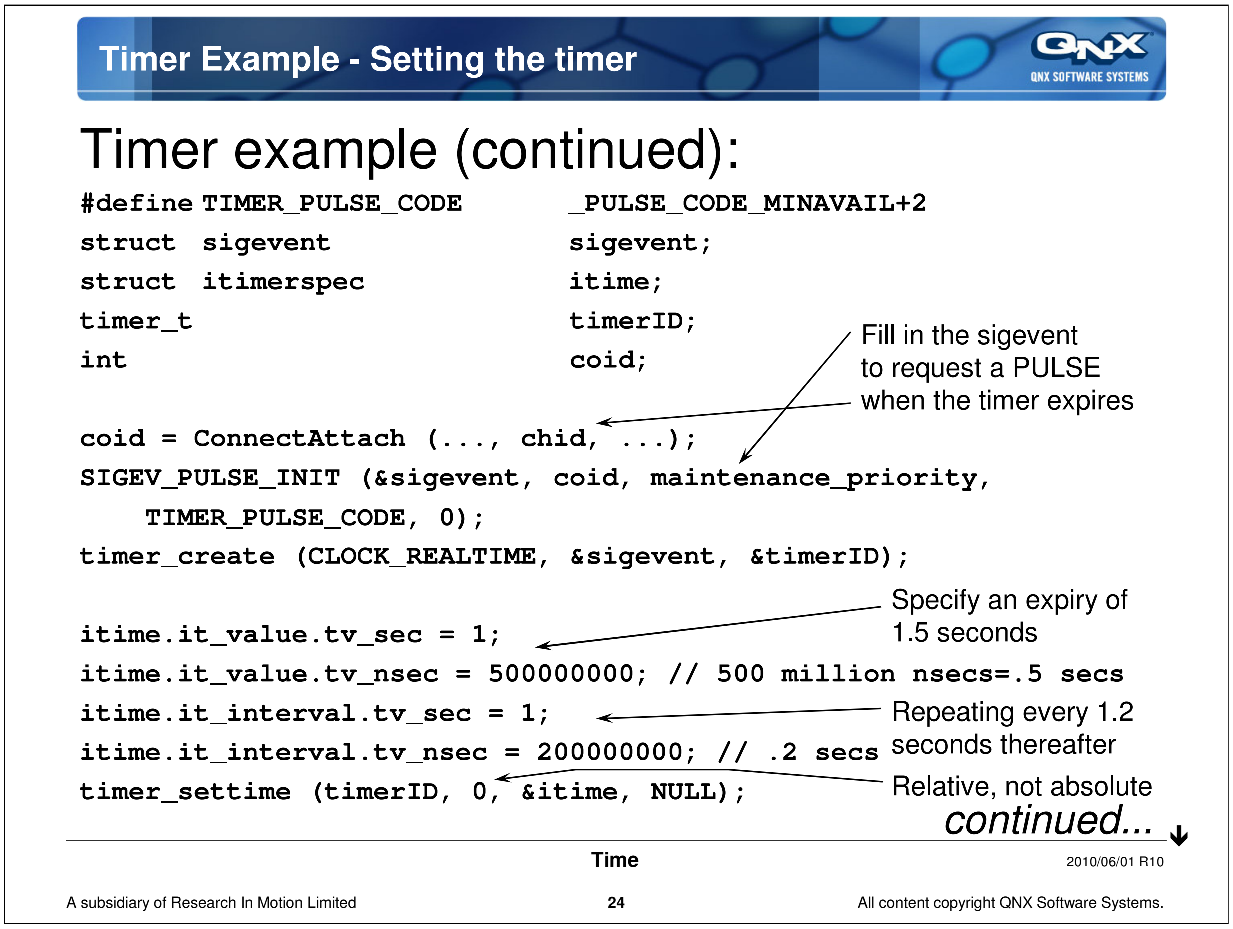
Timer example:



We want to have the server receive maintenance timer messages every 1.2 seconds, so that it can go out and perform housekeeping duties / integrity checks, etcetera.

continued...

NOTES:



Timer Example - Setting the timer

Timer example (continued):

```
#define TIMER_PULSE_CODE        _PULSE_CODE_MINAVAIL+2
struct  sigevent                sigevent;
struct  itimerspec              itime;
timer_t                         timerID;
int                             coid;

coid = ConnectAttach (..., chid, ...);
SIGEV_PULSE_INIT (&sigevent, coid, maintenance_priority,
    TIMER_PULSE_CODE, 0);
timer_create (CLOCK_REALTIME, &sigevent, &timerID);

itime.it_value.tv_sec = 1;
itime.it_value.tv_nsec = 500000000; // 500 million nsecs=.5 secs
itime.it_interval.tv_sec = 1;
itime.it_interval.tv_nsec = 200000000; // .2 secs
timer_settime (timerID, 0, &itime, NULL);
```

Fill in the sigevent to request a PULSE when the timer expires

Specify an expiry of 1.5 seconds

Repeating every 1.2 seconds thereafter

Relative, not absolute

continued...

NOTES:

For an absolute time put **`TIMER_ABSTIME`** in the second parameter to *timer_settime()*. **`itime.it_value.tv_sec`** would then be the time at which the timer should go off (e.g. 5:00pm Friday.) The time would be given as the number of seconds since Jan 1 1970 00:00 GMT (see *mktime()* for one way of getting this.) **`itime.it_interval`** can still contain a repetition interval.

Timer Example - Receiving the timer pulses

Timer example (continued):

```
typedef union {
   struct _pulse   pulse;
   // other message types you will receive
} myMessage_t;

myMessage_t   msg;
    ... // the setup code from the previous page goes here
    while (1) {
      rcvid = MsgReceive (chid, &msg, sizeof(msg), NULL);
      if (rcvid == 0) {
        // it's a pulse, check what type...
        switch (msg.pulse.code) {
          case  _TIMER_PULSE_CODE:
                periodic_maintenance();
                break;
        }
      }
    }
```

NOTES:

Query Functions

How much time is left before expiry?

```
struct itimerspec  timeLeft;

timer_gettime (timerID, &timeLeft);
```

Returns:

`timeLeft.it_value.tv_sec` `timeLeft.it_value.tv_nsec`	Time left before expiry, or zero if timer is disarmed
`timeLeft.it_interval.tv_sec` `timeLeft.it_interval.tv_nsec`	Timer reload value, zero if timer is a one-shot, nonzero if timer is a repetitive timer.

NOTES:

Cancelling a Timer

Often you'll want to cancel a timer without destroying it (i.e. without removing it from the timer list)

- useful if you will frequently be canceling it then restarting it
- to cancel a timer:

```
struct itimerspec itime;
itime.it_value.tv_sec = 0;
itime.it_value.tv_nsec = 0;
timer_settime (timeID, 0, &itime, NULL);
```

- to restart it simply fill in the timing and call *timer_settime()* again

NOTES:

Exercise:

- In you **`time`** project is a file called **`reptimer.c`**
- When finished, it will wake up 5 seconds from the time it runs and then every 1500 milliseconds after that.
  - It will wake up by receiving a pulse.
- All of the code is in *main()* for setting up the pulse event structure and for receiving the pulse.
- However, the code for creating the timer and starting it ticking is missing. Add it. If you search for the word "class" you will see a good place to put your code.
- To test it do:

  ```
  reptimer
  ```

  You should see a message displayed to the screen 5 seconds from the time you ran it, and then every 1500 milliseconds (1.5 seconds) after.

NOTES:

QNX SOFTWARE SYSTEMS

Topics:

Timing Architecture

Getting and Setting the System Clock

Timers

$\longrightarrow$ Design Considerations

Kernel Timeouts

Conclusion

NOTES:

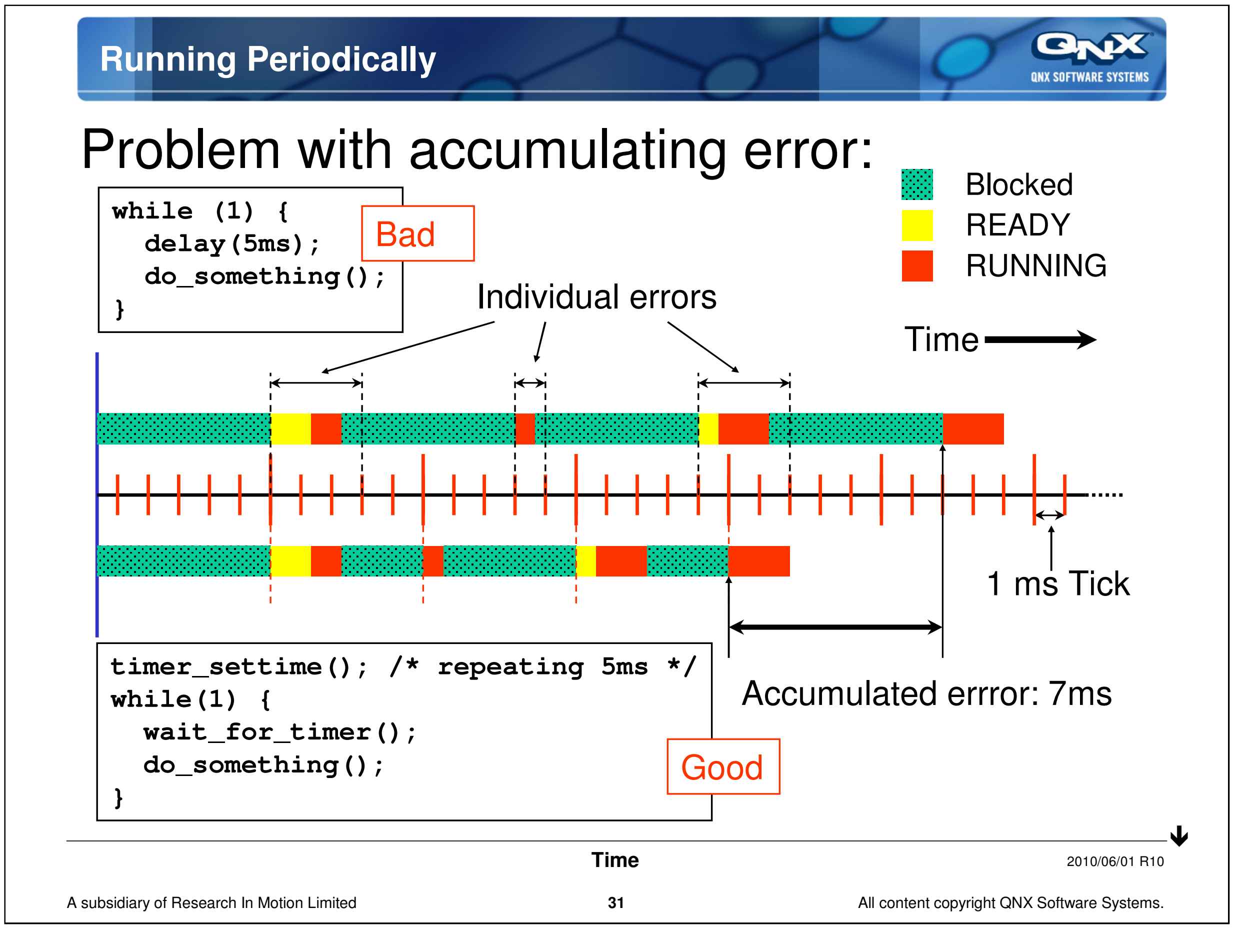
Design Considerations

QNX SOFTWARE SYSTEMS

Two timing design issues:

- how to run periodically without accumulating error
- timer frequency issues which can make timer expiry erratic

NOTES:



Running Periodically

Problem with accumulating error:

```
while (1) {
  delay(5ms);
  do_something();
}
```

Bad

🟩 Blocked
🟨 READY
🟥 RUNNING

```
timer_settime(); /* repeating 5ms */
while(1) {
  wait_for_timer();
  do_something();
}
```

Good

Time 2010/06/01 R10

A subsidiary of Research In Motion Limited 31 All content copyright QNX Software Systems.

NOTES:

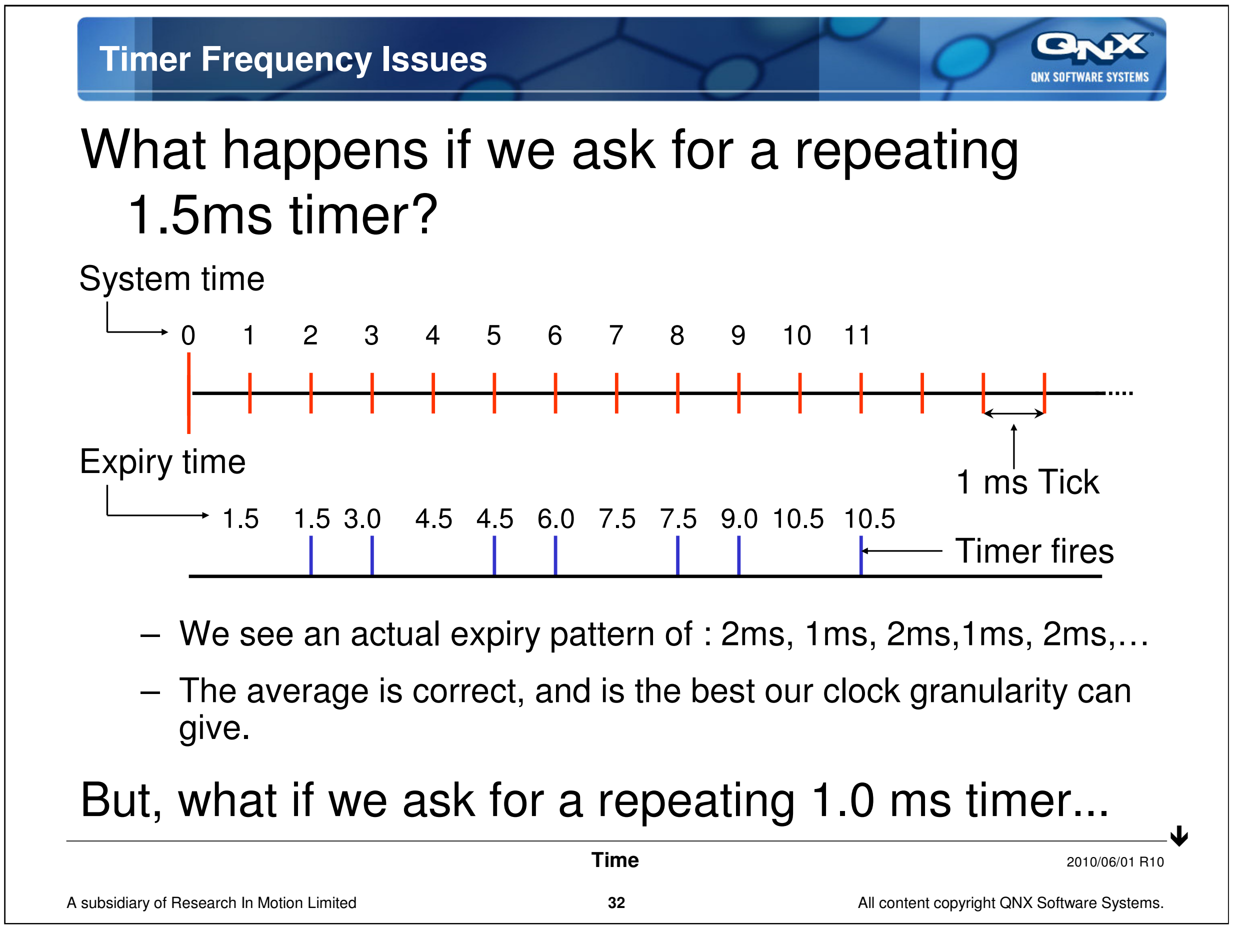
For both of these approaches we have the same initial error for the first 5 ms delay, and the same errors due to being made READY by our timer expiring and some other thread at our priority or a higher priority preventing us from running.

In the *delay()* case, though, we start a new 5 ms interval after we complete our work, so we get cumulative error from:

- any time READY but not RUNNING
- time spent doing the work
- timer initiatilizaton error do to ticksize round up

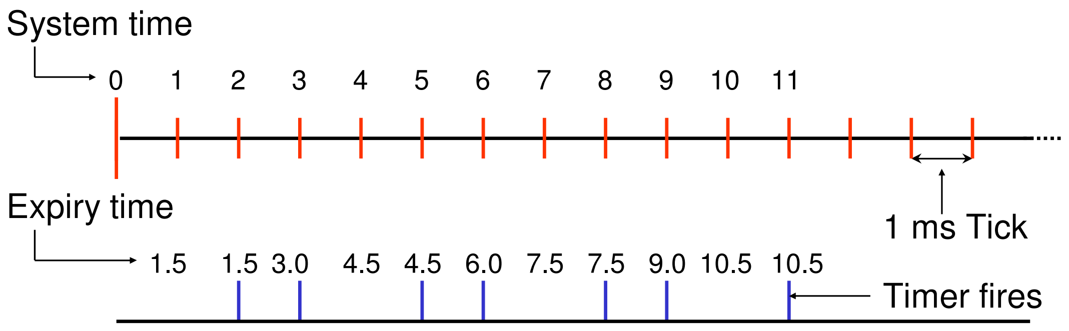
While in the repeating timer case, we have the same startup error, and the same errors from not getting scheduled immediately, the 5ms intervals are always measured from the startup of the timer, so we don't accumulate any error.

In this example, which is reasonably typical, when using the *delay()* loop, we've accumulated 7ms of error over 4 iterations.



Timer Frequency Issues

What happens if we ask for a repeating 1.5ms timer?



- We see an actual expiry pattern of : 2ms, 1ms, 2ms,1ms, 2ms,…
- The average is correct, and is the best our clock granularity can give.

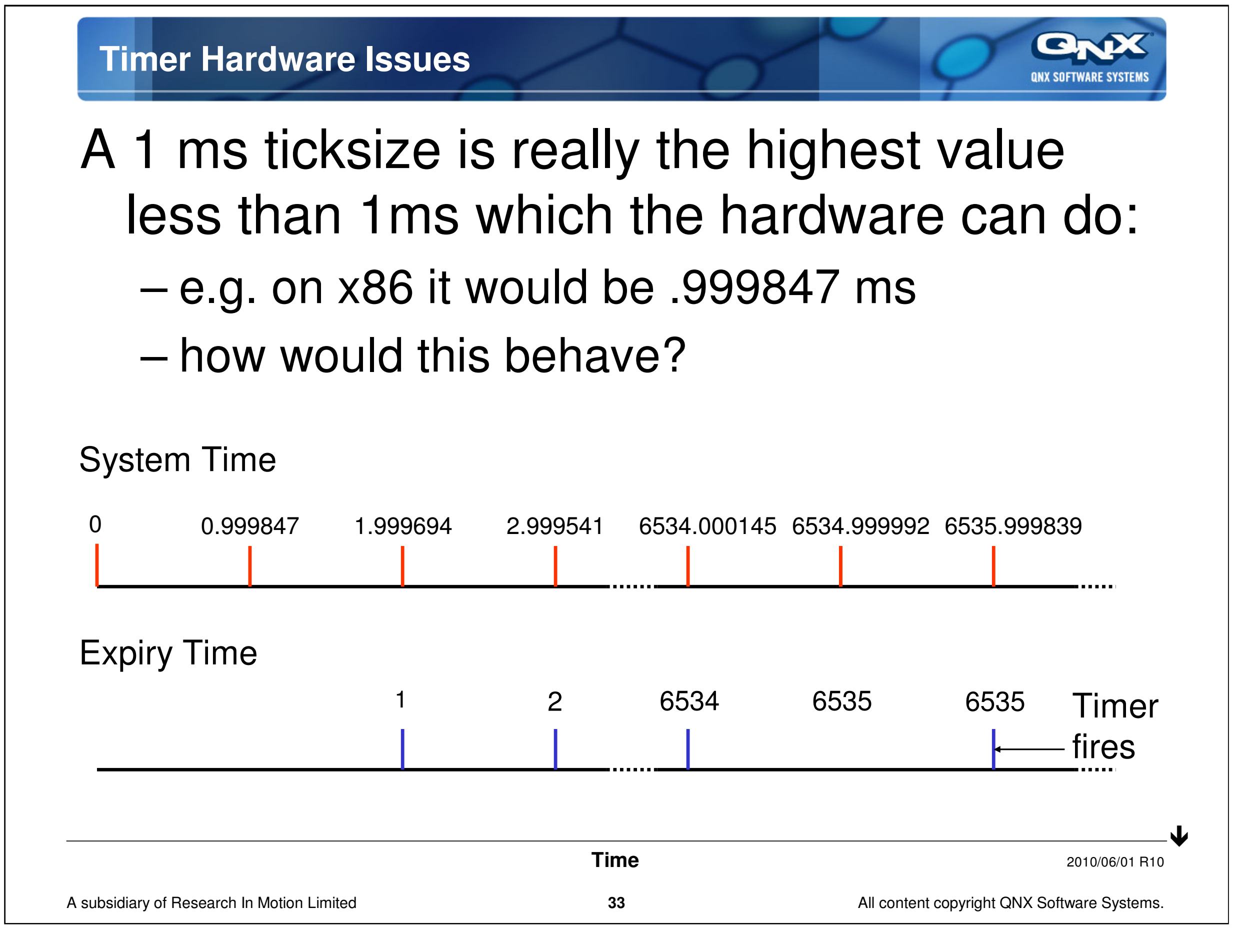
But, what if we ask for a repeating 1.0 ms timer...

Time 2010/06/01 R10

NOTES:

On each tick, QNX checks to see if the system time has exceeded the expiry time for the timer, and once it has, the timer will fire.

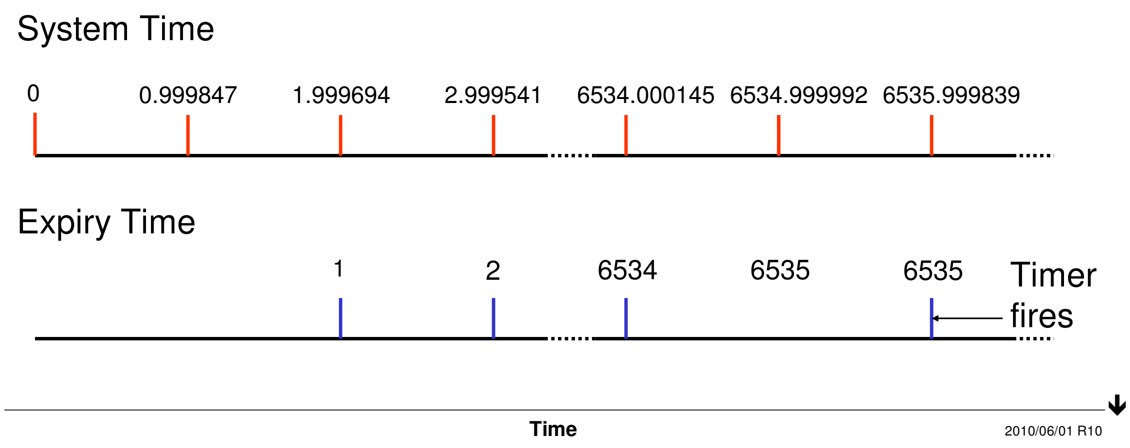
In the real world, the time values would not be counting from 0, but would be counting actual clock time.



Timer Hardware Issues

A 1 ms ticksize is really the highest value less than 1ms which the hardware can do:

- e.g. on x86 it would be .999847 ms
- how would this behave?



NOTES:

Because of the initialization error, we do miss the first tick -- this is expected. But, we will also, very occasionally, have a (almost) 2ms interval instead of our desired 1ms interval. This can often cause a problem.

Other hardware will have different values for the actual ticksize, but this same problem will occur, just at different intervals.

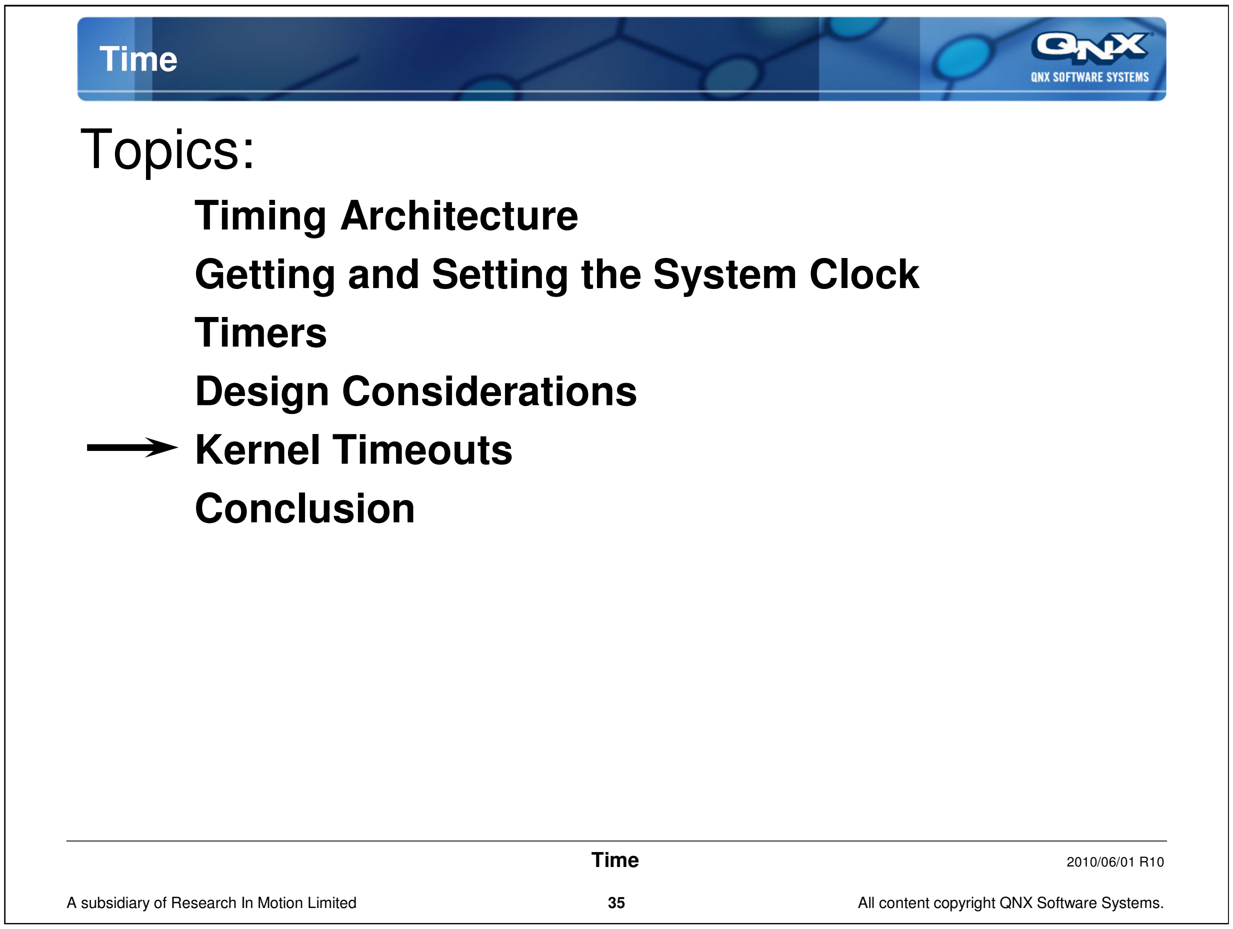
This issue will also add an additional error of almost 1ms to every iteration of the *delay()* based method of repeatedly running.

Timer Frequency Issues - What to do

So what do you do? There are some choices:

- make sure your tick size is quite a bit smaller than the smallest timer period you need
  - smaller ticksizes will impose more system overhead from the kernel handling the timer interrupt more frequently
- make your timer expiry an exact multiple of the actual tick size
  - use *ClockPeriod*() to get this value
  - use **`CLOCK_MONOTONIC`** so that time changes and *ClockAdjust()* don't affect your timing

NOTES:



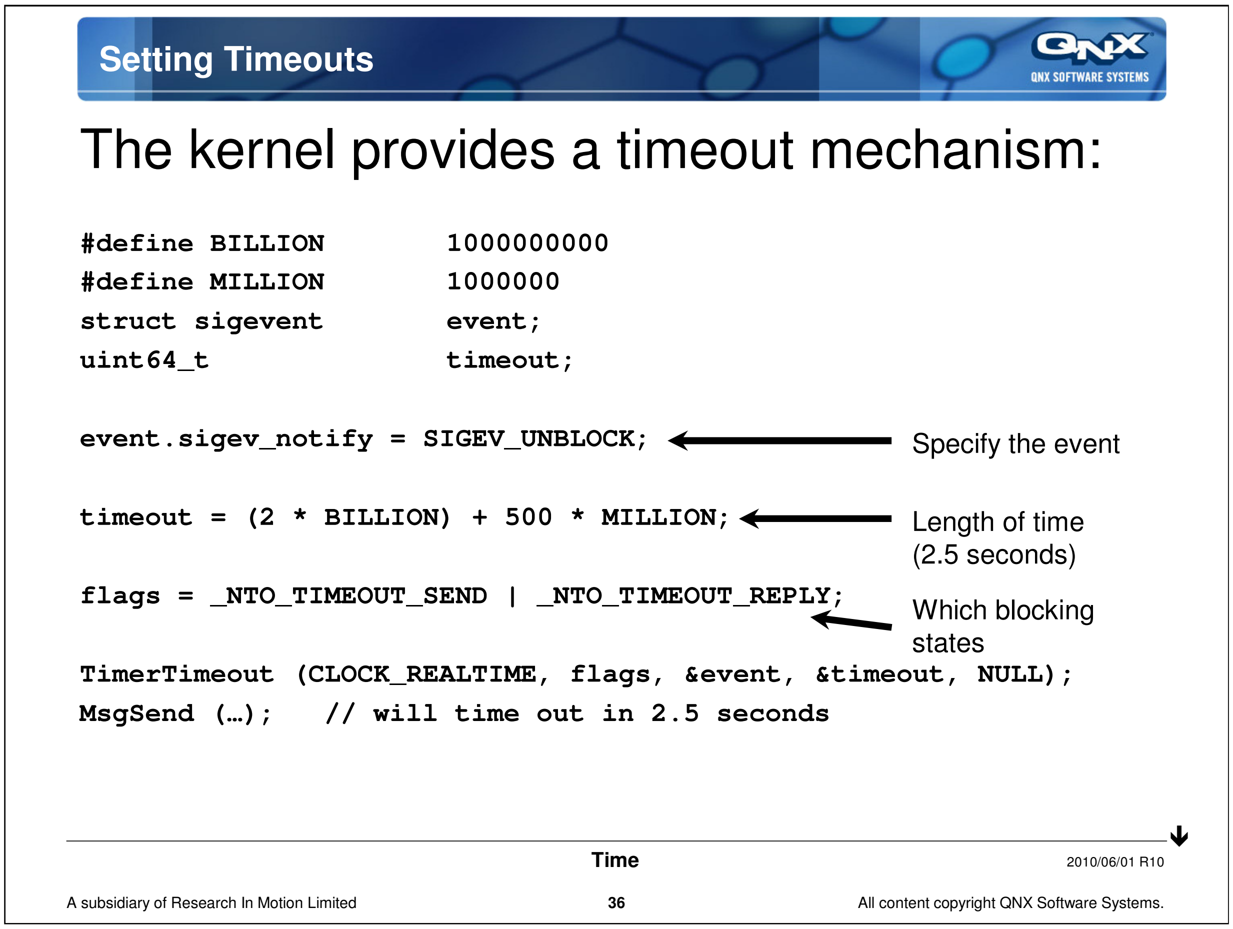
Time

Topics:

- **Timing Architecture**
- **Getting and Setting the System Clock**
- **Timers**
- **Design Considerations**
- → **Kernel Timeouts**
- **Conclusion**

Time 2010/06/01 R10

NOTES:



Setting Timeouts

QNX SOFTWARE SYSTEMS

The kernel provides a timeout mechanism:

```
#define BILLION         1000000000
#define MILLION         1000000
struct sigevent         event;
uint64_t                timeout;

event.sigev_notify = SIGEV_UNBLOCK;

timeout = (2 * BILLION) + 500 * MILLION;

flags = _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY;

TimerTimeout (CLOCK_REALTIME, flags, &event, &timeout, NULL);
MsgSend (…);   // will time out in 2.5 seconds
```

Specify the event

Length of time (2.5 seconds)

Which blocking states

Time — 2010/06/01 R10

NOTES:

Note that this timeout is done on a per-thread basis.

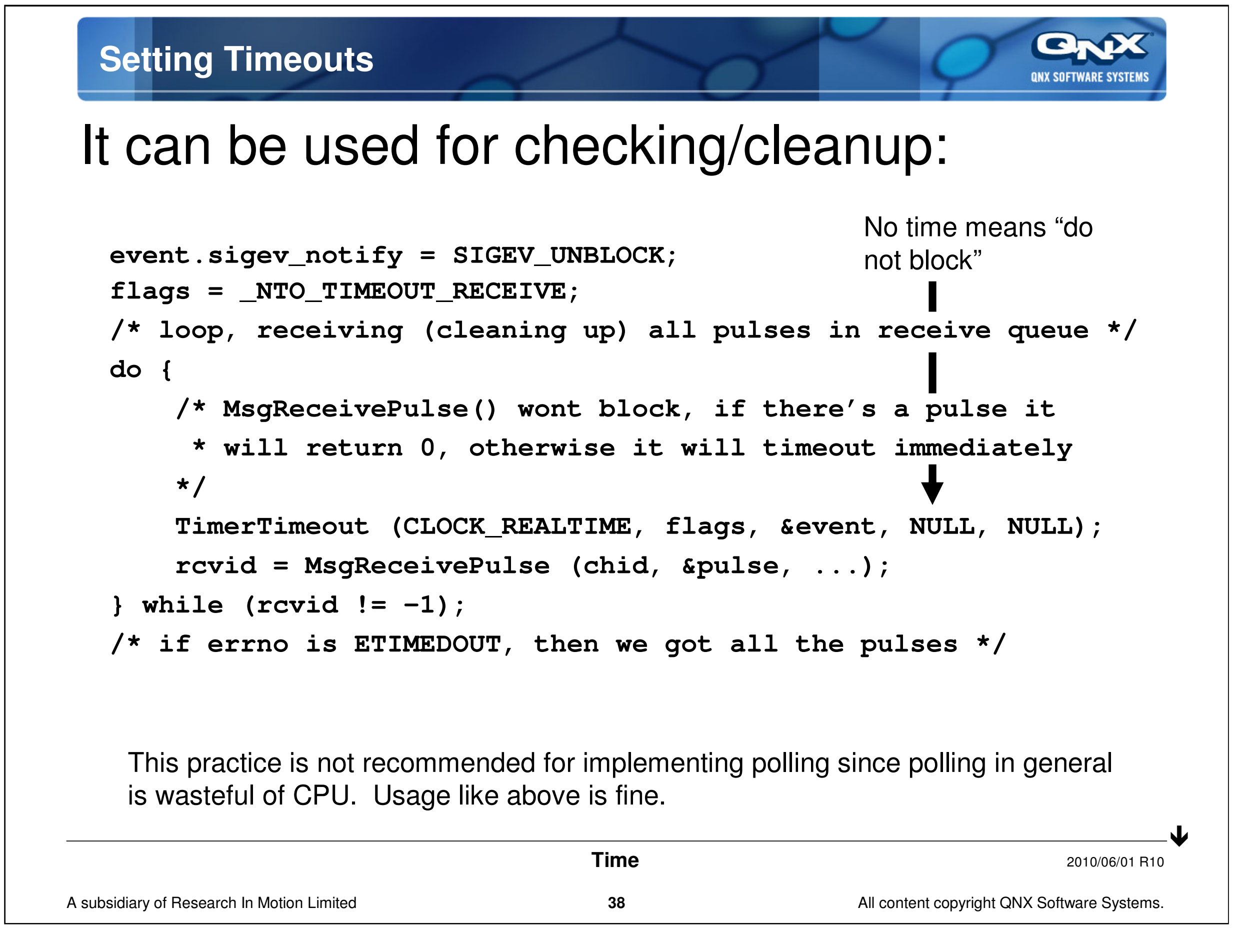
If you don't like working in 64-bit nanosecond values, *timer_timeout()* is a cover function that takes a **`struct itimerspec`** to specify the timeout instead.

Setting Timeouts

Some notes on timeouts:

- timeout is relative to when *TimerTimeout()* is called
- the timeout is automatically cancelled when the next kernel call returns
  - therefore you should not do anything else between the call to *TimerTimeout()* and the function that you are trying to timeout
  - but what if a signal handler is called? Might there be kernel calls made there? The same does not apply if the kernel call is made from within a signal handler that had preempted

NOTES:



Setting Timeouts

It can be used for checking/cleanup:

No time means “do not block”

```
event.sigev_notify = SIGEV_UNBLOCK;
flags = _NTO_TIMEOUT_RECEIVE;
/* loop, receiving (cleaning up) all pulses in receive queue */
do {
    /* MsgReceivePulse() wont block, if there’s a pulse it
     * will return 0, otherwise it will timeout immediately
    */
    TimerTimeout (CLOCK_REALTIME, flags, &event, NULL, NULL);
    rcvid = MsgReceivePulse (chid, &pulse, ...);
} while (rcvid != -1);
/* if errno is ETIMEDOUT, then we got all the pulses */
```

This practice is not recommended for implementing polling since polling in general is wasteful of CPU. Usage like above is fine.

NOTES:

Using *TimerTimeout()* with **`NULL`** for the timeout parameter says, timeout immediately *if we are about to block*.

The above example is good if you’ve received one pulse at your main receive loop and expect that you might have gotten a burst of pulses. Rather than go back to the main receive loop and get each one, we sit in a loop cleaning them all out of our receive queue.

In the example above, if there is a pulse message waiting for us then *MsgReceivePulse()* is not about to block and will therefore not timeout. Instead it will just return with the pulse message. However, if there is no pulse message waiting then *MsgReceivePulse()* is about to block so it times out immediately and returns an error (**`-1`** and **`errno`** set to **`ETIMEDOUT`**).

Time

Topics:

NOTES:

Conclusion

QNX SOFTWARE SYSTEMS

You learned:

- ticksize is the fundamental quantum of time
- how to set or gradually adjust the system time
- how to get periodic notification
- how to timeout kernel calls

NOTES: