

Lesson 2: Introduction

- ☒ 2.1 [RTOS basics](#)
- ☒ 2.2 [Zephyr structure](#)
- ☒ 2.3 [Tutorial structure](#)

Why do I need an RTOS?

An RTOS is rarely a requirement; however, as you start to increase the functionality of your embedded applications, it becomes increasingly harder to do everything within one single main loop and some interrupt routines. Usually the next level of complexity is some kind of state machine, where the output of your electronic device changes depending on this (internal) state. This however only gets you so far. For example, what if you need to be able to operate multiple complex inputs and outputs simultaneously? A good example could be an TCP/IP connection, over which you'll be receiving some kind of data which then has to be used to operate a robotic arm, control an electric motor, send out a signal... It quickly becomes clear that a new level of abstraction is required to not drown in the complexity that would be required to implement something like that. This is where an RTOS steps in.

What makes Zephyr different from other RTOS's?

As I haven't had the time to study other RTOS'es as in-depth I might be biased here. However from my point of view, studying Zephyr has the following benefits:

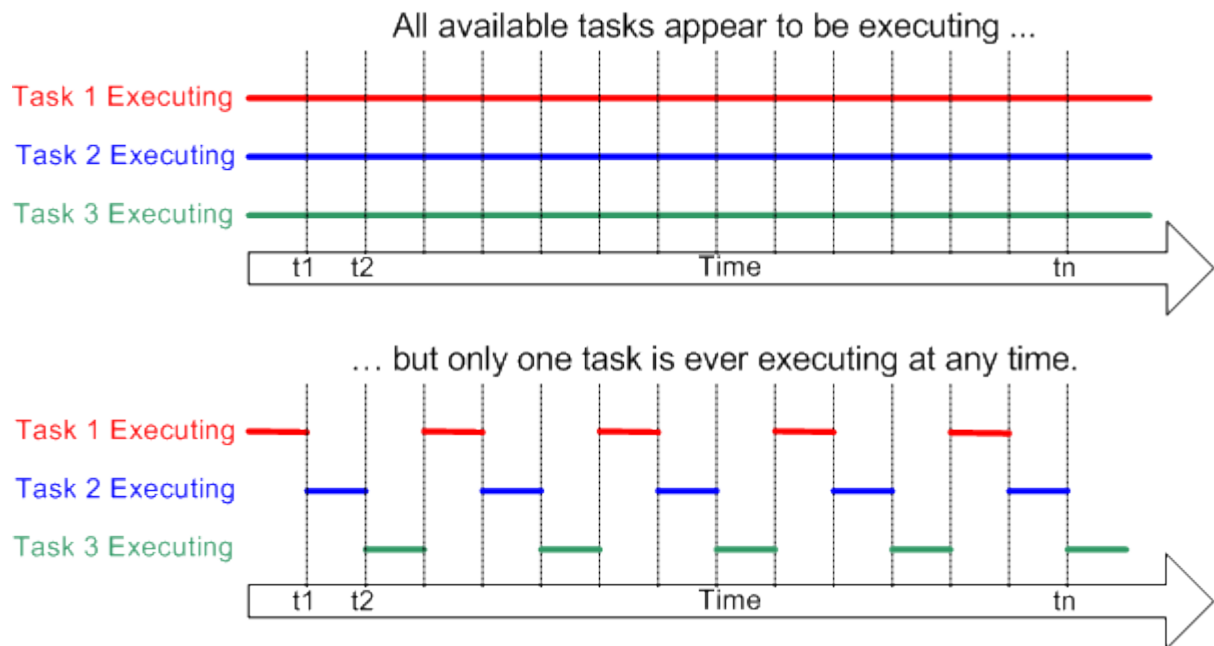
- Zephyr is supported by the Linux Foundation
- By learning Zephyr you'll automatically get a taste for the Linux kernel. Both show some overlap in the way they are implemented, for example: Kconfig and devicetrees are concepts borrowed by Zephyr from Linux.
- Zephyr is flexible: you can trade off footprint versus extra functionality (through Kconfig). If you don't know how this works: don't worry, I'll explain this later.
- Zephyr supports a wide variety of different dev boards/SoCs.
- A more extensive lists of reasons can be found [here](#)

How does an RTOS work and what are some key concepts?

Ok, so now that I've hopefully convinced you of the *use* of an RTOS, let's start by taking a look at *how* this all works. The first and most fundamental part of an RTOS is the **kernel**. The kernel is responsible for scheduling CPU time for each particular task so that they *appear* to be happening simultaneously. The particulars of how this scheduling algorithm (or also called: **scheduler**) works is not important for now.

Each **thread** (or task) will use registers and memory as it executes. The whole of these processor registers and stack (memory) compromise the **context** of that particular thread. Once the RTOS decides to switch the thread and run something else, it will need to first *store* context away and then *load* in the context for the thread it wants to run next. This process is called **context switching**.

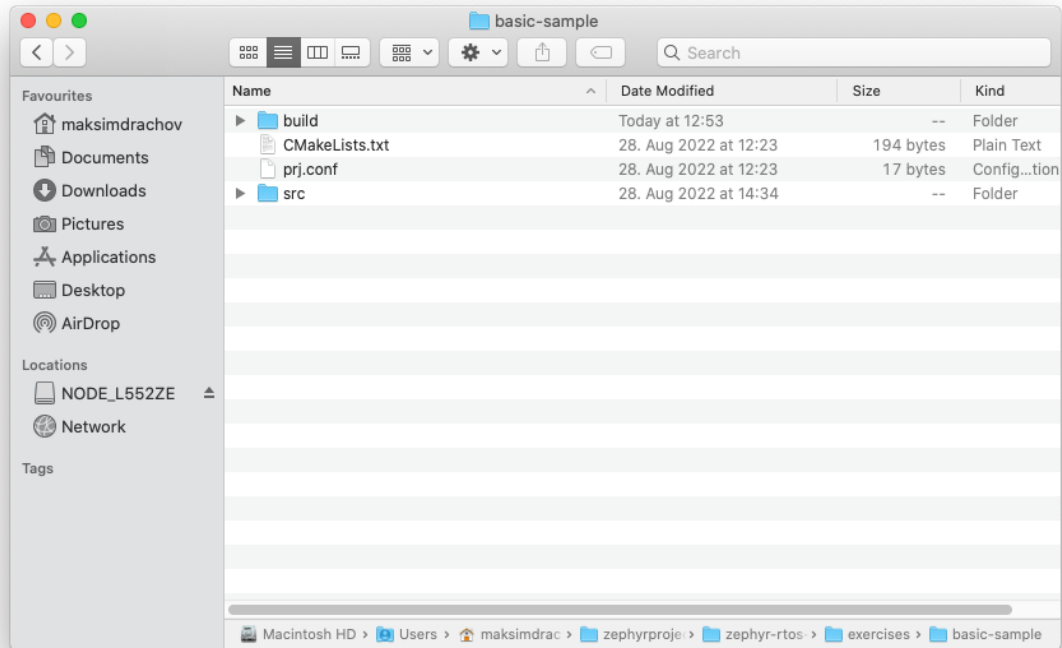
Along with threads, you'll be using primitives such as **queues**, **mutexes** and **semaphores** for inter-thread communication. Then each RTOS provides varying levels of support for different protocols such as **TCP/IP**, **Bluetooth**, **LoRaWan**,... This makes your life easier, since now you don't need to study these protocols as in-depth. You'll get a series of API calls which should increase speed of development.



Zephyr structure

How to work with Zephyr?

Before we start writing our first applications in Zephyr, it might be a good step to take a look at the folder structure that Zephyr provides us.



Let's go one-by-one:

build

This folder appears only once you have build your application and contains the files that are used to flash your microcontroller.

The following files are sometimes interesting to take a look at:

- **build/zephyr/zephyr.dts:** CMake uses a devicetree to tailor the build towards your specific architecture/board. This is the final version of that file. Here you'll be able to find all the different functionality (GPIO, Timers, PWM, DMA, UART, SPI, I2C, DAC, USB,...) that is present on your MCU (that can then be called upon in your application).
- **build/zephyr/.config:** The final Kconfig used for your built. This can be useful to verify if a setting has been set correctly.

CMakeLists.txt

This file will be used by [CMake](#) to set up your build, during this tutorial you won't need to change this one.

prj.conf

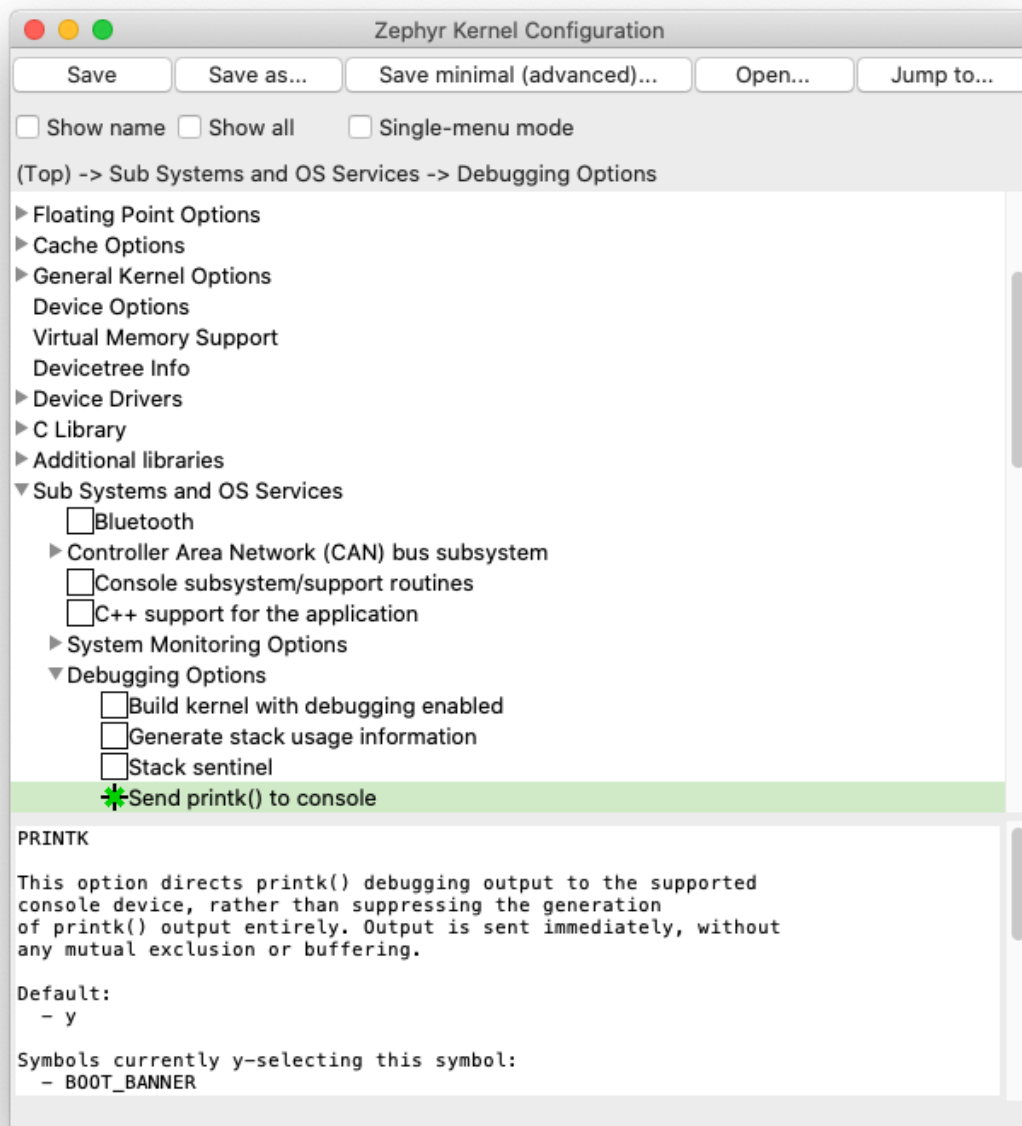
This is your Kconfig file. **Important!** This file will contain any *additional* settings you want set for your particular Zephyr build. Depending on your particular application, you might want to (for example) include a TCP/IP stack or make some changes to the scheduler. We'll explore some of these options throughout the tutorial.



Right now it's pretty empty, since for basic-sample we don't require any "fancy" functionality, just the basic Zephyr kernel. The only additional functionality we set is CONFIG_PRINTK, which allows us to use the printk function to output to the serial port (which is then displayed on your computer screen using Coolterm).

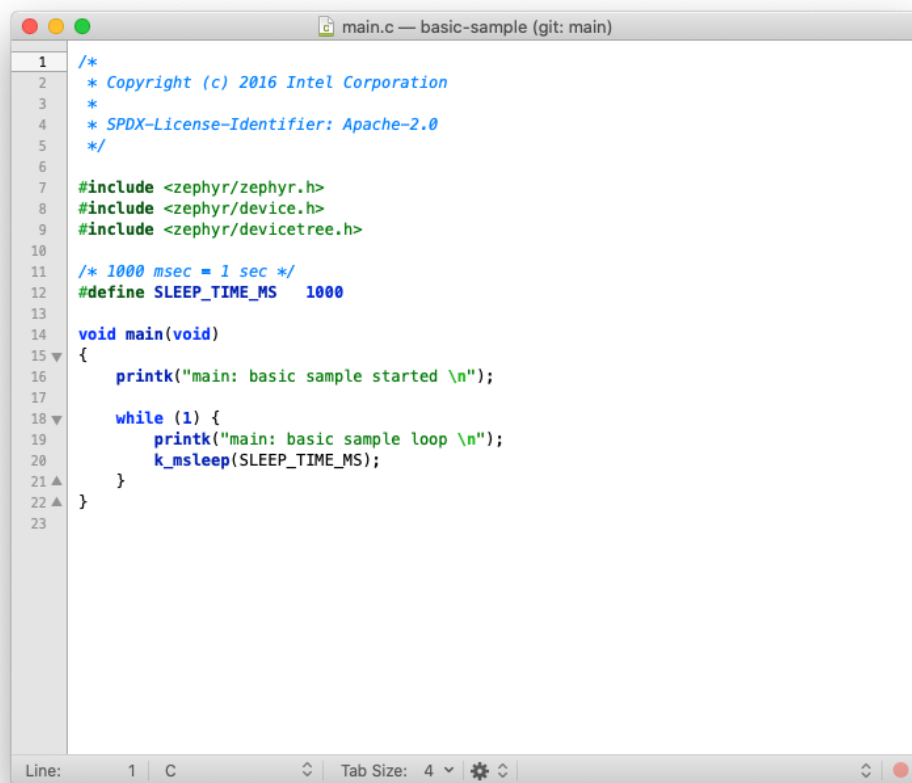
If you're ever unsure about what a particular config setting does, you have 2 options:

- Use [google](#): usually Zephyr Documentation is one of the first links
- Use the guiconfig: in your basic-sample folder execute `west build -t guiconfig`. This will show you a menu of all the possible configuration settings and a small description of what each one does. (Use Jump to to find a particular config)



src

Where the magic happens! This folder should contain all of your custom application code. For now it contains one file: `main.c`, which prints out a message to the serial port and then sleeps for 1 second.



```
1  /*
2   * Copyright (c) 2016 Intel Corporation
3   *
4   * SPDX-License-Identifier: Apache-2.0
5   */
6
7  #include <zephyr/zephyr.h>
8  #include <zephyr/device.h>
9  #include <zephyr/devicetree.h>
10
11  /* 1000 msec = 1 sec */
12  #define SLEEP_TIME_MS 1000
13
14  void main(void)
15  {
16      printk("main: basic sample started \n");
17
18      while (1) {
19          printk("main: basic sample loop \n");
20          k_msleep(SLEEP_TIME_MS);
21      }
22  }
23
```

Tutorial structure

How is the tutorial structured?

Each lesson covers 1 aspect of Zephyr. The order of lessons is chosen deliberately, unless you already know a thing or two, it is recommended to not skip ahead.

Each lesson consist of 4 parts:

- **Introduction:** introduce each new topic/concept, explain why it is useful.
- **Commands:** go over all the commands/api calls that are relevant to that particular topic.
- **Kconfig:** here we take a look at the different Kconfig settings that might be interesting.
- **Examples:** some examples to show how to apply the discussed concepts.