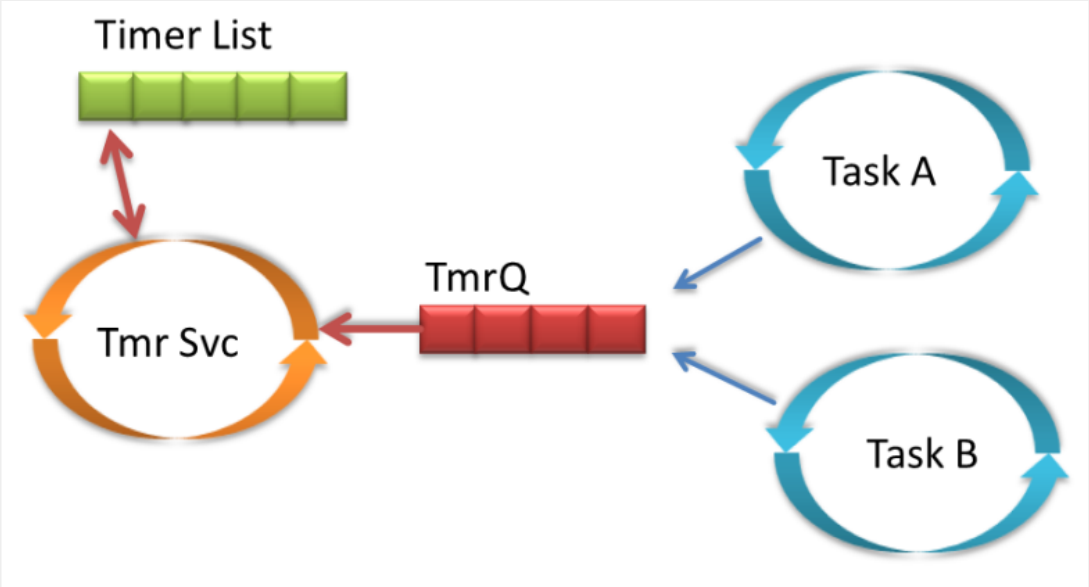
Tutorial: Understanding and Using FreeRTOS Software Timers

Posted on **May 27, 2018** by **Erich Styger**

Hardware Timers are essential to most embedded applications: I use them mostly for triggering actions at a given frequency, such as acquiring data from a sensor. With using an RTOS I can do a similar thing using a task: the task will run with a given frequency and I can periodic work in it. However, using a task might be too much overhead doing this. The good news is that there is a much more efficient way to do this in FreeRTOS with Software Timers. And this is what this tutorial is about: how to use Software Timers with FreeRTOS.



— FreeRTOS Software Timers

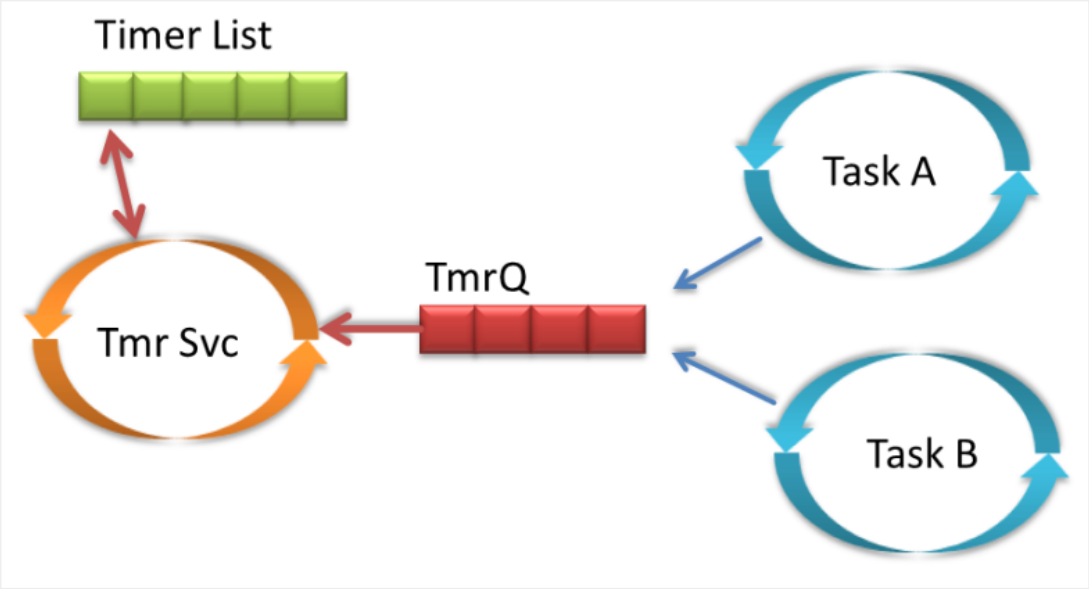
Outline

In this tutorial, I show how to create FreeRTOS Software Timers and how to use them. I have put the example code in a project on GitHub (MCUXpresso IDE, but applicable for any other IDE too). If you want to add the code used in this tutorial to your own project, make sure you have a working FreeRTOS project first.

💡 *I'm using MCUXpresso IDE 10.0.2 and FreeRTOS 10.0.1 in this article.*

How it works

The following diagram gives an overview, how Software Timers in FreeRTOS are implemented:



— FreeRTOS Software Timers

There is a dedicated '**Tmr Svc**' (Timer Service or Deamon) Task which maintains an ordered list of software timers, with the timer to expire next in front of the list). The Timer Service task is not continuously running: from the Timer List the task knows the time when he as to wake up each time a timer in the timer list has expired. When a timer has expired, the Timer Service task calls its callback (the Timer callback).

The other concept the Timer task is using is a queue: this queue is used for inter-process communication. Using that queue, other tasks can send commands to the timer task, for example to start or stop a timer. The Timer Task will wake up when something is sent to that queue. And that way the Timer API has parameters like 'ticksToWait' to specify the waiting time if the timer queue is full.

One-Shot and Auto-Reload

FreeRTOS software timers support two different software timer, configured at timer creation time:

- **One-Shot**: when the timer expires, it is not restarted again. I still can restart it at a later time. Making it ideal for a 'one-shot' kind of thing.
- **Auto-Reload**: this timer will automatically be restarted when it expires, making it ideal for periodic-kind of things.

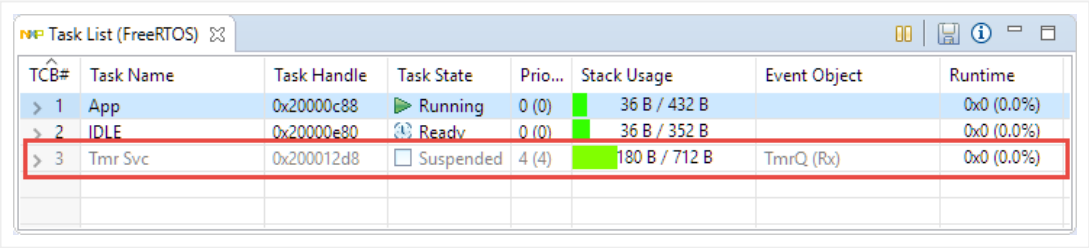
Timer Task and FreeRTOSConfig.h

To use FreeRTOS timers, you have to turn them on with the following entry in FreeRTOSConfig.h:

```
#define configUSE_TIMERS 1
```

If you are not using FreeRTOS software timers, set that macro to 0, otherwise your application is using more resources than necessary. Because this creates the 'Tmr Svc' task during

vTaskStartScheduler():



— TmrSvc Created

In above screenshot (Event Object column) you can see as well that the Tmr Svc is waiting for receiving an object in the TmrQ queue.

This timer task is responsible to handle all FreeRTOS software timers in the system. Basically it checks if a timer has been expired and calls the associated timer hook.

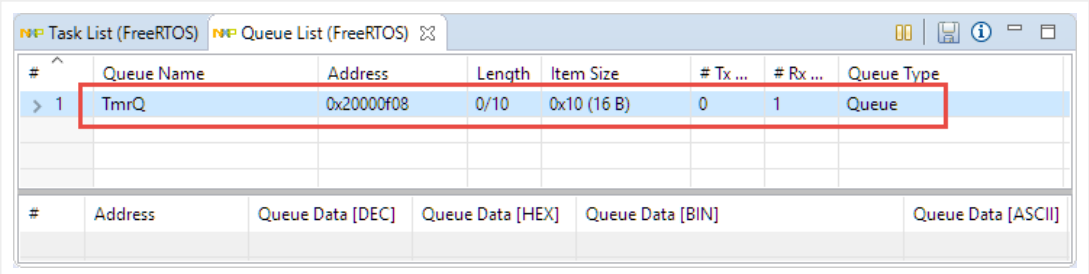
The timer task name, priority and stack size can be configured with the following macros in FreeRTOSConfig.h

```
#define configTIMER_SERVICE_TASK_NAME "Tmr Svc"
#define configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1)
#define configTIMER_TASK_STACK_DEPTH (configMINIMAL_STACK_SIZE * 2)
```

I recommend to give the timer task the highest task priority in the system, otherwise you will see some latency in the timer hook execution. The timer stack size really depends on what you are doing in the timer hooks called from the timer task. To find out what your tasks are using on the stack, see "Understanding FreeRTOS Task Stack Usage and Kernel awareness Information".

Timer Queue and FreeRTOSConfig.h

The other thing which gets created is a queue for the timer task, named 'TmrQ':



— TmrQ Queue for Timer Task

💡 *What you see here is the **queue** for the timer task, not the list of timers created.*

This queue is used for IPC (Inter-Process Communication) between the timer task and the other tasks of the system. The length of the queue can be configured with the following macro, while the name of the queue “TmrQ” is hard-coded in the RTOS:

```
#define configTIMER_QUEUE_LENGTH 10
```

The question might be: what is a useful length for that queue? Because the longer, the more RAM is used. Basically the queue needs to hold as many command items which can be sent by other tasks (or interrupts) until they can be served by the Timer Task. With the Timer task having the highest priority in the system, the queue can be smaller. But if multiple commands can be sent from higher priority tasks or from interrupts, make sure your queue is long enough for this.

Creating a Software Timer

In this example, I’m going to create a software timer which blinks a LED every second. To create a new software timer, I need a timer handle for it:

```
xTimerHandle timerHndl1Sec;
```

Below is the API to create a new timer:

```
TimerHandle_t xTimerCreate( const char * const pcTimerName,
  const TickType_t xTimerPeriodInTicks,
  const UBaseType_t uxAutoReload,
  void * const pvTimerID,
  TimerCallbackFunction_t pxCallbackFunction )
```

To create a new timer, I use:

```
timerHndl1Sec = xTimerCreate(
      "timer1Sec", /* name */
      pdMS_TO_TICKS(1000), /* period/time */
      pdTRUE, /* auto reload */
      (void*)0, /* timer ID */
      vTimerCallback1SecExpired); /* callback */
if (timerHndl1Sec==NULL) {
  for(;;); /* failure! */
}
```

The first parameter is a string for the timer name, followed by the timer (initial) period in RTOS timer ticks. The next parameter with ‘pdTRUE’ requests an ‘auto-reload’ timer, which means the timer is a periodic one, and not a one-shot timer. Next I can provide a timer ID (pointer to void) and finally the most important thing: the timer callback. The callback is the function which will get called when the timer expires.

💡 *Naturally, the shortest period time you can reach with a FreeRTOS software timer is a single tick period. So if your FreeRTOS is running with a 1 kHz tick period, you only can implement a 1 kHz software timer that way. If it needs to be faster, you have to consider using a hardware timer.*

The **xTimerCreate()** only creates the timer, but does not start it. This will be the next topic.

Because a timer creation can fail (e.g. running out of memory), it is good practice to check the returned timer handle.

Timer Callback

During **xTimerCreate()** I had to specify a callback (function pointer). I want to toggle an LED, so my callback looks like this:

```
static void vTimerCallback1SecExpired(xTimerHandle pxTimer) {
  GPIO_PortToggle(BOARD_INITPINS_LED_RED_GPIO, 1<<BOARD_INITPINS_LED_F
}
```

As the timer callbacks are called from the Timer Service task, it is important that the callback function does not block (e.g. waits for some time or waits for a semaphore), as otherwise all other timers get delayed. So the general rule for a timer callback function (as for interrupt service routines) is: keep it short and simple!

Starting a Timer

A timer perviously created with **xTimerCreate()** gets started with

```
BaseType_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait
```

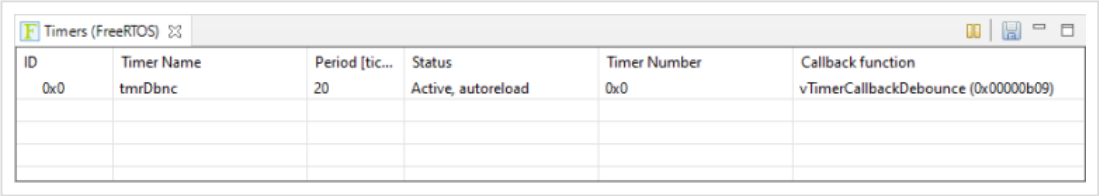
Here the 'ticksToWait' indicates the queue API and specifies how long the caller shall wait if the timer queue is already full.

I create a timer with

```
if (xTimerStart(timerHndl1Sec, 0)!=pdPASS) {
  for(;;); /* failure!?! */
}
```

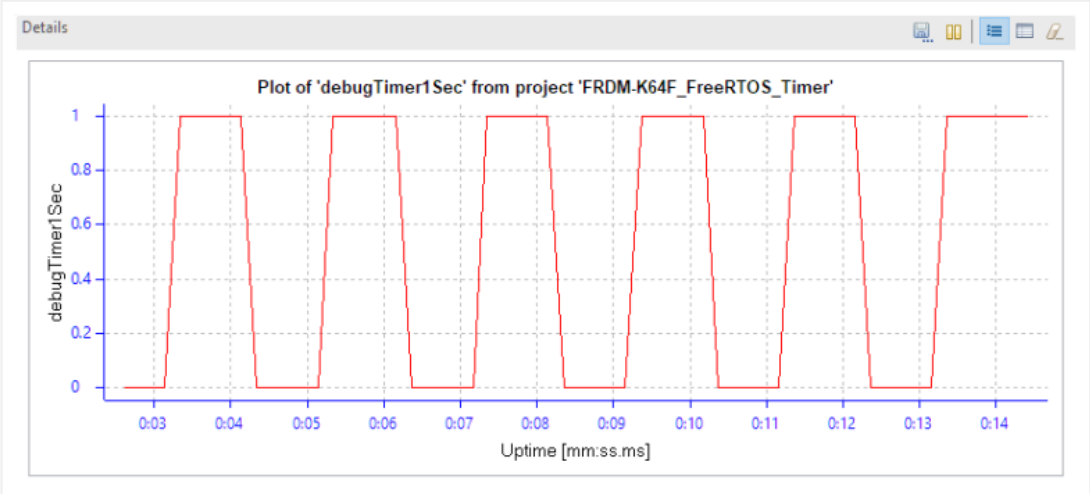
The above code will fail if the timer queue will not be large enough. This will start the timer with the given period (at creation time). If the timer is started before the scheduler is started, the timer will start to run at scheduler start time. If the timer is started say from a task, it starts at the current tick time at which **xTimerStart()** is called.

With a timer active, it is shown in the FreeRTOS Timer view in Eclipse:



This list only shows timers which are 'active' or running, but not otherwise.

Below is a live plot of the 1 second software timer in MCUXpresso IDE (Eclipse based):



— Plot of 1 Sec Timer

Resetting a Timer

With

```
BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait
```

a timer can be reset. And if that timer has not already been started, it will start the timer. I'm going to use that for my next example. For a display I want to keep the backlight on for 5 seconds, and when the user presses a button, that 5 seconds shall start again. So that timer acts as a timeout timer: if it does not get reset by a button, it will expire after 5 seconds.

Again, I create a software timer. But this time I do not start it:

```
xTimerHandle timerHndl5SecTimeout;

timerHndl5SecTimeout = xTimerCreate(
  "timerGreen", /* name */
  pdMS_TO_TICKS(5000), /* period/time */
  pdFALSE, /* auto reload */
  (void*)1, /* timer ID */
  vTimerCallback5SecExpired); /* callback */
  if (timerHndl5SecTimeout==NULL) {
```

```
    for(;;); /* failure! */
}
```

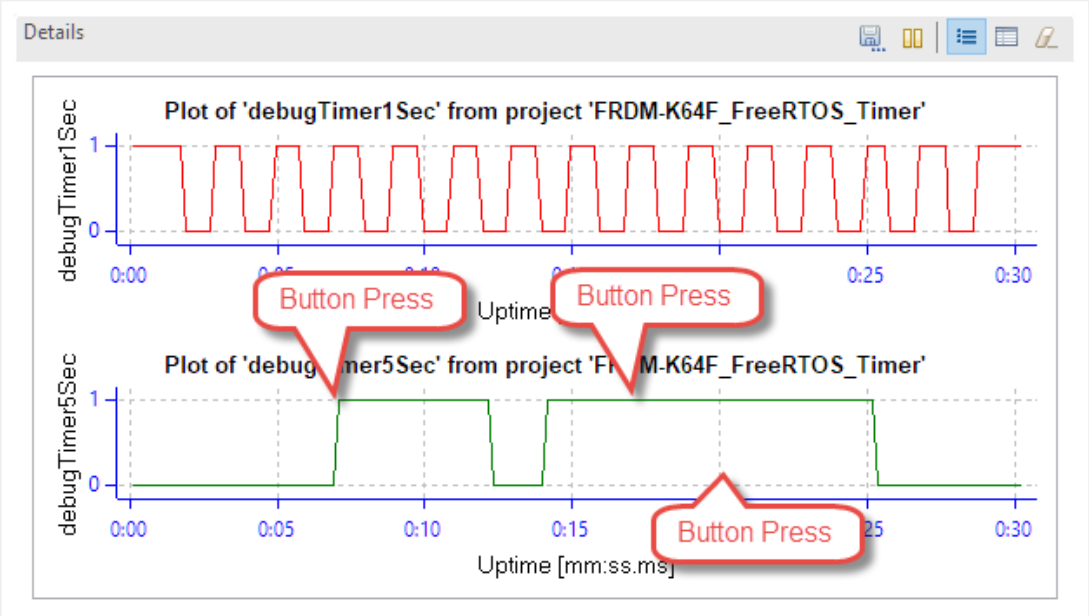
The callback looks like this:

```
static void vTimerCallback5SecExpired(xTimerHandle pxTimer) {
  /* this timer callback turns off the green LED */
  GPIO_PortSet(BOARD_INITPINS_LCD_BACKGROUND_LIGHT, BOARD_INITPINS_LCI
}
```

Inside a task I'm checking the user buttons (e.g. for menu navigation). Here I turn on the LCD backlight and start the timeout timer:

```
if (!GPIO_PinRead(BOARD_INITPINS_SW3_GPIO, BOARD_INITPINS_SW3_PIN)) {
  GPIO_PortClear(BOARD_INITPINS_LCD_BACKGROUND_LIGHT, BOARD_INITPINS_I
  if (xTimerReset(timerHndl5SecTimeout, 0)!=pdPASS) { /* start timer t
    for(;;); /* failure?!? */
  }
}
```

So when the user presses a button, it will restart the timer, and if it expires, it will turn off the LCD backlight. Below a plot with the two timers in action:

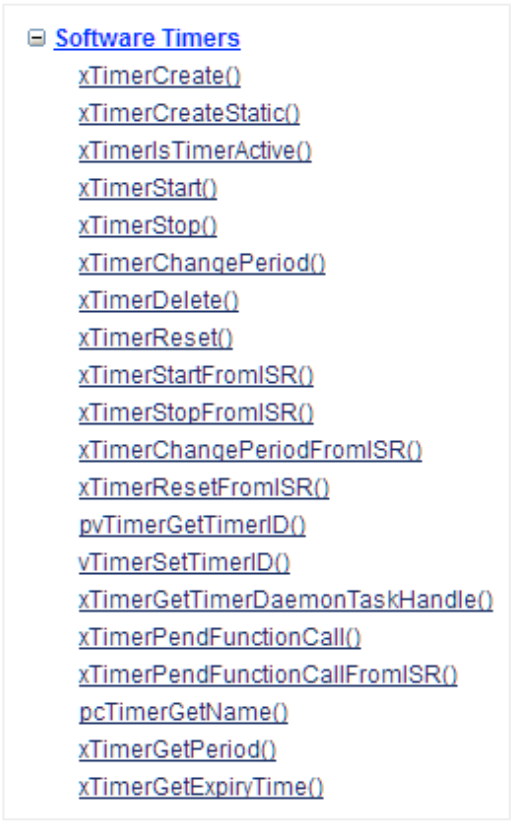


— LCD Backlight with FreeRTOS Sotware Timers

Other Timer Functions

The FreeRTOS Software Timer API provides more:

- Changing timer period
- Deleting a timer
- Get and Set the Timer ID
- Get timer name, period and expiry time
- Get the timer task handle



— FreeRTOS Software Timer Functions

Timer functions can be called from an interrupt service routine, if they have the **FromISR** in the name.

A rather special one is **xTimerPendFunctionCall()**: With that API method I can send to the timer deamon task a callback/function pointer to be called. Basically a 'dear timer task, please execute this function for me'. This function will be called immediately (and only once) when the timer deamon tasks removes the command from the timer queue.

Summary

Software timers are very useful: I can implement multiple periodic or one-shot/timeout timers with a single timer (deamon) task. That way I don't need extra hardware timers and I need less resources than using a task for each periodic action.

Happy Timing 🙂

LINKS

- Example code of this tutorial on GitHub: https://github.com/ErichStyger/mcuoneclipse/tree/master/Examples/MCUXpresso/FRDM-

K64F/FRDM-K64F_FreeRTOS_Timer

- https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html
- https://www.freertos.org/RTOS-software-timer.html

This entry was posted in **Building**, **CPU's**, **Debugging**, **Eclipse**, **FreeRTOS**, **MCUXpresso IDE**, **NXP**, **Tips & Tricks**, **Tutorial**, **Uncategorized** and tagged **Building**, **Eclipse**, **FreeRTOS**, **NXP**, **software**, **software project**, **Software Timer**, **technology**, **Tips&Tricks**, **Tutorial** by **Erich Styger**. Bookmark the **permalink [https://mcuoneclipse.com/2018/05/27/tutorial-understanding-and-using-freertos-software-timers/]** .

About Erich Styger

Embedded is my passion....

View all posts by Erich Styger →

5 THOUGHTS ON “TUTORIAL: UNDERSTANDING AND USING FREERTOS SOFTWARE TIMERS”

Marceli Firlej

on **May 28, 2018 at 06:53** said:

Also is very easy is to implement with SysTick component.

```
// update to match use timers' number
#define SYSTEM_TIMER_MAX 7

#define SYSTEM_TIMER 0
#define HEART_BEAT 1
#define FLASH_TIMER 2
#define SEC_TIMER 3
#define RATE_TIMER 4 // communication rate to USB
#define ADC_TIMER 5 // measurements rate
#define USB_TIMER 6

//#define DEBOUNCE_TIMER 7
//#define MEASURE_TIMER 8
//#define PROCESS_TIMER 9
//#define TRIP_TIMER 10

// Timer Values (based on 1ms system timer)
// Timer Values (based on 1ms system timer done in SYSTICK)
#define T1MS 1
#define T2MS 2
#define T5MS 5
#define T10MS 10
#define T15MS 15
#define T25MS 25
```

```
#define T50MS 50
#define T100MS 100
#define T200MS 200
#define T500MS 500
#define T1SEC 1000
#define T30S 30000 // System not response timeout (failure)

void init_system_time(void);
unsigned long get_system_time(void);
void set_system_timeout(unsigned char timer_number, unsigned short timeout);
unsigned short get_system_remaining_time(unsigned char timer_number);
unsigned char is_system_timeout(unsigned char timer_number);
void system_delay(unsigned short timeout);
void increment_system_time(void);

// in source have:
//=========================================
PE_ISR(ISR_SYSTICK) // set to 1ms
{
// NOTE: The routine should include actions to clear the appropriate
// interrupt flags. Relocated due to Processor Expert compilation
increment_system_time();
}

void init_system_time(void)
{
unsigned char i;

system_time = 0;
for (i = 0; i < SYSTEM_TIMER_MAX; i++)
system_timer[i] = 0;
}

unsigned long get_system_time(void)
{
return system_time;
}

void set_system_timeout(unsigned char timer_number, unsigned short timeout)
{
if (timer_number < SYSTEM_TIMER_MAX)
system_timer[timer_number] = timeout;
}

unsigned short get_system_remaining_time(unsigned char timer_number)
{
if (timer_number < SYSTEM_TIMER_MAX)
return (system_timer[timer_number]);
```

```
else
return 0;
}

void system_delay(unsigned short timeout)
{
set_system_timeout(SYSTEM_TIMER, timeout);
while (!is_system_timeout(SYSTEM_TIMER))
__asm__("NOP");
}
// result = 0 when timer timeout
unsigned char is_system_timeout(unsigned char timer_number)
{
unsigned char result;

result = 1;
if (timer_number 0)
result = 0;
}
return result;
}

// system function, decrease system_timer[i] to 0 timeout
void increment_system_time(void)
{
unsigned char i;

system_time++;
if (system_time == SYSTEM_TIME_MAX)
system_time = 0;

for (i = 0; i 0)
system_timer[i]–;
}
```

★ Like

Erich Styger
on **May 28, 2018 at 07:58** said:

Thanks for sharing!
I have implemented a similar way using the ‘Trigger’ Processor Expert component, which can use any kind of timer. That approach is different from the FreeRTOS approach as the FreeRTOS approach runs rather decoupled from the timer, and is only driven by the scheduler ticks. Your (or the trigger) approach work very well in a non-RTOS application.

★ Like

Naga
on **May 29, 2018 at 22:34** said:

Hi Eric, Great Blog and very educational articles. Apologize for asking something out of this topic, but i started to evaluate MCUXpresso 10.2. Is there a way to add FreeMASTER to the MCUXpresso project as we are missing ProcessorExpert tool in MCUxpresso 10.2.

Thanks You
Naga

★ Like

Erich Styger
on **May 30, 2018 at 07:59** said:

I have used FreeMaster for a long while: Segger J-Scope and now the new graphing function in MCUXpresso does it very well for me.
About Processor Expert: this is not officially supported, but you can install it, see https://mcuoneclipse.com/2017/04/09/mcuxpresso-ide-installing-processor-expert-into-eclipse-neon/.
Be aware of a Oxygen issue described here: https://mcuoneclipse.com/2017/10/02/how-to-fix-an-eclipse-workspace-that-does-not-open-any-more/ it happens if you have the component library view open.
I hope this helps,
Erich

★ Like

Pingback: New NXP MCUXpresso IDE v11.0 | MCU on Eclipse