Exercise 2

In this exercise we practice the usage of nRF's internal peripherals such as on-die temperature sensor as well as timer and *General Purpose I/O* (GPIO) interrupts. We initialize the peripherals and monitor the temperature using Keil's debug capabilities. We use a timer to read the temperature periodically every 5 seconds and a physical button to get the data right away. For each peripheral check the nRF522832 Product Specification.

Initializing the timer

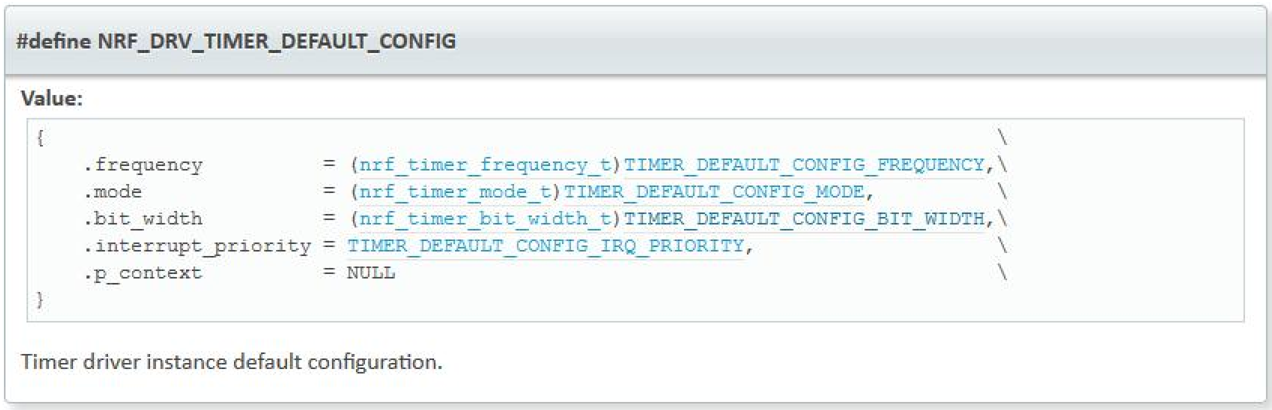
First we need to create a global timer instance.

```
const nrf_drv_timer_t TIMER_TEMP = NRF_DRV_TIMER_INSTANCE(0);
```

After this, we create a function for basic timer initializations. We give *nrf_drv_timer_init* a function pointer that points to a timeout handler which will be declared later.

```
/**
 * Function for initializing the timer.
 */
uint32_t init_timers()
{
    uint32_t time_ms = 5000; //Time(in milliseconds) between consecutive compare 
events.
    uint32_t time_ticks;
    uint32_t err_code = NRF_SUCCESS;

    //Configure TIMER_TEMP for measuring the temperature
    nrf_drv_timer_config_t timer_cfg = NRF_DRV_TIMER_DEFAULT_CONFIG;
    err_code = nrf_drv_timer_init(&TIMER_TEMP, &timer_cfg, temp_timeout_handler);
    APP_ERROR_CHECK(err_code);

    time_ticks = nrf_drv_timer_ms_to_ticks(&TIMER_TEMP, time_ms);

    nrf_drv_timer_extended_compare(&TIMER_TEMP, NRF_TIMER_CC_CHANNEL0, time_ticks, 
NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, true);

    nrf_drv_timer_enable(&TIMER_TEMP);

    return err_code;
}
```

Note that instead of writing each configuration explicitly, the drivers use preprocessor defined macros for simplicity and better readability. For example NRF_DRV_TIMER_DEFAULT_CONFIG is defined as follows in the API:



#define NRF_DRV_TIMER_DEFAULT_CONFIG

Value:

```
{                                                                               \
    .frequency          = (nrf_timer_frequency_t)TIMER_DEFAULT_CONFIG_FREQUENCY,\
    .mode               = (nrf_timer_mode_t)TIMER_DEFAULT_CONFIG_MODE,          \
    .bit_width          = (nrf_timer_bit_width_t)TIMER_DEFAULT_CONFIG_BIT_WIDTH,\
    .interrupt_priority = TIMER_DEFAULT_CONFIG_IRQ_PRIORITY,                    \
    .p_context          = NULL                                                  \
}
```

Timer driver instance default configuration.

Figure 1: An example of a macro definition

Configuring GPIO

In order to use a pin as an input, we need to use the GPIOTE module which functionality for GPIO pins. Each GPIOTE module can be assigned to a pin and is used to control its events. The button pin is defined as follows:

```
#define BUTTON_PIN      13
```

The actual function does all of the necessary routines to initialize the GPIOTE module. Note that also here *nrf_drv_gpiote_in_init* is provided with a function pointer to a button_handler function.

```
/**
 * Function for configuring General Purpose I/O.
 */
uint32_t config_gpio()
{
    uint32_t err_code = NRF_SUCCESS;
    if(!nrf_drv_gpiote_is_init())
    {
        err_code = nrf_drv_gpiote_init();
    }
    // Set which clock edge triggers the interrupt
    nrf_drv_gpiote_in_config_t config = GPIOTE_CONFIG_IN_SENSE_HITOLO(true);
    // Configure the internal pull up resistor
    config.pull = NRF_GPIO_PIN_PULLUP;

    // Configure the pin as input
    err_code = nrf_drv_gpiote_in_init(BUTTON_PIN, &config, button_handler);
    if (err_code != NRF_SUCCESS)
    {
        // handle error condition
    }
    // Enable events
    nrf_drv_gpiote_in_event_enable(BUTTON_PIN, true);

    return err_code;
}
```

Reading the temperature sensor

First we declare a global variable *temp*, which holds the temperature data.

```
int32_t volatile temp;
```

Then we create a function for reading the temperature from the TEMP peripheral. TEMP is started by triggering the START task. When the temperature measurement is completed, a DATARDY event will be generated and the result of the measurement can be read from the TEMP register.

```
/**
 * Function for reading on-die temperature
 */
void meas_temp()
{
    NRF_TEMP->TASKS_START = 1; // Start the temperature measurement

    // Busy wait while temperature measurement is not finished
    while (NRF_TEMP->EVENTS_DATARDY == 0)
    {
        // Do nothing.
    }
    NRF_TEMP->EVENTS_DATARDY = 0;
    // The value in the register is a multiple of 4
    temp = (nrf_temp_read() / 4);

    NRF_TEMP->TASKS_STOP = 1; // Stop the temperature measurement
}
```

Handler functions

In order to react to different events, we have to declare handlers for the timer and button interrupts. These simple functions initiate the temperature measurement routine declared above. Using a context pointer of type void* we can pass additional information to the timer handler if needed. In GPIO event handler we can determine which pin caused the interrupt by checking the *pin* variable.

```
/**
 * Handler for timer events.
 */
void temp_timeout_handler(nrf_timer_event_t event_type, void* p_context)
{
    meas_temp();
}

/**
 * Handler for GPIO events.
 */
void button_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t action)
{
    meas_temp();
}
```

Main function

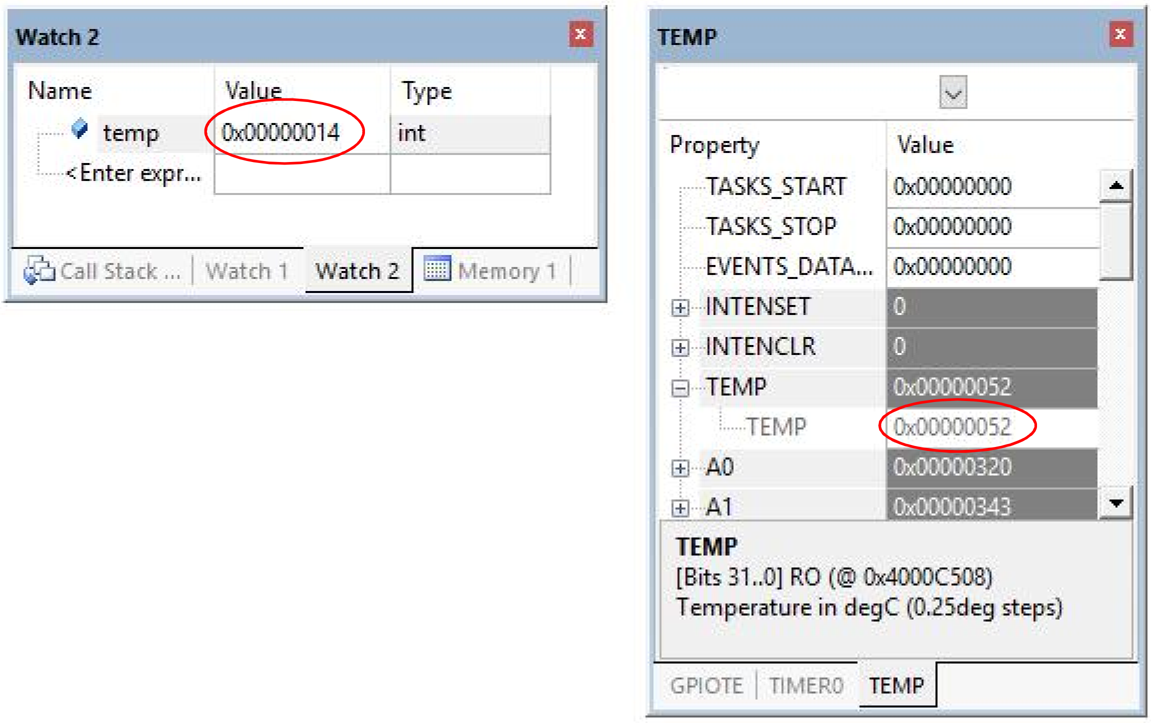
Next you have to execute the actual routines mentioned above in your main function. To follow a good programming style you should do error checks after each function call. This can be done simply by inserting the following if-check:

```
if (err_code != NRF_SUCCESS)
{
    // error check failed!
}
```

Additionally, you have to include the necessary header files in your *main.c*.

Debugging

Compile and flash the code, start the debugger and run the program. Note that compiler optimization level should be set to –O3 for better debugging accuracy. Periodical window update should also be checked from *View* tab. Open the TEMP peripheral view from *Peripherals -> System viewer -> TEMP.* Also right click the global *temp* variable we created and add it to a watch window.



Name	Value	Type
temp	0x00000014	int
<Enter expr...		

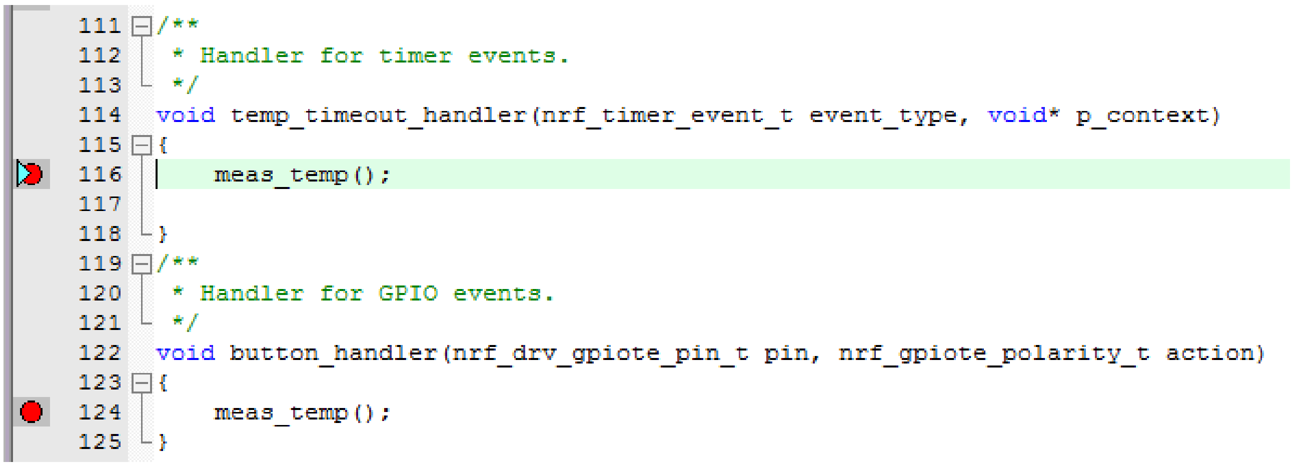
Property	Value
TASKS_START	0x00000000
TASKS_STOP	0x00000000
EVENTS_DATA...	0x00000000
INTENSET	0
INTENCLR	0
TEMP	0x00000052
TEMP	0x00000052
A0	0x00000320
A1	0x00000343

TEMP
[Bits 31..0] RO (@ 0x4000C508)
Temperature in degC (0.25deg steps)

Figure 2: Watch window and peripheral viewer displaying temperature data

Note that the peripheral register holds the temperature with a resolution of 0.25 Celsius degrees and has to be divided by 4 to get the absolute value. Try holding your finger on the microcontroller package and observe the value increase as the internal temperature rises. This value is by no means the correct outside temperature value but a silicon die temperature.

Next put breakpoints inside the two handler functions and run the code. Observe how the temperature timeout handler is called every time the timer expires and the GPIO handler is called when Button 1 is pressed.



```
/**
 * Handler for timer events.
 */
void temp_timeout_handler(nrf_timer_event_t event_type, void* p_context)
{
    meas_temp();

}
/**
 * Handler for GPIO events.
 */
void button_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t action)
{
    meas_temp();
}
```

Figure 3: Handler functions with breakpoints