EE458 - Embedded Systems
Timers and Clocks

• Outline
  – Timers
  – Clocks

• References
  – RTC: Chapter 11
  – CUG: Chapters 9, 10
Timers and Clocks

Timers

- A timer allows the scheduling of an event according to a predefined time value in the future, similar to setting an alarm clock.
- Hard timers are derived from physical timer chips that directly interrupt the processor. Soft timers are software events that are scheduled through software. Operations with demanding precision requirements should use hard timers.
Timers and Clocks

Timers

- Soft timers allow for efficient scheduling of non-high-precision software events. Applications requiring timeouts with coarse granularity may use soft timers. (Many timeouts in network protocols are an example where course granularity is sufficient.)

- Soft timers can also reduce the system-interrupt overhead incurred by multiple hard timers.
Timers and Clocks
Real-Time and System Clocks

- A real-time clock is a part of many embedded systems. It uses battery powered DRAM to keep track of the date and time even when system power is off.

- The job of the system clock is identical to that of the real-time clock. It keeps track of the time and date while the system is running. It is usually initialized from the real-time clock at system start up.
Timers and Clocks
Programmable Interval Timers

- A programmable interval timer (PIT) or timer chip functions as an event counter, elapsed time indicator, or periodic event generator.
- The PIT is usually initialized at system start up: the timer interrupt rate is set, the timer operation mode is set, the timer ISR is installed, and the timer interrupt is enabled.
- Each timer interrupt is known as a tick. The RTOS performs task scheduling at each tick.
Timers and Clocks
Programmable Interval Timers

• The timer ISR typically performs these duties:
  – The system clock is updated, both absolute time (wall-clock time and date) and elapsed time are updated.
  – A kernel routine is called to indicate the passing of a tick.
  – The timer chip is reinitialized (if necessary).
RTEMS Timers

- RTEMS timers are objects and must be created like other kernel objects (tasks, semaphores, etc.) You must also configure the `CONFIGURE_MAXIMUM_TIMERS` parameter correctly.

- RTEMS provides two **soft timers**: task-based (TB) and non-task-based (NTB) timers. They are created and deleted using the same directives. The directives used to fire the timers are different.
Timers and Clocks

RTEMS Timers

- NTB timers run as part of the timer interrupt service routine and should not use any directive that can not be used by an ISR.
- Task-based timer routines are more flexible than NTB routines. The Timer Server can be configured to support a floating point context.
- Task-based timer service routines are executed by the Timer Server task. The timer server task must be accounted for when configuring the system.
Timers and Clocks
Soft (Task-Based) Timers

- The Timer Server task runs at a higher priority than any application task.
- The Timer Server must be initiated (started) before any task-based timer directives are used. The `rtems_timer_initiate_server()` directive is used for this. The directive takes stack size and attribute arguments. These arguments have the same meaning as they do for the `rtems_task_create()` directive.
Timers and Clocks

Firing a Timer

There are four directives that can be used to schedule a timer routine to be run:

- `rtems_timer_fire_after()`: NTB timer that fires after a specified number of ticks.
- `rtems_timer_fire_when()`: NTB timer that fires at a specified wall clock time.
- `rtems_timer_server_fire_after()`: task-based timer that fires after a specified number of ticks.
- `rtems_timer_server_fire_when()`: task-based timer that fires at a specified wall clock time.
Timers and Clocks
Firing a Timer

- Here is the prototype for the rtems_timer_fire_after() directive:

  rtems_status_code rtems_timer_fire_after(
    rtems_id id, rtems_interval ticks,
    rtems_timer_service_routine_entry routine, void *user_data);

- The **id** is the timer id, **ticks** is the number of ticks until the timer “**fires**”, **routine** is the name of the routine to be run when the timer fires, and **user_data** is data that can be passed to the routine.
There are, of course, timer directives to create and delete timers. Deleting an unexpired timer cancels the timer.

The cancel directive can be used to halt a timer. It will not fire unless re-initiated by a reset or fire directive.

The reset directive can restart an unexpired timer initiated by a “fire after” directive.

Refer to CUG Chapter 10 for timer details.
The system clock is set by a call to the `rtems_clock_set()` directive:

```c
rtems_status_code rtems_clock_set(rtems_time_of_day *time_buffer);
```

The `time_buffer` is a structure with year, month, day, hour, minute, second, and ticks fields. (See next slide and CUG Chapter 7).

To use the real-time clock define the macro:

`CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER`
Timers and Clocks
RTEMS Clock Directives

struct rtems_tod_control {
    uint32_t year;   /* greater than 1987 */
    uint32_t month;  /* 1 - 12 */
    uint32_t day;    /* 1 - 31 */
    uint32_t hour;   /* 0 - 23 */
    uint32_t minute; /* 0 - 59 */
    uint32_t second; /* 0 - 59 */
    uint32_t ticks;  /* elapsed between seconds */
};

typedef struct rtems_tod_control rtems_time_of_day;
Timers and Clocks
RTEMS Clock Directives

- The system date and time can be obtained using the `rtems_clock_get_tod()` directive:

  ```c
  rtems_status_code
  rtems_clock_get_tod(
      rtems_time_of_day *time_buffer);
  ```

- This returns the time of day in a structure like that used to set the time. There are also directives to return the number of seconds since the epoch (1/1/1970), ticks per second, ticks since boot, time since boot, etc.