EN 173 Lab Guides

Lab 11

Version 2026, 2025-03-04

Blinking LEDs with threads

This first program will simply flash LEDs and use threads with their default, most simple configuration. There are a total of three threads in this program, the unnamed one associated with the Zephyr RTOS (which starts `main()`) and two additional ones that we call `thread1` and `thread2`. The main thread runs by default but the other threads must be explicitly started, assigning them their own stack memory and what to do (associating a function with them). Each of these functions also have superloops so they run forever.

Exercise 11.1

1. Create a new application.
2. Type the code below into `main.c`.
3. Build and flash the application to your nRF52840 DK.

Demonstrate successful blinking of LEDs with threads.

Program 1. Basic blinky with threads

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>

void blink1(void); ①
void blink2(void);

#define LED0_NI DT_ALIAS(led0)
#define LED1_NI DT_ALIAS(led1)
#define LED2_NI DT_ALIAS(led2)
const struct gpio_dt_spec led0 = GPIO_DT_SPEC_GET(LED0_NI, gpios);
const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(LED1_NI, gpios);
const struct gpio_dt_spec led2 = GPIO_DT_SPEC_GET(LED2_NI, gpios);

#define STACKSIZE 1024
K_THREAD_DEFINE(thread1_id, STACKSIZE, blink1, NULL, NULL, NULL, 7, 0, 0); ②
K_THREAD_DEFINE(thread2_id, STACKSIZE, blink2, NULL, NULL, NULL, 7, 0, 0);

int main(void) {
    gpio_pin_configure_dt(&led0, GPIO_OUTPUT_ACTIVE);
    while (true) {
        gpio_pin_toggle_dt(&led0);
        k_msleep(500);
    }
}

void blink1(void) { ③
    gpio_pin_configure_dt(&led1, GPIO_OUTPUT_ACTIVE);
    while (true) {
        gpio_pin_toggle_dt(&led1);
```

```
        k_msleep(1100);
    }
}

void blink2(void) {
    gpio_pin_configure_dt(&led2, GPIO_OUTPUT_ACTIVE);
    while (true) {
        gpio_pin_toggle_dt(&led2);
        k_msleep(700);
    }
}
```

① The prototypes for the blink functions are declared at the top of the file. Theses functions will be associated with threads.

② The `K_THREAD_DEFINE` macro is used to define a thread. The first parameter is the name of the thread, the second is the stack size, the third is the function to run, the fourth through sixth parameters are arguments to be passed to the function. In this case these are set to `NULL` because the function run by this thread does not accept any arguments. The seventh parameter is the priority level (preemptible threads must have priorities between 0 and 9, with a larger number representing a *lower* priority). The eighth parameter is the options flag (which we will always set to 0). The final parameter is the time to delay starting this thread (in milliseconds).

③ The `blink1` function is defined. It configures the LED pin as an output and then toggles the pin every 1100 ms. Other threads may run while this thread is sleeping.

Threads with user parameters

The functions we use with threads can have up to three arguments. Because `K_THREAD_DEFINE` does not know the type of these arguments, the arguments have the generic type `void *`. We must cast them to the appropriate type inside of the function.

Exercise 11.2

1. Create a new application and configure logging in `prj.conf`.
2. Replace the code in `main.c` with the code below. Notice that there is now a single `blink` function for all of the LEDs. This example also illustrates that having a `main` function is not necessary.
3. Build and flash the application to your nRF52840 DK.

Program 2. Launching threads with user parameters

```
#include <zephyr/kernel.h>
#include <zephyr/logging/log.h>
#include <zephyr/drivers/gpio.h>

void blink(void* led, void* delay); ①

LOG_MODULE_REGISTER(Lab11b, LOG_LEVEL_DBG);
```

```
#define LED0_NI DT_ALIAS(led0)
#define LED1_NI DT_ALIAS(led1)
#define LED2_NI DT_ALIAS(led2)
const struct gpio_dt_spec led0 = GPIO_DT_SPEC_GET(LED0_NI, gpios);
const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(LED1_NI, gpios);
const struct gpio_dt_spec led2 = GPIO_DT_SPEC_GET(LED2_NI, gpios);

#define STACKSIZE 1024
K_THREAD_DEFINE(thread0_id, STACKSIZE, blink, (void*)&led0, (void*)300, NULL, 7,
0, 0); ②
K_THREAD_DEFINE(thread1_id, STACKSIZE, blink, (void*)&led1, (void*)1700, NULL, 7,
0, 0);
K_THREAD_DEFINE(thread2_id, STACKSIZE, blink, (void*)&led2, (void*)2300, NULL, 7,
0, 0);

void blink(void* led, void* delay) {
    int delay_ms = (int)delay; ③
    const struct gpio_dt_spec led_spec = *(const struct gpio_dt_spec*)led; ④
    LOG_INF("Blinking LED at P0.%d with delay %d ms", led_spec.pin, delay_ms);  ⑤
    gpio_pin_configure_dt(&led_spec, GPIO_OUTPUT_ACTIVE);
    while (true) {
        gpio_pin_toggle_dt(&led_spec);
        k_msleep(delay_ms);
    }
}
```

① The `blink` function is declared at the top of the file. It takes two arguments: a pointer to a `gpio_dt_spec` structure and a pointer to an integer. However, because `K_THREAD_DEFINE` does not know the type of these arguments, they are declared as `void *`.

② The `K_THREAD_DEFINE` macro is used to define three threads. Each thread runs the `blink` function with different arguments. The first user argument is a pointer to the `gpio_dt_spec` structure for the LED, and the second user argument is the delay in milliseconds. The third user argument is unused and must be set to `NULL`.

③ The delay is cast from a `void *` to an `int`.

④ The LED structure is cast from a `void *` to a `const struct gpio_dt_spec`. The memory address of the LED structure is passed to the thread, so we must dereference it to access the structure, thus the `*` in front of `(const struct gpio_dt_spec*)led`.

⑤ The LED pin number and delay are printed to the console.

Demonstrate successful blinking of LEDs with threads launched with user parameters.

Threads with custom structures as parameters

The functions we use with a thread can have only three argument but we can smuggle in multiple variables if we package them into a structure. In this example we use a structure to package a reference to an LED, an on-time for the blink, an off-time for the blink, the number of times it has flashed, and the maximum number of times to flash it.

A structure is declared with the keyword `struct`, followed by the name for this **type** of structure. Inside of curly braces you then define the variables held by this type of structure.

Suppose we wanted to keep track of the mass and radius for several spheres. We could create a structure for that with the following:

```
struct sphereData {
  float radius; // in cm
  float mass;   // in g
};
```

Later in your code you could declare several objects of this type and then set their properties.

```
sphereData redSphere, blueSphere;
redSphere.radius = 12.0;
redSphere.mass = 257.3;
blueSphere.radius = 7.0;
blueSphere.mass = 490.0;
```

If you pass a structure pointer (the memory location of a structure) to a function, then you might have to write code like `(*sphere).radius` to access its parts. This is a common situation so to make the notation a bit less messy there is a preferred alternative approach that accomplishes the same thing: `sphere→radius`.

Exercise 11.3

1. Create a new application and configure logging in `prj.conf`.
2. Replace the code in `main.c` with the code below.
3. Build and flash the application to your nRF52840 DK.

Demonstrate successful blinking of LEDs with threads passed a structure.

Program 3. Thread functions can have a structure as an argument.

```
#include <zephyr/kernel.h>
#include <zephyr/logging/log.h>
```

```c
#include <zephyr/drivers/gpio.h>
#include <string.h>

void blink(void* blink_param);

struct blinkParam { ①
    const struct gpio_dt_spec led;
    int on_time;
    int off_time;
    int max_count;
    volatile int count; ②
};

LOG_MODULE_REGISTER(Lab11c, LOG_LEVEL_DBG);

#define LED0_NI DT_ALIAS(led0)
#define LED1_NI DT_ALIAS(led1)
const struct gpio_dt_spec led0 = GPIO_DT_SPEC_GET(LED0_NI, gpios);
const struct gpio_dt_spec led1 = GPIO_DT_SPEC_GET(LED1_NI, gpios);

#define STACKSIZE 1024
struct blinkParam fastBlinker = {led0, 200, 100, 100, 0}; ③
struct blinkParam slowBlinker = {led1, 1500, 500, 30, 0};
K_THREAD_DEFINE(thread0_id, STACKSIZE, blink, (void*)&fastBlinker, NULL, NULL, 7, 0,
0);
K_THREAD_DEFINE(thread1_id, STACKSIZE, blink, (void*)&slowBlinker, NULL, NULL, 7, 0,
0);

int main(void) {
    while ( (fastBlinker.count < fastBlinker.max_count) || (slowBlinker.count <
slowBlinker.max_count) ) {
        k_msleep(5000);
        LOG_INF("%d (thread0), %d (thread1)", fastBlinker.count, slowBlinker.count);
    }
    LOG_INF("Goodbye! Both blinking threads are done.");
    return 0;
}

void blink(void* blink_param) {
    struct blinkParam* p = (struct blinkParam*)blink_param; ④
    const struct gpio_dt_spec led_spec = *(p->led); ⑤
    LOG_DBG("LED P0.%d has an on-time of %d ms and an off-time of %d ms", led_spec.
pin, p->on_time, p->off_time);
    gpio_pin_configure_dt(&led_spec, GPIO_OUTPUT_ACTIVE);
    while (p->count < p->max_count) {
        gpio_pin_set_dt(&(p->led), 1);
        k_msleep(p->on_time);
        gpio_pin_set_dt(&(p->led), 0);
        k_msleep(p->off_time);
        p->count++;
    }
```

```
}
```

① A structure is defined to hold the parameters for the blink function. It contains a pointer to the LED, the on-time, the off-time, the maximum number of times to blink, and the current count.

② The `count` variable is declared as `volatile` because it is modified by the thread and read by the main thread.

③ Two `blinkParam` structures are declared and initialized with the LED, on-time, off-time, maximum count, and count.

④ The `blink` function takes a `void*` argument and casts it to a `struct blinkParam*`.

Return to the Button Responder

It is all about threads

When a button is pushed, it will turn on a green LED. Four seconds later this LED will be turned off. To illustrate how threads interact, a red LED will be blinked off-and-on in a separate thread.

Exercise 11.4

1. Connect a red LED and 330 Ω resistor in series between P0.28 and ground.
2. Connect a green LED and 330 Ω resistor in series between P0.29 and ground.
3. Connect a push button between P0.03 and the power bus.
4. Create an application, generate a build configuration, and then create an appropriate overlay.
5. Type the code of Program 4 into `main.c`.
6. Build and flash the application to your nRF52840 DK.
7. You might think there is a problem because there is no heartbeat. While it is true that this may not be what **you** expected, all might be well. Push the button and see what happens. You should see some LED action now.
8. Explain why the microcontroller is behaving the way it is.
9. Using the *Analog Discovery 2* measure the latency (as you did in previous labs).
10. Change priority of the heartbeat thread to 6 (from 8) and flash the new program onto the microcontroller.
11. Explain the new behavior.
12. Measure the latency in this modified configuration.

Program 4. Button responder using threads

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>
```

```
void buttonHandler(void);
K_THREAD_DEFINE(buttonThread_id, 1024, buttonHandler, NULL, NULL, NULL, 7, 0, 0); ①

void heartbeatHandler(void);
K_THREAD_DEFINE(heartBeatThread_id, 1024, heartbeatHandler, NULL, NULL, NULL, 8, 0,
0); ②

#define BUTTON_NI DT_NODELABEL(pb)
const struct gpio_dt_spec button = GPIO_DT_SPEC_GET(BUTTON_NI, gpios);

#define REDLED_NI DT_NODELABEL(redled)
#define GREENLED_NI DT_NODELABEL(greenled)
const struct gpio_dt_spec heartbeat = GPIO_DT_SPEC_GET(REDLED_NI, gpios);
const struct gpio_dt_spec buttonAlert = GPIO_DT_SPEC_GET(GREENLED_NI, gpios);

void heartbeatHandler(void) {
    gpio_pin_configure_dt(&heartbeat, GPIO_OUTPUT_ACTIVE);
    while (true) {
        k_msleep(250);
        gpio_pin_toggle_dt(&heartbeat);
    }
}

void buttonHandler(void) {
    gpio_pin_configure_dt(&button, GPIO_INPUT);
    gpio_pin_configure_dt(&buttonAlert, GPIO_OUTPUT_INACTIVE);
    int prevState = gpio_pin_get_dt(&button);
    int currState;

    while (true) {
        currState = gpio_pin_get_dt(&button);
        if (currState && !prevState) {
            gpio_pin_set_dt(&buttonAlert,1);
            k_msleep(3000);
            gpio_pin_set_dt(&buttonAlert,0);
        }
        prevState = currState;
    }
}
```

① The button handler thread is defined with a priority of 7.

② The heartbeat handler thread is defined with a lower priority (8).

Threads plus interrupts

You have explored various versions of the button responder. In a previous day we used an all-interrupt approach: using an event-based interrupt to respond to the button push and then an time-based interrupt to turn off the LED. The button push might be time-critical event but turning off the LED is probably not and it should not interrupt other things. A hybrid approach, combining

interrupts and threads may be best.

The interrupt will communicate with a thread by posting an **event**. The thread waits for an event to be set and then responds when it is (and also clears the event).

Program 5. Button responder using interrupt plus event-monitoring thread

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>


void heartbeatHandler(void);
K_THREAD_DEFINE(heartBeatThread_id, 1024, heartbeatHandler, NULL, NULL, NULL, 8, 0,
0);

void alertHandler(void);
K_THREAD_DEFINE(alertThread_id, 1024, alertHandler, NULL, NULL, NULL, 7, 0, 0);

#define BTN_NI DT_NODELABEL(pb)
const struct gpio_dt_spec btn = GPIO_DT_SPEC_GET(BTN_NI, gpios);
static struct gpio_callback btn_cb_data;
void buttonISR(const struct device *dev, struct gpio_callback *cb, uint32_t pins);

#define REDLED_NI DT_NODELABEL(redled)
#define GREENLED_NI DT_NODELABEL(greenled)
const struct gpio_dt_spec heartbeat = GPIO_DT_SPEC_GET(REDLED_NI, gpios);
const struct gpio_dt_spec alert = GPIO_DT_SPEC_GET(GREENLED_NI, gpios);

#define BTN_EVENT BIT(0) ①
K_EVENT_DEFINE(eventManager);

int main(void) {
    gpio_pin_configure_dt(&alert, GPIO_OUTPUT_INACTIVE);
    gpio_pin_configure_dt(&btn, GPIO_INPUT);
    gpio_init_callback(&btn_cb_data, buttonISR, BIT(btn.pin));
    gpio_add_callback(btn.port, &btn_cb_data);
    gpio_pin_interrupt_configure_dt(&btn, GPIO_INT_EDGE_TO_ACTIVE);
}

void buttonISR(const struct device *dev, struct gpio_callback *cb, uint32_t pins) {
    gpio_pin_set_dt(&alert, 1);
    k_event_post(&eventManager, BTN_EVENT);
}

void heartbeatHandler(void) {
    gpio_pin_configure_dt(&heartbeat, GPIO_OUTPUT_ACTIVE);
    while (true) {
        k_msleep(250);
        gpio_pin_toggle_dt(&heartbeat);
    }
}
```

```
void alertHandler(void) {
    while (true) {
        k_event_wait(&eventManager, BTN_EVENT, false, K_FOREVER);
        k_event_clear(&eventManager, BTN_EVENT);
        k_msleep(3000);
        gpio_pin_set_dt(&alert, 0);
    }
}
```

① Although events correspond to bits, it is more readable to label them in a human-friendly way.

Exercise 11.5

1. Create an application based on Program 5 and upload it to your nRF52840 DK.
2. Using the *Analog Discovery 2* measure the latency.

Your Turn

In this assignment you will write a program that incorporates the following features:

- a GPIO-triggered interrupt
- an event-monitoring thread

Although there are ways to complete the assigned task without those features, the point of this assignment is to practice using them (so do so if you want full credit).

Assignment 11.1

You will create a program that blinks an LED (with a period of 1 s) while it waits for an interrupt triggered by a single tap (as detected by our accelerometer). The LED is then held on while twenty temperature data points are gathered, spaced 500 ms apart, with a temperature sensor (either the analog TMP36 or the digital TMP102, your choice). Upon completion the average temperature (in either Celsius or Fahrenheit, your choice) is logged to the console, the accelerometer is reset to wait for the next tap, and LED blinking resumes.

The communication with the accelerometer cannot take place in the interrupt service routine. It must be done in a separate thread because SPI and I^2C communication are not allowed in an ISR. You get to decide which protocol to use.

The assignment link is available on Blackboard.

When your program and circuit are working, create a video demonstrating this. The video should show the LED blinking, a tap on the accelerometer, the LED staying on as temperature data is gathered, the average temperature displayed, and the LED blinking again. The video should be uploaded to

Blackboard.