EN 173 Lab Guides

Lab 12

Version 2025, 2025-01-20

Protecting access to a resource with a mutex

If you have more than one thread that wants access to something but proper operation will only occur if that access is exclusive, it can be protected with a mutex. Whichever thread has control of the mutex is allowed to access that resource. In this example we have two threads that want to flash different patterns on the internal LEDs. You will explore how the behavior depends on the locking and unlocking of the mutex.

Program 1. Protecting access to LEDs with a mutex

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

void blinkHandler(void);
K_THREAD_DEFINE(blinkThread_id, 1024, blinkHandler, NULL, NULL, NULL, 7, 0, 100); ①

void rotateHandler(void);
K_THREAD_DEFINE(rotateThread_id, 1024, rotateHandler, NULL, NULL, NULL, 7, 0, 10000);
②

#define FIRST_PIN 13
#define NUM_LEDS 4
#define PORT0_NI DT_NODELABEL(gpio0)
const struct device *port = DEVICE_DT_GET(PORT0_NI);
gpio_port_value_t led_mask = ((1<<NUM_LEDS) - 1) << FIRST_PIN; ③
K_MUTEX_DEFINE(ledMutex); ④

int main(void) {
    if (device_is_ready(port)) {
        for (int pin = FIRST_PIN; pin < FIRST_PIN+NUM_LEDS; pin++) {
            gpio_pin_configure(port, pin, GPIO_OUTPUT_INACTIVE | GPIO_ACTIVE_LOW);
        }
    } else return -1;
    return 0;
}

void blinkHandler(void) {
    while (true) {
        k_mutex_lock(&ledMutex, K_FOREVER); ⑤
        for (int i = 0; i < 10; i++) {
            gpio_port_toggle_bits(port, led_mask);
            k_msleep(500);
        }
        k_mutex_unlock(&ledMutex); ⑥
        k_msleep(5000);
    }
}
```

```
void rotateHandler(void) {
    while (true) {
        k_mutex_lock(&ledMutex, K_FOREVER);
        for (int pin = FIRST_PIN; pin < FIRST_PIN + NUM_LEDS; pin++) {
            gpio_port_set_masked(port, led_mask, BIT(pin));
            k_msleep(200);
        }
        gpio_port_set_masked(port, led_mask, 0);
        k_mutex_unlock(&ledMutex);
    }
}
```

① Thread that will blink all of the LEDs together is started after a delay of 100 ms. This gives the main thread time to set up the GPIO pins.

② Thread that will rotate through the LEDs is started after a delay of 10 seconds.

③ Define the mask for the LEDs using bit-shifting.

④ Define a mutex to protect access to the LEDs.

⑤ Lock the mutex so that only this thread can access the LEDs.

⑥ Unlock the mutex so that other threads can access the LEDs.

Exercise 12.1

1. Create an application with Program 1.
2. While your program is building, predict what you will observe when it runs.
3. Upload and run the code. Compare the observed behavior to your predicted behavior.
4. Comment out the `k_mutex_lock` and `k_mutex_unlock` throughout the program.
5. While your program is compiling, predict if the behavior will change and if so, how.
6. Upload and run the code. Compare the observed behavior to your predicted behavior.

Message in a queue

Message queues are one way to pass data between threads (and ISRs). A pool of memory is set aside for the messages. Before putting a new message into the pool, the program first checks to see if a spot is available. Similarly, after a message has been read, the spot is cleared.

In this example one thread will periodically store temperatures from a TMP36 as well as the time they were recorded (in milliseconds since power-up).

Program 2. Use a message queue to collect and then print data from a TMP36 sensor

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

```
struct temp_data_t { ①
    float temperature; // in Celsius
    uint32_t timestamp; // in milliseconds
};
K_MSGQ_DEFINE(temp_msgq, sizeof(struct temp_data_t), 5, 1); ②

void recordData(void);
K_THREAD_DEFINE(recorderThread_id, 1024, recordData, NULL, NULL, NULL, 7, 0, 0);

#define TMP36 DT_NODELABEL(temp0)
#define MV_AT_25C DT_PROP(TMP36, vtemp25)
#define MV_PER_DEG_C DT_PROP(TMP36, sensor_slope)
#define NUM_ADC_READINGS 10
const struct adc_dt_spec tmp36 = ADC_DT_SPEC_GET(TMP36);

LOG_MODULE_REGISTER(Lab12b, LOG_LEVEL_DBG);

int main(void) {
    struct temp_data_t data;
    while (true) {
        while (k_msgq_get(&temp_msgq, &data, K_NO_WAIT) == 0) { ③
            LOG_INF("Temperature: %.1f C, Timestamp: %u", data.temperature, data
.timestamp);
        }
        k_msleep(10000);
    }
}

void recordData(void) {
    struct temp_data_t data;
    int err;
    uint16_t buf;
    int val_mV;
    float T_in_C;
    struct adc_sequence tmp_reading = {
        .buffer = &buf,
        .buffer_size = sizeof(buf)
    };
    adc_channel_setup_dt(&tmp36);
    adc_sequence_init_dt(&tmp36, &tmp_reading);

    while (true) {
        adc_read_dt(&tmp36, &tmp_reading);
        val_mV = (int)buf;
        adc_raw_to_millivolts_dt(&tmp36, &val_mV);
        T_in_C = 25.0 + ((float)val_mV - (float)MV_AT_25C)/(float)MV_PER_DEG_C;
        data.temperature = T_in_C; ④
        data.timestamp = k_uptime_get_32(); ⑤
        k_msgq_put(&temp_msgq, &data, K_FOREVER); ⑥
        k_msleep(1000);
    }
}
```

① Define the format of a message (typically using a structure).

② Set up a message queue with slots for 5 messages. The final argument specifies alignment of the messages slot addresses. A value of 1 means that the addresses will be aligned to the size of the message.

③ Try to get a message immediately (`K_NO_WAIT`). If there is one (a return value of 0), print it out. If not, wait 10 seconds before trying again.

④ Store the temperature in the data structure

⑤ Get the current time in milliseconds since power-up and store it.

⑥ Request an open slot in the message queue. In this case we are willing to wait forever, but you could set it to give up and do something else after a while. You can also use a message queue in an ISR, but in that case you would need to use `K_NO_WAIT` as the timeout.

Exercise 12.2

1. Create a new application, generate a build configuration, and create an overlay.
2. Add the following to the overlay:

```
&adc {
  #address-cells = <1>;
  #size-cells = <0>;
  status = "okay";
  channel@0 {
    reg = <0>;
    zephyr,gain = "ADC_GAIN_1_2";
    zephyr,reference = "ADC_REF_INTERNAL";
    zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
    zephyr,input-positive = <NRF_SAADC_AIN1>;
    zephyr,resolution = <12>;
  };
};

/{
  temp0: temp0 {
    compatible = "adi,tmp36";
    io-channels = <&adc 0>;
  };
};
```

3. Create a new folder named `dts` at the top-level of your application (not inside any folder other than the one holding application itself). Inside of the `dts` folder create another folder `bindings`.
4. Create a file named `adi,tmp36.yaml` inside of the `bindings` folder.
5. Add the following to the `adi,tmp36.yaml` file:

```yaml
description: Analog Devices analog temperature sensor TMP36
compatible: "adi,tmp36"
include: sensor-device.yaml
properties:
  io-channels:
    required: true
    description: ADC channel for temperature sensor
  vtemp25:
    type: int
    default: 750
    description: |
      Temperature sensor voltage at 25 degrees Celsius
      in millivolts
  sensor-slope:
    type: int
    default: 10
    description: |
      Temperature sensor slope in millivolts per degree Celsius
```

6. Create a `main.c` containing Program 2.
7. While your program is compiling, predict what you will observe when it runs.
8. Upload and run the code. Compare the observed behavior to your predicted behavior. Watch for at least one minute.
9. Delete the `k_msleep(10000);` inside of `main`.
10. While your program is compiling, predict what you will observe when it runs.
11. Upload and run the code. Compare the observed behavior to your predicted behavior. Watch for at least one minute.

Your Turn

You will create a program that measures temperature with the digital TMP102. The temperature (in Celsius) will be measured every time a true single tap (not a tap part of a double tap) is detected by the ADXL345 accelerometer is detected and the temperature will be put in a message queue. At the same time, a log message should display the current number of messages stored. The queue will have 20 slots for messages. If the queue gets full, you will want to free the oldest slot and replace it with the new measurement.

You may find it helpful to consult the Message Queue API documentation.

When a double tap is detected, the console will be updated to show the average of the stored temperature data (including units). Your program needs to pay attention to the actual number of data points being averaged (it will be between 1 and 20). Your program should also gracefully handle the case of no data by displaying “No data” instead. Displaying the average should clear the message queue.

A button will be used to select the displayed temperature units (rotating through the options `C`, `K`, `F`, and `R`). After being selected the console should display that choice with a message of the form “Units set to C”.

Assignment 12.1

Your program must also incorporate the following RTOS features:

- a mutex to control access to the I2C bus
- appropriate use of threads
- appropriate use of a message queue