Freescale Semiconductor

# Using the MQX Timers

By: Technical Information Center

freescale
semiconductor

# System TICK (SysTick)

The RTOS tick is the operating system time unit. MQX measures time in ticks, instead of in seconds and milliseconds, the features depending on the tick are:

- Time delay
- Timeout while waiting for events, semaphores
- Software timers
- Time-slice scheduling

When MQX RTOS starts, it installs the periodic timer ISR, which sets the time resolution for the hardware. The periodic timer interrupt is used to keep track of time; the priority of the interrupt is higher than tasks.

The tick service routine is located in the init_bsp.c file, the path of this file is
<freescaleMQXpath>\mqx\source\bsp\<board_name>\

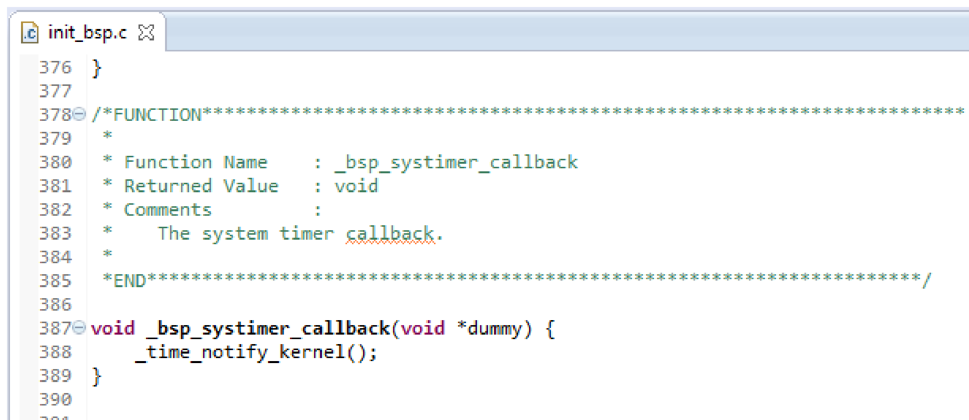


Figure 1 init_bsp.

The resolution defines, how often MQX RTOS updates time, or how often a tick occurs. The resolution is usually 200 ticks per second = 5 milliseconds. It is defined in <board_name>.h file located at the path:
<freescaleMQXpath>\mqx\source\bsp\<board_name>\

```
twrk64f120m.h
** The clock tick rate in miliseconds - be cautious to keep this value such
** that it divides 1000 well
**
** MGCT: <option type="number" min="1" max="1000"/>
*/
#ifndef BSP_ALARM_FREQUENCY
    #define BSP_ALARM_FREQUENCY             (200)
#endif

/*
** System timer definitions
*/
#define BSP_SYSTIMER_DEV          systick_devif
#define BSP_SYSTIMER_ID           0
#define BSP_SYSTIMER_SRC_CLK      CM_CLOCK_SOURCE_CORE
#define BSP_SYSTIMER_ISR_PRIOR    2
/* We need to keep BSP_TIMER_INTERRUPT_VECTOR macro for tests and watchdog.
 * Will be removed after hwtimer expand to all platforms */
#define BSP_TIMER_INTERRUPT_VECTOR INT_SysTick

/** MGCT: </category> */

/*
** Old clock rate definition in MS per tick, kept for compatibility
*/
#define BSP_ALARM_RESOLUTION                (1000 / BSP_ALARM_FREQUENCY)

/*
```

Figure 2 twrk64f120m.

If there is no tick, then it is not possible to use the time delay, timeouts, and software timers; so in this case it is necessary to use hardware timer interrupts or other interrupts to synchronize tasks.

The BSP timer is clocked by system tick; it is possible to edit **BSP_ALARM_FREQUENCY** in order to have smaller ticks.

Delays are calculated in the following way:

$$N_TICKS = \frac{REQ_ms \times ticks_per_s}{1000 \quad ms}$$

**N_TICKS** = Number of ticks
**REQ_ms** = Number of milliseconds requested
**ticks_per_s** = Number of ticks per second

For example, if it is used the function ***_time_delay (55)*** then it is required a 55ms delay. If **BSP_ALARM_FREQUENCY** is set as default 200 then:

$$N_TICKS = \frac{55 \quad ms \times 200 \quad ticks}{1000 \quad ms} = 11$$

We have to also take into consideration inaccuracy due to principle of this timer. Argument of **_time_delay()** function is minimum number of milliseconds to suspend the task because time between last tick and call of **_time_delay** could vary in range 0..1 tick. It means that jitter in case of small delays could be high.

If the **BSP_ALARM_FREQUENCY** is modified for example to 1000 then it is possible to get a minimum 1ms delay.

$$REQ_ms = \frac{1_tick \times 1000_ms}{1000_ticks} = 1ms$$

A task can get the resolution in milliseconds with **_time_get_resolution()** and in ticks per second with **_time_get_resolution_ticks().**

A task can get elapsed time in microsecond resolution by calling **_time_get_elapsed()**, followed by **_time_get_microseconds()**, which gets the number of microseconds since the last periodic timer interrupt.

A task can get elapsed time in nanosecond resolution by calling **_time_get_elapsed()** followed by **_time_get_nanoseconds()**, which gets the number of nanoseconds since the last periodic timer interrupt.

A task can also get the number of hardware ticks since the last interrupt by calling **_time_get_hwticks()**. A task can get the resolution of the hardware ticks by calling **_time_get_hwticks_per_tick().**

Eg.

```
uint32_t nanoseconds, i, counter = 0;
MQX_TICK_STRUCT ticks1, ticks2;
bool overflow;

            _time_get_ticks(&ticks1);   /* get start time */
            for (i = 0; i < 1000; i++)
            {
              counter++;
            }
            _time_get_ticks(&ticks2);   /* get end time */
            nanoseconds = _time_diff_nanoseconds (&ticks2, &ticks1, &overflow); /* calculate difference between
start time and end time in nanoseconds*/
```

Function **_time_diff_nanoseconds** doesn't mean that we measure in ns granularity. Granularity is given by HWTICKS.

In conclusion if you want to have smaller ticks it is necessary to modify the **BSP_ALARM_FREQUENCY** value, however it is necessary to remember, we cannot guarantee the correct behavior if you modify original settings. Some of drivers, for example USB, Ethernet PHY, ESDHC, etc, use commands like _time_delay; if the resolution is changed then we could get unexpected behavior of these drivers.
**Only change this value if you know what you are doing. BSP_ALARM_FREQUENCY higher than 1000 (1ms tick time) could also affects some of drivers due to rounding in time equation.**

If it is required some precise time interrupt without modify the SysTick within the BSP then it is possible to use **MQX hwtimer driver**.

## MQX hwtimer driver

The MQX hwtimer driver provides a C language API for uniform access to the features of various HW timer modules such as Periodic Interrupt Timer (PIT) and Low-Power Timer (LPT ).

The source code for HWTIMER drivers is located in source <freescaleMQXpath>\mqx\source\io\hwtimer\ directory.

The <borad_name>.h file located at the path <freescaleMQXpath>\mqx\source\bsp\<board_name>\ contains the definitions for the HWTIMER driver.

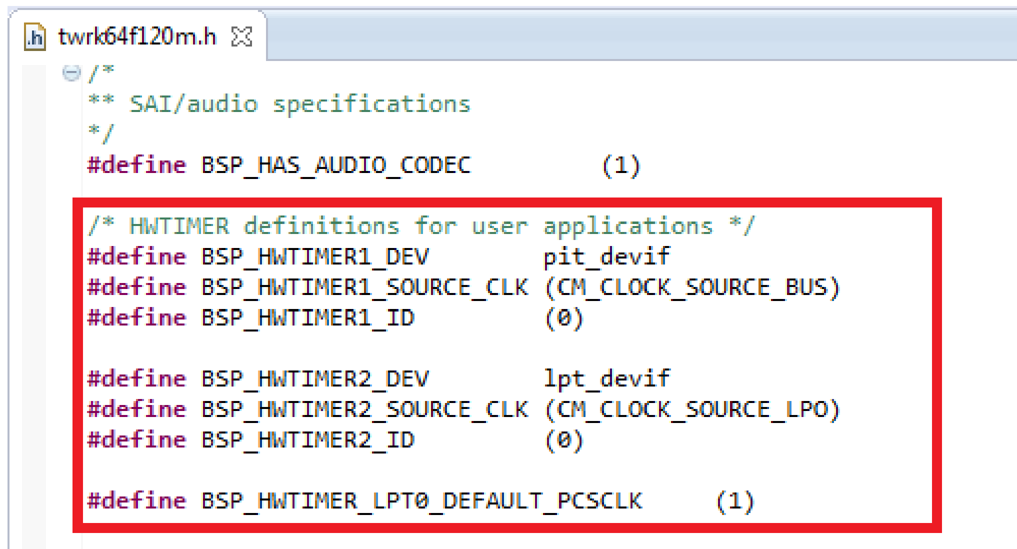


Figure 3 twrk64f120m.

You can modify these definitions and add here your additional timers. Please check MCU reference manual for available timer resources.

For Kinetis devices, when the LPT module is used with this driver (**BSP_HWTIMER2_DEV**), the default clock source is provided by low power oscillator (LPO), the LPO is a 1KHz clock that is enabled in all modes of operation, including all low power modes. This means that the LPT can be set only times with 1ms granularity.

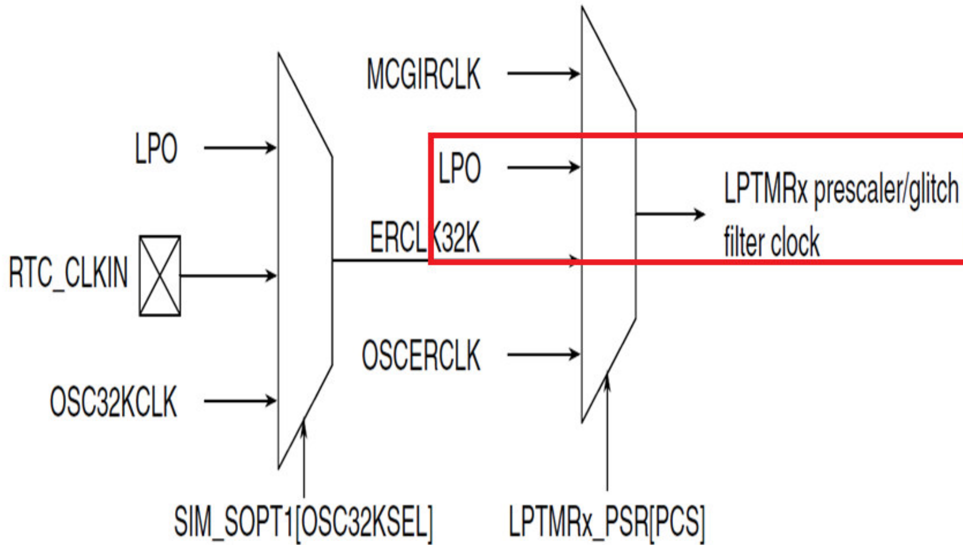


Figure 4 K64F clock sources.

The PIT module is an array of timers (channels). The MQX hwtimer driver access to the PIT module using the **BSP_HWTIMER1_DEV** definition, then it is required to select the channel using **BSP_HWTIMER1_ID**.

**#define BSP_HWTIMER1_ID (0)** : select the PIT timer 0,
...
**#define BSP_HWTIMER1_ID (n)** : select the PIT timer n,

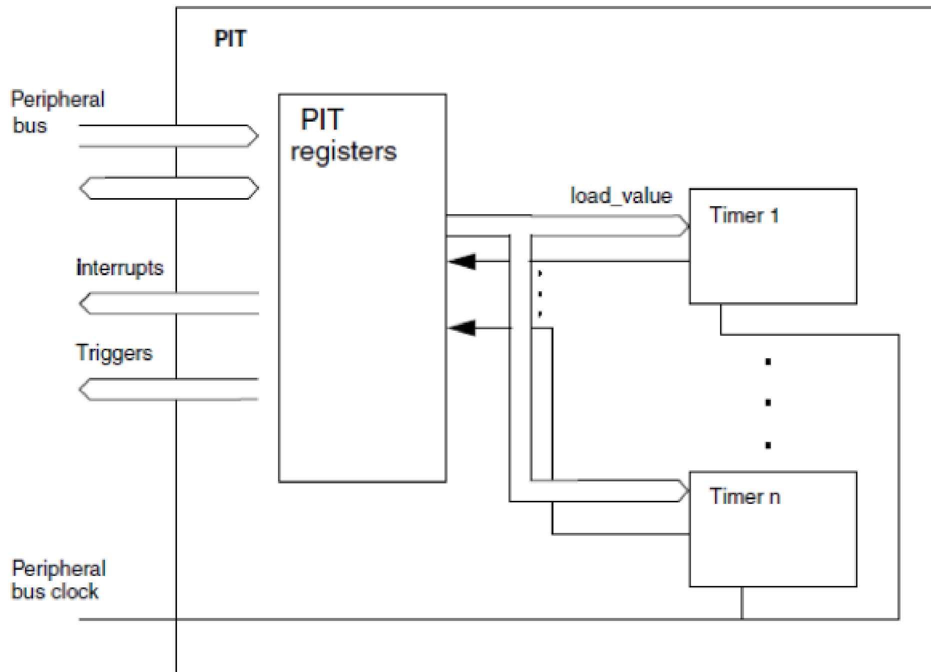


Figure 5 Block diagram of the PIT.

The number of channels that can be used depends on the device, so it is necessary to check the reference manual for each device.

The Getting Started with Freescale MQX™ RTOS document provides details about all boards and BSPs supported, in this document it is possible to find the bus clock value for every board. For example the Bus Clock for the TWR-K64F120M is 60MHz.

**TWR-K64F120M**

| | | |
|---|---|---|
| Core Clock | 120 Mhz | - |
| Bus Clock | 60 Mhz | - |
| Default Console | ttyb: | OpenSDA - USB mini connector |
| BSP Timer | Systick | - |

Figure 6 Table from the Getting Started with Freescale MQX™ RTOS.

# MQX hwtimer driver example

This example has only one task (main_task) in this task are configured the LWGPIO and the HWTIMER drivers in order to toggle a pin.

The **hwtimer_init ()** function has to be called prior to calling any other API function of the HWTIMER driver. This function initializes the HWTIMER structure.
The parameters for this function are:

- **HWTIMER_PTR:** The pointer to the HWTIMER is passed as a handle parameter (HWTIMER_PTR). It is necessary to have a variable to assign the pointer to the HWTIMER structure, for do that it is required to add the following line:

  **HWTIMER hwtimerLPT;**

- **HWTIMER_DEVIF_STRUCT_PTR:** The device interface pointer determines low layer driver to be used (PIT or LPT). Device interface structure is exported by each low layer driver and is opaque to the applications. The **BSP_HWTIMER1_DEV** is for PIT and **BSP_HWTIMER2_DEV** is for LPT, please check the below image.

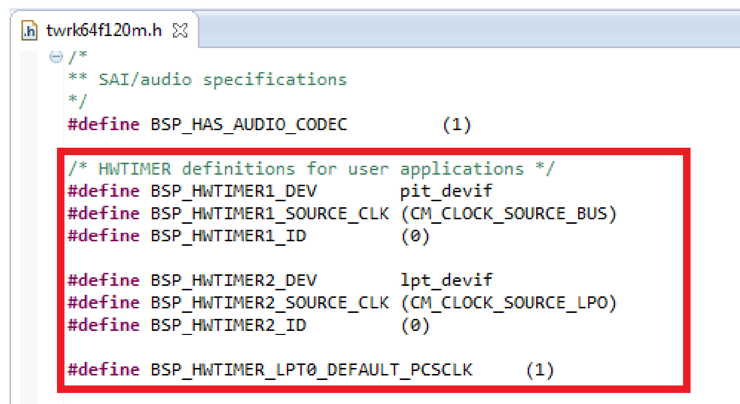


Figure 7 twrk64f120m.

- **id:** The meaning of the numerical identifier varies depending on the low layer driver used. Typically, it identifies a particular timer channel to initialize. For example the figure 1, shows that it is define **BSP_HWTIMER1_ID (0)**, so it is used the PIT channel 0.

The following line initializes the hwtimer driver using the LPT.

**hwtimer_init(&hwtimerLPT, &BSP_HWTIMER2_DEV, BSP_HWTIMER2_ID, (BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX + 1))**

Once it is initialized, it is necessary to configure the timer. For do that it is possible to use the functions:

- **hwtimer_set_freq():** This function configures the timer to tick at a frequency as closely as possible to the requested one. Actual accuracy depends on the timer module. The function gets the value of the base

frequency of the timer via the clock manager, calculates required divider ratio, and calls the low layer driver to set up the timer accordingly.

- **hwtimer_set_period():** This function provides an alternate way to set up the timer to a desired period specified in microseconds rather than to a frequency in Hertz. The function gets the value of the base frequency of the timer via the clock manager, calculates required divider ratio, and calls the low layer driver to set up the timer accordingly.

For this example the hwtimer_set_period() is used, the parameters for this function are:

- **Hwtimer:** Pointer to hwtimer structure, for this example the pointer was called hwtimerLPT.
- **clock_id:** Clock identifier used for obtaining timer's source clock. According with the definitions in the board_name.h file, the source clock used for LPT is LPO and for PIT is BUS CLOCK.
- **Period:** Required period of the timer in us. For a better organization, in this example there is a definition for set the period **#define HWTIMER2_PERIOD 1000000.**

The following line set up the timer.

**hwtimer_set_period(&hwtimerLPT, BSP_HWTIMER2_SOURCE_CLK, HWTIMER2_PERIOD)**

In order to check if the frequency or period fit what we set, the hwtimer_get_freq(), hwtimer_get_period(), should be used.

**printf("Read period from hwtimerLPT    : %d us\n", hwtimer_get_period(&hwtimerLPT));**

Then it is necessary to use the function hwtimer**_callback_cancel().** This function cancels pending callback, if any.

**hwtimer_callback_cancel(&hwtimerLPT)**

The next step is to use the **hwtimer_callback_reg()** function, this function registers function to be called when the timer expires. The callback_data is arbitrary pointer passed as parameter to the callback function. This function must not be called from a callback routine.

The parameters for **hwtimer_callback_reg()** function are:

- **Hwtimer:** Pointer to hwtimer structure, for this example the pointer was called hwtimerLPT.
- **callback_func:** Function pointer to be called when the timer expires. In this example the function that is called is named hwtimer1_callback.
- **callback_data:** Arbitrary pointer passed as parameter to the callback function.

The below line is used.

**hwtimer_callback_reg(&hwtimerLPT,(HWTIMER_CALLBACK_FPTR)hwtimerLPT_callback, NULL)**

Finally it is necessary to enable the timer in order it stars running. For do that the function used is **hwtimer_start().** After this function the timer starts counting and generating interrupts each time it rolls over.

At this point it is possible to build the project, flash it to the board and run the project. For code details, check the main.c file below.

## main.c file

```
/******************************************************************************
*
*   This file contains MQX example code.
*
*****************************************************************************/
#include "main.h"

#if !BSPCFG_ENABLE_IO_SUBSYSTEM
#error This application requires BSPCFG_ENABLE_IO_SUBSYSTEM defined non-zero in user_config.h. Please recompile
BSP with this option.
#endif

#ifndef BSP_DEFAULT_IO_CHANNEL_DEFINED
#error This application requires BSP_DEFAULT_IO_CHANNEL to be not NULL. Please set corresponding
BSPCFG_ENABLE_TTYx to non-zero in user_config.h and recompile BSP with this option.
#endif

#define HWTIMER_PERIOD     1000000                //period set to 1s to hwtimer

static void hwtimerLPT_callback(void *p);
HWTIMER hwtimerLPT;                               //hwtimer handle
LWGPIO_STRUCT led1;

TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
/*  Task number, Entry point, Stack, Pri, String, Auto? */
   {MAIN_TASK,   Main_task,   1500,  9,   "main", MQX_AUTO_START_TASK},
   {0,           0,           0,     0,   0,      0,                  }
};

/*TASK*-----------------------------------------------------
* Task Name    : Main_task
* This task enables the LWGPIO and HWTIMER *
*END*-----------------------------------------------------*/

static void hwtimerLPT_callback(void *p)
{
        lwgpio_toggle_value(&led1); /* toggle pin value */
}

void Main_task(uint32_t initial_data)
{

/*************************LWGPIO DRIVER****************************/

   lwgpio_init(&led1, BSP_LED1, LWGPIO_DIR_OUTPUT, LWGPIO_VALUE_NOCHANGE  );    /* initialize lwgpio driver */
   lwgpio_set_functionality(&led1, BSP_LED1_MUX_GPIO );                         /* swich pin functionality (MUX)
to GPIO mode */

/***********************HWTIMER DRIVER**********************************************/

   hwtimer_init(&hwtimerLPT, &BSP_HWTIMER2_DEV, BSP_HWTIMER2_ID, (BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX +
1));    /* initialize HWTIMER driver */
   hwtimer_set_period(&hwtimerLPT, BSP_HWTIMER2_SOURCE_CLK, HWTIMER_PERIOD);
/* set up the timer in microseconds */
printf("Read period from hwtimerLPT    : %d us\n", hwtimer_get_period(&hwtimerLPT));
   hwtimer_callback_cancel(&hwtimerLPT);
// Clear pending callback for hwtimer1
   hwtimer_callback_reg(&hwtimerLPT,(HWTIMER_CALLBACK_FPTR)hwtimerLPT_callback, NULL);
/* registers function to be called when the timer expires. */
   hwtimer_start(&hwtimerLPT);           /* enables the timer and gets it running. The timer starts counting and
generating interrupts each time it rolls over. */

}
/* EOF */
```