6th semester module

Real-time facilities

Pierre Pompili

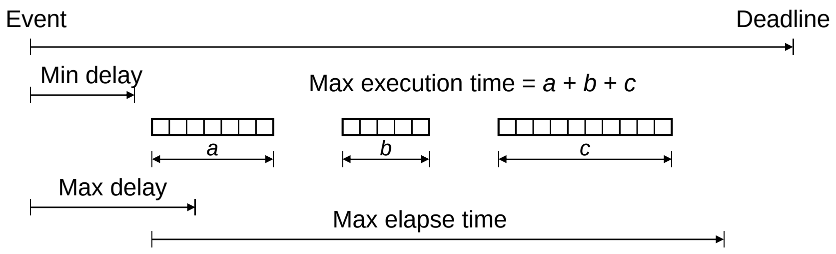
Times requirements

- Time notion into a programming language can be described in terms of four requirements
  - Access to a clock
    - so that the passage of time can be measured
  - Delaying a task
    - so that it is suspended until some future time
  - Programming timeouts
    - so that the non-occurrence of some event, within a specified period of time, can be recognised and dealt with
  - Deadline specification and scheduling
    - so that the necessary time constraints can be specified and met

Temporal scopes

- Specification of real-time applications use the notion of temporal scopes **TS**
  - Deadline
    - the time by which the execution must be finished
  - Minimal delay
    - min amount of time that must elapse before the start of execution
  - Maximal delay
    - max amount of time that must elapse before the start of execution
  - Maximal execution time
    - max serviceable time for the execution
  - Maximal elapse time
    - serviceable time to finish the execution

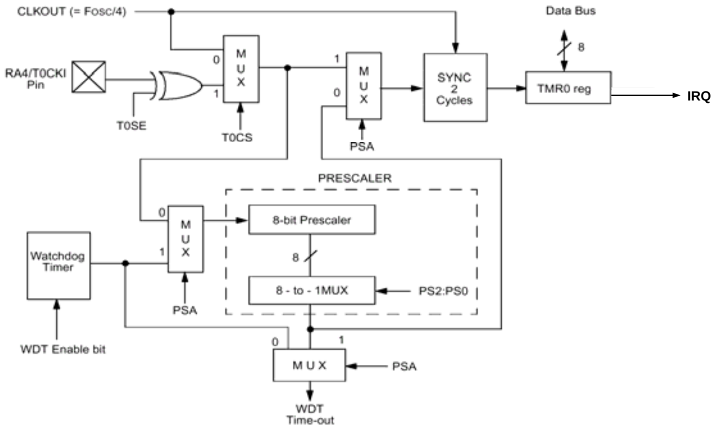
Hard and soft real-time



- A system is said to be **hard real-time** if it has deadlines that cannot be missed. If they are, the system fails
- A system is said to be **soft real-time** if the application is tolerant of missed deadline

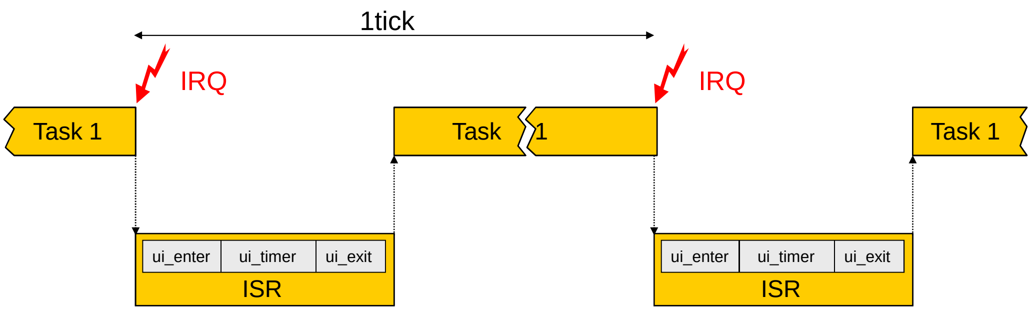
Hard timer

- Hard timer can be set to interrupt the computer after some period time



Clock tick

- Real-time kernel require a periodic time source to keep track of time. **Tick** is the periodic time



Time granularity

- "Timer and clock" manager interfaces with the hard timer provide a real-time clock
  - time structure definition

```
struct  timeSpec
{ unsigned long  seconds;
  unsigned long  nanoseconds;
};
```

  - granularity

Granularity	*Range (approximately)*
1 microsecond	71.6 minutes
100 microseconds	119 hours
1 millisecond	50 days
100 milliseconds	13.6 years

Time structure in C

- In C, time structure is defined in the `time.h` header

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time  (0 to 23   */
};
```

Timers functionality

- Timer model providing the ability to have the timer expire on :
  - an absolute date — one-shot
  - a relative date (*n* nanoseconds from now) — one-shot
  - cyclical (every *n* nanoseconds) — periodic
- One-shot model is used for virtual timer
  - implementation of the communications protocols
- Periodic model is used for continue data flow acquisition
  - sampling period with A/D converters

Virtual timers

- A single periodic hard timer and “Timer and clock” manager can be used to drive multiple **virtual timers**
  - A task may start a virtual timer with a predefined time-out value
  - When the count reaches the time value, a user-specified mask is written to an event flag
- Start a virtual timers and :
  - then wait (i.e., `pend`) on the event flag until it expires
  - go do something else for a while, then pend on the event flag until it expires
  - that will wake another task when it expires

Timer Management

- Virtual timer objects
  - These timer objects can trigger the execution of a function (not threads)
  - When a timer expires, a **callback function** is executed to run associated code with the timer

- Virtual timer models
  - One-shot timer
    - The timer is not automatically restarted once it has elapsed. It can be restarted manually using `osTimerStart` as needed
  - Repeating timer
    - The timer repeats automatically and triggers the callback continuously while running, see `osTimerStart` and `osTimerStop`

Virtual timer on RTX5

```
osTimerId_t osTimerNew ( osTimerFunc_t          func,
                         osTimerType_t          type,
                         void *                 argument,
                         const osTimerAttr_t *  attr);
```

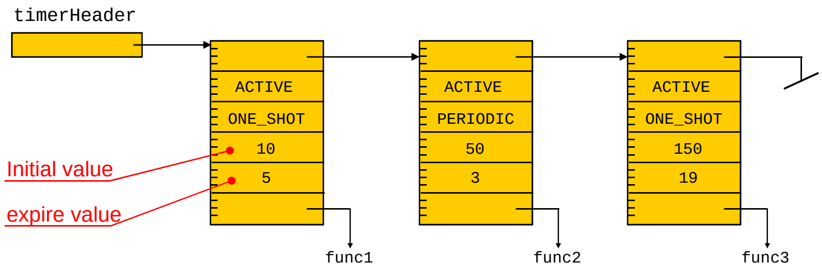
- Parameters

`func`	function pointer to callback function
`type`	`osTimerOnce` for one-shot or `osTimerPeriodic` for periodic behavior
`argument`	argument to the timer callback function
`attr`	timer attributes; `NULL`: default values

```
osStatus_t osTimerStart ( osTimerId_t     timer_id,
                          uint32_t        ticks );
```

Timers linked list

- The linked list is ordered according to the `expires` value
  - only offset time between two timer is stored in `expires` field



  - The manager decrement only the first `expires` value
  - when `expires` arrives at 0, the first element is removed from the list

Real-time clock system calls

- The timer interface counts from zero or from a user-supplied start value (epoch time)
  - many systems use the Universal Coordinated Time (1.1.1970)
- Timer interrupt is called a timer tick
  - each timer-tick increases the internal system clock by one
  - set / get the current value of the *Thread*X internal timer clock in ticks

```
VOID tx_time_set(ULONG new_time)
ULONG tx_time_get(VOID);
```

RTX application timers 1/2

- Example

```
#include "cmsis_os2.h"

void Timer1_Callback (void *arg);               // prototypes for timer callback function
void Timer2_Callback (void *arg);

uint32_t  exec1;                                // argument for the timer call back function
uint32_t  exec2;                                // argument for the timer call back function

void TimerCreate_example (void)  {
  osTimerId_t id_T1;                              // timer id
  osTimerId_t id_T2;                              // timer id

  /*************************************/
  /* Create one-shoot timer           */
  /*************************************/
  exec1 = 1;
  id_T1 = osTimerNew (Timer1_Callback, osTimerOnce, &exec1, NULL);

  /*************************************/
  /* Create periodic timer            */
  /*************************************/
  exec2 = 2;
  id_T2 = osTimerNew (Timer2_Callback, osTimerPeriodic, &exec2, NULL);
}
```

RTX application timers 2/2

```
void Timer1_Callback (void *arg) {            // timer callback function
                                              // arg contains &exec1
                                              // called every second after osTimerStart
                                 }

void Timer2_Callback (void *arg) {            // timer callback function
                                              // arg contains &exec2
                                              // called every second after osTimerStart
                                 }

----------------

  uint32_t     timerDelay;                    // timer value
  osStatus_t   status;                        // function return status

  timerDelay = 500;
  status = osTimerStart (id_T1, timerDelay);  // start one-shoot timer for 500 ticks
  if (status != osOK) {
                                              // Timer could not be started
                      };

  timerDelay = 40;
  status = osTimerStart (id_T2, timerDelay);  // start periodic timer, repeat all 40 ticks
  if (status != osOK) {
                                              // Timer could not be started
                      };
```