Table of Contents

1. Zephyr timers and work queues 2
1.1. Timer use 2
1.2. Reading the status of the timer 3
1.3. Zephyr work queue 5
1.4. Summary of the tasks for the laboratory 6

1. Zephyr timers and work queues

A timer in Zephyr OS is a kernel object that measures the time using the internal system clock. When a timer reaches a specified time limit it can carry an action as specified in the application. In an operating system, the timer entity is software-based opposite to the hardware timer unit implemented in the microcontroller. Software timers use hardware timers to operate, and in general, the operating system keeps time-dependent actions. Zephyr OS allows one to define the number of timers which can be used to measure periods, asynchronously start some actions, control LED blinking, etc. When the timer is fired the system work queue task is started. The system work queue is optional and it's defined by a kernel if any application makes use of it. It is available to any application or kernel code that requires work queue support. We will present the work queues at the end of this exercise.

As with other tasks, the timer task should be as short as possible and non-blocking.

Timer object has some properties which define its behaviour.

- duration - specifies the duration for the first-time use
- period - specifies the duration of all other uses, the period set as K_FOREVER defines the one-time timer
- expiry function - the function which is called after every timer expiration
- status - the value that tells how many times the timer expired since the last read of the status

1.1. Timer use

The timer object must be defined and initialized. Its function is automatically added to the system work queue as the task, so we don't need to do anything with the queue. Let's try to modify the blinky code from the first laboratory on GPIOs to add a second LED blinking as the result of timer expiration. First, we need to add the timer object declaration.

```
        struct k_timer led_timer;
```

Next, we add the expiry function which will toggle the state of LED every call.

```
void led_timer_function(struct k_timer *timer_id)
    {
        gpio_pin_set(led_dev, LED1_PIN, (int)led1_is_on);
        led1_is_on = !led1_is_on;
    }
```

In the main, we have to add two functions: one to initialize the timer, and a second for starting it.

```
    k_timer_init(&led_timer, led_timer_function, NULL);
    k_timer_start(&led_timer, K_MSEC(200), K_MSEC(300));
```

Of course, there is a need to add and configure a second LED. This can be done as LED0 in the blinky example so we are not going to repeat the detailed description here.

1.2. Reading the status of the timer

We can read the status of the timer with the function

```
k_timer_status_get(timer_address);
```

Status is incremented whenever the timer expires, and it's cleared while it's read. We can use it to measure the time between some events. Let's try to measure how many times our led_timer for LED1 expires between LED0 toggles. Let's start with increasing the frequency of LED1 blinking to have higher numbers read by modifying the k_timer_start() function:

```
k_timer_start(&led_timer, K_NO_WAIT, K_MSEC(33));
```

To print the current status we can add the following line inside the infinite while loop in the main:

```
printk("Timer ticks %d",k_timer_status_get(&led_timer));
```

Don't forget to set the speed of transmission in platformio.ini:

```
monitor_speed = 115200
```

and enable serial port, and the printk() function in the prj.conf:

```
CONFIG_LOG=y
CONFIG_SERIAL=y
```

The resulting code can look as follows:

```
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr.h>
#include <device.h>
#include <devicetree.h>
#include <drivers/gpio.h>

/* 1000 msec = 1 sec */
#define SLEEP_TIME_MS   1000
#define MY_TIMER_PERIOD 33

/* The devicetree node identifier for the "led0" alias. */
#define LED0_NODE DT_ALIAS(led0)

#if DT_NODE_HAS_STATUS(LED0_NODE, okay)
#define LED0    DT_GPIO_LABEL(LED0_NODE, gpios)
#define PIN     DT_GPIO_PIN(LED0_NODE, gpios)
```

```
#define FLAGS   DT_GPIO_FLAGS(LED0_NODE, gpios)
#else
/* A build error here means your board isn't set up to blink an LED. */
#error "Unsupported board: led0 devicetree alias is not defined"
#define LED0    ""
#define PIN 0
#define FLAGS   0
#endif

#define LED1_NODE DT_ALIAS(led1)

#if DT_NODE_HAS_STATUS(LED1_NODE, okay)
#define LED1    DT_GPIO_LABEL(LED1_NODE, gpios)
#define PIN1    DT_GPIO_PIN(LED1_NODE, gpios)
#define FLAGS1  DT_GPIO_FLAGS(LED1_NODE, gpios)
#else
/* A build error here means your board isn't set up to blink an LED. */
#error "Unsupported board: led0 devicetree alias is not defined"
#define LED1    ""
#define PIN1    0
#define FLAGS1  0
#endif

const struct device *led_dev;
bool led1_is_on = true;
struct k_timer led_timer;

void led_timer_function(struct k_timer *timer_id)
{
   gpio_pin_set(led_dev, PIN1, (int)led1_is_on);
   led1_is_on = !led1_is_on;
}

void main(void)
{
   const struct device *dev;
   bool led_is_on = true;
   int ret;

   k_timer_init(&led_timer, led_timer_function, NULL);
   k_timer_start(&led_timer, K_NO_WAIT, K_MSEC(MY_TIMER_PERIOD));

   dev = device_get_binding(LED0);
   if (dev == NULL) {
    return;
   }

   led_dev = device_get_binding(LED1);
   if (led_dev == NULL) {
    return;
   }
```

```
   ret = gpio_pin_configure(dev, PIN, GPIO_OUTPUT_ACTIVE | FLAGS);
   if (ret < 0) {
    return;
   }

   ret = gpio_pin_configure(led_dev, PIN1, GPIO_OUTPUT_ACTIVE | FLAGS1);
   if (ret < 0) {
    return;
   }

   while (1) {
    gpio_pin_set(dev, PIN, (int)led_is_on);
    led_is_on = !led_is_on;
    k_msleep(SLEEP_TIME_MS);
    printk("Timer ticks %d",k_timer_status_get(&led_timer));
   }
}
```

1.3. Zephyr work queue

A work queue is an object that uses a dedicated thread to process work items. Work queue organization is FIFO-like so work items are processed in a first-in, first-out manner. Each work item is processed by calling the function assigned to it while initialization. Zephyr's work queue can help to synchronize the execution of time-consuming tasks triggered from time-constraint functions like ISRs, timer expiry functions, or from a high-priority thread to offload non-urgent processing to a lower-priority thread so it does not impact time-sensitive processing. If an ISR or a thread attempts to submit a work item that is already queued the work item is not affected. The work item remains in its current place in the FIFO queue, and the work is not doubled, so it will be performed only once. Although it is possible to create a user work queue, it is recommended to use a system work queue because every additional queue is resource-consuming. The system work queue is enabled while needed by the operating system, eg. if any timer is activated. There is no need to configure or initialize the system work queue. To be able to submit the work item to the system work queue the item must be defined as the structure of k_work type:

```
struct k_work my_work_item;
```

then the function for the work item must be defined:

```
void my_work_item_function(struct k_work *my_work_item){
    printk("Work item executed\n");
}
```

and in main.c the work item must be initialized:

```
k_work_init(&my_work_item, my_work_item_function);
```

An initialized work item can be submitted to the system work queue by calling k_work_submit()

```
k_work_submit(&my_work_item);
```

In this part, we will use it to execute simple actions triggered by the timer and the main loop. We will modify the example created with the use of a timer in the previous part of this laboratory.

1.4. Summary of the tasks for the laboratory

1. Modify the blink example according to the instructions presented in the first part of the instruction
2. Add to the timer expiry function the line that is executed longer than the timer period. This can be the line as follows:

```
for (int i=0; i<40; i++)
   {
      printk("counter = %d \n", i);
   }
```

3. Observe the results while adjusting the max iteration value, and explain observations in the report.
4. Add at the end of led_timer_function the line shown below and observe the behaviour of the program.

```
 k_msleep(1);
```

5. Define two work items with their functions. One for the code from the timer function, and the second for the code from the main function. Instead of executing the full code in the timer and main function submit corresponding work items. The code for the timer work item can look as follows (the code for the main function as well as the work items definition and initialization write on your own). Experiment with the maximal iteration value in the “for” loop. Describe observation results in the report.

```
void led_timer_function(struct k_timer *timer_id)
{
   gpio_pin_set(led_dev, PIN1, (int)led1_is_on);
   led1_is_on = !led1_is_on;
   k_work_submit(&timer_work_item);
}

void timer_work_item_function (struct k_work *timer_work_item)
{
   for (int i=0; i<40; i++)
   {
    printk("counter = %d \n", i);
    k_msleep(1);
   }
}
```

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:aih:lab:nrf52zephyrtimers?rev=1701900117

Last update: **2023/12/06 23:01**