

Lab 2 - RTOS Basics

Welcome to lab 2! In this lab we will start using Zephyr RTOS to get a feeling for programming with a real-time operating system.

Requirements

For this lab, you will need a working Zephyr installation. If you are using your own computer you can find the installation instructions [here](#).

Be aware that Zephyr does not permit spaces in the installation path

For part 3 you will also need a terminal program, such as minicom, screen, or Putty.

You cannot use Pico SDK for this lab.

We softly advise against using the VSCode Raspberry Pico extension. The extension has only just come out and seems preliminary. Hence, our ability to assist you with it is limited.

Template

You can download the template for this lab from Studium! There is a subfolder for each part.

Resources

Here are resources you might find helpful during this lab:

- [Kernel services](#): tasks, mutexes, semaphores, queues, etc.
- [Devicetree HOWTOs](#)
- [Raspberry Pi Pico 2 Board Support](#): build instructions, debugging, etc.
- [debugprobe UF2 file](#): for debugging, UART-USB bridge
- [Zephyr demo applications](#)
- [Zephyr peripheral APIs](#): drivers for GPIO, I2C, UART, etc.
- [West documentation](#): West is the command line tool used to build Zephyr applications
- [Zephyr Application Development Guide](#): Describes folder layout for new applications and detailed build instructions
- [Zephyr video tutorial by The Pull-up Resistor Channel](#): Excellent video playlist which covers Zephyr and helps demystify device trees!

You might also want to check the example applications we covered in class ([blinky](#) and [deferred interrupt processing](#)). You are free to copy/use the applications' code in your assignment.

Building an application

To build an application you use the **west** command line tool. A link to its documentation is given in the section above. In general, to build an application you have to navigate to the folder that holds the application's **CMakeLists.txt** file and issue

```
west build -b rpi_pico2/rp2350a/m33
```

This will automatically create a `/build` folder in the current directory and start compiling your application. West has many optional flags (such as the `-b pi_pico2/rp2350a/m33`) so it is worth looking into the documentation which ones might be interesting for you!

If west fails to build your project with an error about a missing “build” command you probably have not set all required environment variables. This can usually be solved by sourcing the *zephyr-env.sh* (Linux, macOS) or *zephyr-env.cmd* (Windows) file in the `/zephyrproject/zephyr/` folder.

You can program your board as you did in lab 1 by copying the resulting `.uf2` file (located at `../build/zephyr/zephyr.uf2`) onto the USB device that shows up when you start your board in boot mode.

You can also use **west flash** by connecting the picoprobe debugger and building your project as outlined in the [Raspberry Pi Pico 2 Board Support](#) page. Notice that you can set build variables (such as `-DOPENOCD`) permanently in your application's `CMakeLists.txt` file (outlined [here](#)).

Hint: Before getting started on the tasks we **strongly advise** that you first build/run the lecture examples, to ensure everything is set up correctly (setting up Zephyr can be difficult). Then you can proceed with the assignment.

Part 1 - Extended Blinky

The goal of this part is to get 4 LEDs to blink with different frequencies, i.e. with an ON/OFF delay of 100, 200, 300, and 500ms respectively. You should define a different task for each of the LEDs.

You can freely choose which LEDs you want to use. In order to use them with Zephyr you will have to adapt the devicetree for your application. You can have a look at the example blinky application which we did in the lecture and read the section about [devicetree overlays](#) in the Zephyr documentation.

Hint: You can use the same **entry function** for all tasks, giving the LED and the delay as input arguments.

Part 2 - Interrupts, Semaphores, Mutexes

In this part of the assignment you shall create a program that always blinks one of 4 LEDs. To switch between different LEDs, use the first button (GP20). You don't have to care about the state of the other LEDs that are currently not blinking (e.g. they can be ON or OFF).

It is up to you to decide which LEDs you want to use. You will also have to include the button in the devicetree for this part!

You must use both a semaphore and a mutex in this part of the assignment

Hints:

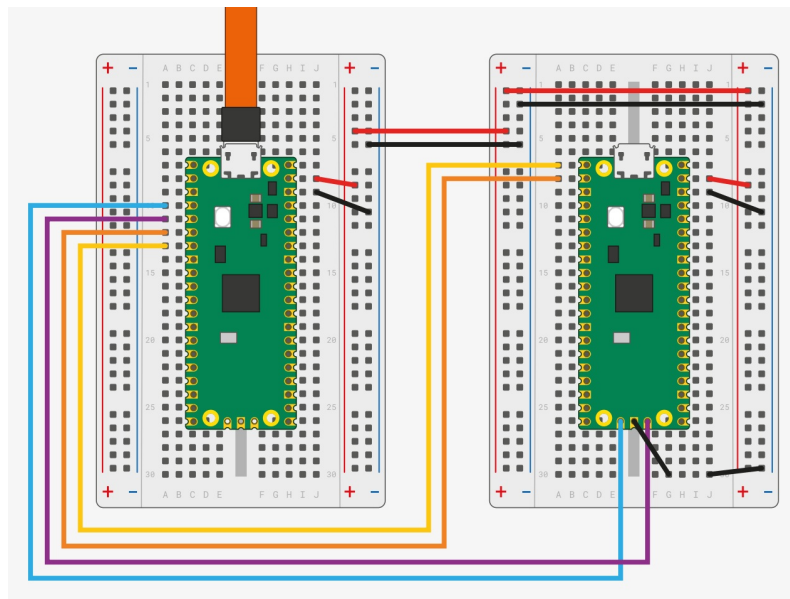
- There should be two tasks in this system, one handling the blinking, and one handling the button presses
- Use deferred interrupt handling. The button interrupt routine should only give a signal to the button handler task indicating that the button was pressed
- Pay attention to guard any access to shared variables

Part 3 - Three second game

In this last part of the assignment you shall create a time estimation game. The user shall be prompted (via the terminal), to push the first button. The idea is to push the same button again after exactly 3 seconds. The program shall give the user feedback on how close they were to 3 seconds. There are many ways of how to implement this. Be creative (see next page)!

Hint:

To read the output from your `printf` statements you will need to use a UART-USB bridge. `debugprobe` implements such a bridge along with debugging support. You can deploy the debugprobe on the second Pico board, which you then connect to the main board according to the diagram shown in Appendix A of the [Getting started guide](#), diagram which is also included here for convenience. With this connection, your print-outs are relayed to your computer, indirectly via the second Pico board. You can then view these print-outs in a console (see section “Use the UART” of the same document).



Wiring between Pico A (left) and Pico B (right), configuring Pico A as a debugger/UART-USB bridge.

If you use the debugprobe as a UART-USB bridge, you have to connect the UART TX and RX signal lines (orange and yellow wire) and ground. If you don't want to debug/program with the debugprobe you can leave away the blue, purple, and red wires. If you want to program your device without debugprobe you will want to use two USB cables (one for each Pico) in this setup!

The Kconfig file in the template (`prj.conf`) enables `Newlib`, a complete C library implementation. This gives you access to full-fledged implementations of statements such as `printf`. If you use such statements, don't forget to flush the output buffer after printing (e.g., using `fflush(stdout)`) otherwise you might not see any output! Alternatively, you can use `printfk`, which does not require flushing but supports only basic format specifiers.