CENG 448 — Real Time Operating Systems

Timers and Timer Services

Dr. Larry D. Pyeatt

South Dakota School of Mines and Technology

We Need Timers

System and user tasks often need to schedule activities to occur at specific times, or periodically at a fixed rate.

The scheduler should be called periodically to switch between running tasks with equal priorities, using round-robin or other task selection method.

Devices may need to be polled periodically.

Networking protocols depend on timeouts for retransmission to ensure data integrity.

Monitor software may need to send system information to the host-based analysis tool periodically.

Types of Timer

Hardware timers are physical timer devices that send an interrupt to the CPU when they expire. They are typically flexible enough to be programmed as auto-reset timers or one-shot timers. Most systems provide a small number of hardware timers.

Software timers provide a mechanism for multiple software events to be triggered by one hardware timer (usually the “SysTick” timer).

Hardware: Programmable Interval Timer (PIT)

The PIT (a.k.a. timer device) is a device designed mainly to function as

- an event counter,
- elapsed time indicator,
- rate-controllable periodic event (interrupt) generator,
- or to be used in other ways for solving system timing problems.

Most systems have one or more PIT. There are many designs, but they all have a few things in common.

Timer devices have an input clock source with a fixed frequency, and a set of control registers which allow the timer to be configured.

System Timer

The system timer is often a very specific PIT that is defined as part of the CPU architecture specification.

For example, the Cortex-M3 processor specification includes a 24-bit timer, named SysTick, which must be present on all Cortex-M3 processors: System timer, SysTick

The system timer ISR typically:

- re-sets the system timer (if timer can't reset itself),
- updates the system clock,
- calls the scheduler,
- acknowledges the interrupt (clear flags), and
- returns from the interrupt (it may resume a different task than the one that was running when the interrupt occurred)

All of this ***must*** happen within the space of one system tick.

Hardware Timer Management

The hardware platform may have additional hardware timers.

The RTOS programmer should provide a set of functions to access them. Functions should be provided to:

- allocate a timer from those that are available,
- specify a handler function to call when the timer generates an interrupt,
- set the timeout value for the timer,
- set it up for repeating interrupts or only one interrupt,
- start or stop the timer, and
- deallocate the timer.

Note that the handler (A.K.A. callback) function will be called by an ISR, so it must execute very quickly. In some cases it may be wise to have it do a minimum amount of work, and wake up a task to finish the processing.

Software Timers

Provide mechanism to efficiently schedule non-high-precision software events. If precise timing is required, then a hardware timer must be used, but there are a limited number of hardware timers, and they operate asynchronously to the scheduler.

A software timing facility should provide:

- efficient timer maintenance (should not spend much CPU time counting down for each waiting task)
- efficient timer installation (starting a timer should not take much CPU time)
- efficient timer removal (stopping a timer should not take much CPU time)

An application may have some tasks that require high precision timing, but for most tasks, the timing requirements are not so stringent.

Software Timer Implementation

Software timers are usually implemented using a special “timer task” that is running at very high priority, along with some code implemented within the kernel.

The software timer system must:

- Maintain a list of tasks/functions that are waiting for their timer to run out, along with their current timer value.
- On every tick, decrement all the timers. If a timer reaches zero, schedule execute the associated function, or unblock the associated task and call the scheduler.

FreeRTOS software timers

FreeRTOS timer callback functions execute within the timer task context. So they run at the same priority as the timer task. They must not call `vTaskDelay()` or any other function that could cause the task to block, even briefly.

What if my Callback Function is Long, or Needs to Block?

1. Convert your callback function into a task.
2. Write a new (short) callback function that signals the task to run.

```
void handleTimer()
{
  /* big long code that may block or delay */
}

void setUpTimer() {
  static StaticTimer_t timerControlBlock;
  xTimerCreateStatic("nItimer",pdMS_TO_TICKS(1000 / FPS),
                     pdTRUE,( void * ) 0,handleTimer,
                     &timerControlBlock);
  xTimerStart(nInvader_timer,0);
}
```

Long/Blocking Callback Solution

```
static SemaphoreHandle_t TmrSem;
static StaticSemaphore_t TmrSCB;

void handleTimer_task(void *p){
  while(1) {
      xSemaphoreTake(TmrSem,portMAX_DELAY);
      /* big long code that may block or delay */
}}

void handleTimer() {xSemaphoreGive(TmrSem);}

void setUpTimer() {
  static StaticTimer_t timerControlBlock;
  TmrSem = xSemaphoreCreateCounting(2,0,&TmrSCB);
  xTimerCreateStatic("nItimer",pdMS_TO_TICKS(1000 / FPS),pdTRUE,( void * ) 0,
                     handleTimer,&timerControlBlock);
  xTaskCreate(handleTimer_task,"HandleTimer" .... );
  xTimerStart(nInvader_timer,0);}
```

Delays – Event-driven task-scheduling delay

Timer interrupt occurred,

- Context switch time
- Time to determine that software timer expired
- Time to schedule function to be run by timer task
- Time to finish ISR servicing
- Time to call scheduler
- Context switch time to run the timer task
- Time for timer task to call the function

What if worker function does not have higher priority than all other functions?

Typical System Timer Operations

`sys_timer_enable` enables the system timer interrupt

`sys_timer_disable` disables the system timer interrupt

`sys_timer_connect` installs the system timer ISR

`sys_timer_getrate` returns the number of ticks per second

`sys_timer_setrate` sets number of ticks per second

`sys_timer_getticks` return the number of ticks since system startup

Typical Software Timer Operations

`timer_create` creates a timer and set the callback function and timeout value

`timer_delete` deletes a timer

`timer_start` starts a timer

`timer_cancel` cancels or stops a running timer

`clock_get_time` gets the current clock time

`clock_set_time` sets the current clock time

Typical Hardware Timer Operations

`hardware_timer_allocate` allocate a timer

`hardware_timer_release` deallocate a timer

`hardware_timer_start` start a timer

`hardware_timer_set_timeout` Set the timeout value

`hardware_timer_set_periodic` Set as periodic or one-shot

`hardware_timer_set_handler` set the handler (callback) function

Real-time Clocks and System Clocks

Real-time clocks are hardware devices that track time, date, month, and year. They are often battery-backed (or capacitor-backed) so that they continue to run even when the system is powered-down. (The Raspberry Pi has an RTC, but it is not battery-backed, which means it has to be set every time the Pi is powered-on.)

The System Clock is a software object that is used to track the elapsed time since the system was powered up or rebooted. A programmable timer is used to update the system clock (one “tick” for each timer interrupt).