# Johanne Vigny , April 2025

# Zephyr RTOS

## Setup

### Prerequisites

#### Install

* Go to `C:\` to install zephyr in the directory `zephyrproject`
* Follow the [Getting Started Guide](https://docs.zephyrproject.org/latest/develop/getting_started/index.html) to install the Zephyr source code.
* Install `openocd` if needed or `[jlink](https://www.segger.com/downloads/jlink)` depending on the runner needed by the board:

> choco install openocd

⚠️ For Jlink don’t forget to install the USB driver, otherwise it will not work.

#### Update

`west update`, use to keep your Zephyr repositories up to date, if zephyr is already installed.

Creating a Zephyr Application

* Create a project folder with the following structure:

<app>

├── CMakeLists.txt

├── prj.conf

├── src

| └── main.c

└── boards

└── <board>.overlay or custom\_board.dts

* Set the ZEPHYR\_BASE environment variable to the Zephyr source code path:

set ZEPHYR\_BASE=C:\zephyrproject\zephyr

* Active Python virtual environment.

C:\zephyrproject\.venv\Scripts\activate.bat

**OR**

* Alternatively, use the `env.cmd` script to set up the environment automatically. Make sure to update the script with the correct Zephyr source path.

⚠️ Run the above command every time you open a new terminal.

## Build and Flash the Project:

* `west build` to compile the project, ex: `west build -p always -b samd21\_xpro app`
  + `-b <board>` ex: `-b samd21\_xpro`: Specifies the target board.
  + `-p` pristine: (optional)
    - `auto`
    - `always` => default option
    - `never`
  + Path to the <app> to build, if not specified use `.`
  + `-- -DDTC\_OVERLAY\_FILE=board.overlay` (optional): Specifies the path to an overlay file for the devicetree.
  + `-- -DBOARD\_ROOT=<path>` (optional): Specifies the path to the `boards` directory where the custom board is located. Need to be an absolute path.
* `west flash` to flash the compile project to the board
  + `--build-dir <path>` or `-d` (optional): Specifies the directory to build the project
  + `--runners jlink` or `-r` (optional): Specifies the runners if supported by the board

## Other West command

* `west build -t menuconfig` or `west build -t guiconfig` to see or change Kconfig variables.
* `west config build.board <board>` sets permanently the default board if `-b` not specified
* `west debug` => attach debugger to the board, open `gdb`
  + `--runner openocd` or `-r`

## Vscode setting and extensions

### Extensions

* `ms-vscode.cpptools-extension-pack` => C/C++
* `ms-vscode.cmake-tools` => CMake
* `ms-vscode.vscode-serial-monitor` => Serial monitor
* `marus25.cortex-debug` => Embedded Debugger
* `nordic-semiconductor.nrf-devicetree` => DevicetreeConfiguration

#### Settings.json

* `"C\_Cpp.default.compileCommands": <path>` after building the project add the path for the `compile\_commands.json` in the build directory.

#### Tasks.json

Create tasks to build and flash the code on the board, in the tasks specify the env variable `ZEPHYR\_BASE` to the zephyr code source, otherwise you will have a build error.

You can add input parameters if you want to dynamic parameter to the command run.

#### Launch.json

Create 2 launch configurations, one for launch and one for attach.

In each launch configuration add the following parameters:

* `name` => Name of the configuration
* `svdPath` => Path to svd file or ex: "[Microchip::SAMD21\_DFP@3.6.144](mailto:Microchip::SAMD21_DFP@3.6.144)"
* `deviceName` => Name of the board, need depending on the svd file, this setting is not recognise by Vscode but works.
* `device` => Name of the board
* `cwd`
* `executable` => Path to executable file .elf
* `request` => launch or attach
* `type` => cortex-debug
* `runToEntryPoint` => main function
* `servertype` => openocd, jlink ... or external to connect to an already existing session (with openocdm you want to connect to the debug session launch by `west flash`
* `gdbTarget` => Use only with serverType = external, define address to find the debugger server, ex: `:3333`
* `gdbPath` => path to gdb in zephyr sdk directory
* `preLaunchTask` => only for launch, command to run before debug, to build the project
* `configFiles` => for openocd, configuration file provides by zephyr

## Debug the Project

### Cortex Debug

Install the Vscode extension Cortex-Debug, then you need to create a `[launch.json](#_Launch.json)` file with 2 launch configuration. One to launch a debug session and one to attach a debug session. The launch session will build the project and flash it on the board then connects a debug session. The attach session will directly connect to the board executing code and open a debug session.

In the launch request you need to specify a `preLaunchTask` to build the project. So, you need to add a `[tasks.json](#_Tasks.json)` file with a task to build the project.

Go to the onglet `Run and debug` in Vscode, then chose the configuration you want to use, and start debugging.

To access the register, you need to specify a `svdPath` in the `[launch.json](#_Launch.json)`, and if needed a `deviceName`.

### West debug

Run the command `[west debug](#_Other_West_command)` in the terminal. It will open a `gdb` session.

Gdb Commands:

* `Ctlr + x`, then `2` => open the graphical interface
* `list` => with the graphical interface convert asm in C code
* `b <function>` => create a break point on a function
* `b <file:line>` => create a break point
* `b <address>` => create a break point
* `cont` or `c` => continue to run until the next break point or the end
* `next` or `n` => step over
* `step` or `s` => step in
* `backtrace` or `bt` => print callstack
* `print <var>` or `p <var>` => print variable, register…
* `info registers` or `i r` => print register value

## Devicetree

Used to define the hardware, all the peripherals. The devicetree of each predefined boards are in the directory /zephyr/boards/<vendor>/<board>/<board>.dts. Or for a custom board, the path is specified in BOARD\_ROOT.

The extension file .dtsi is like a header file that can be included in .dts file.

### Overlay

File with the extension `.overlay` are used to add peripheric or overwrite the devicetree of the original board.

Default location for devicetree when building the project:

1. socs/<SOC>\_<BOARD\_QUALIFIERS>.overlay
2. boards/<BOARD>.overlay => used in addition to the above.
3. boards/<BOARD>\_<revision>.overlay =>used in addition to the above.

If a devicetree is already found in the above location, it will stop looking for the next one.

1. <BOARD>.overlay => stop looking if it exists
2. app.overlay

Or use `-- -DDTC\_OVERLAY\_FILE=board.overlay` to explicitly define the path to the board in the `west build` command. It takes the priority over the default location define above.

During build the following line while be print if zephyr generated the devicetree via an overlay file:

-- Found devicetree overlay: .../some/file.overlay

You can add new node(device) to the devicetree or modify existing node.

To add node:

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

led2: led2 the first led2 is a label. It’s used to avoid specifying the full path (leds\_led2) by just using led2.

Warning: There should never be a space between the first led2 and colon, otherwise an error occurred. led2 : led2 => invalid, led2: led2 =>valid

To modify existing node (label):

Une image contenant Police, texte, capture d’écran, Graphique

Le contenu généré par l’IA peut être incorrect.

Depending on the `compatible` key each node will need different parameters, which can be found in the doc. It can be specific to the board used.

Led => `compatible = "gpio-leds"`

Button => `compatible = "gpio-keys"`

After building the project, the complete devicetree used will be in the file `<build directory>/zephyr/zephyr.dts`

### Custom board

In your app directory `boards/<vendor>/<board\_name>/` that will contain all the files describing the board. If no vendor name, it `other`. This directory will contain Kconfig and overlay to define the board, it’s also possible to define some code source with a CMakeLists.txt containing the following:

zephyr\_library()

zephyr\_library\_sources(source\_file.c)

## Kconfig

Used to enable or disable some features. The file `prj.conf` will contain the Kconfig at the app root. Kconfig will include the code source needed for your app.

* To enable Gpio: `CONFIG\_GPIO=y`
* To enable Pwm: `CONFIG\_PWM=y`
* …

The following commands are used to see the Kconfig

west build -t guiconfig => all the red configurations are not user configurable

west build -t menuconfig

Sometimes the config you want to use is not directly user-configurable (has no prompt). To go around the problem, there are 2 solutions. If the config you want to add is `CONFIG\_GPIO` the XX will be `GPIO` in the following solutions.

1. In the board definition folder, in the file `Kconfig.<board>`, under config <board>, add select XX, where XX is the config to add. This will add the config to all the projects that are using this board. If it’s not a custom board, the folder is located in zephyr source code.

Une image contenant texte, Police, capture d’écran, nombre

Le contenu généré par l’IA peut être incorrect.

1. The second option is to add a file named `Kconfig` to your project directory at the root. In the file, create a new configuration like this:

Une image contenant texte, Police, capture d’écran, nombre

Le contenu généré par l’IA peut être incorrect.

The new configuration in named NEW\_CONFIG in this example and is active by default. XX is the existing config that will be enabled by the new config, bypassing the error “not user configurable”. There is no need to specify somewhere that this file exists, zephyr will find it automatically if it has the correct name and location.

After building the project, the complete kconfig used will be in the file `<build directory>/zephyr/.config`

In the board directory the Kconfig as the name `<board>\_deconfig` or `Kconfig.<board>`.

The `<build directory>/zephyr/.config` file is generated using the following files:

* boards/<VENDOR>/<BOARD>/<BOARD>\_defconfig
* soc/<VENDOR>/Kconfig
* CONFIG\_... in CMake
* boards/<board>.conf in application
* boards/<board>\_<revision>.conf in application
* Prj.conf in application

## CMake

The `CMakeLists.txt` is located at the root of the app.

The default file contains:

Une image contenant texte, capture d’écran, Police, ligne

Le contenu généré par l’IA peut être incorrect.

Before pulling the project zephyr, you can define any variable you want.

To add other files in the project you can add a list of source files in `target\_sources`, ex:

Une image contenant texte, capture d’écran, Police, ligne

Le contenu généré par l’IA peut être incorrect.

You can also add the path of the directory where to find header files with:

target\_include\_directories(app PRIVATE src)

Where `src` is a directory in the app project.

## Multiple configuration (Debug, Release …)

To have multiple configurations with different Kconfig or overlays, you need to define a variable in the CMakelists.txt file. This variable will be used to define the configuration used for the compilation. Depending on this configuration, you can then add the right Kconfig or overlay tree.

For example :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

For this example, at the end of the west build command, you need to add the following flag : `-- -DBUILD\_MODE=debug` where `BUILD\_MODE` is the name of the variable defined.

## Power Management

It’s supported by zephyr for the board. Use the following Kconfig parameters to enable the use of low consumption:

* `CONFIG\_PM` => enable power management when the board is idle
* `CONFIG\_PM\_DEVICE` => enable power management for devices.
* `CONFIG\_PM\_DEVICE\_RUNTIME`

Then, in the device tree, add multiple possible states for the `cpu` node and define them in the `power-states` node.

Then, when you use the `k\_sleep` function or anything that puts the cpu in idle mode, the thread idle will choose the most adapted idle mode for the idle duration. So you will have to do nothing, and zephyr will do it for you.

You must NEVER perform manual sleep in the main thread, because it can break something else when you use `k\_sleep` or an equivalent function.

## Initialization before main

If some device or driver needs to be initialized before the main function, you can use the macro `[SYS\_INIT(function, level, priority)](https://docs.zephyrproject.org/latest/doxygen/html/group__sys__init.html#gaf507cc0613add8113c41896bd631254f)`. For example, you can configure the clock with it.

## Lora (SX126X)

Define in the device tree the node `sx126x` in the `spi` node using the compatible ` semtech,sx1261` or ` semtech,sx1262`. By default the device sx126x will be put to sleep during the initialization before the main function. The code source of the loramac node is included by default in the Kconfig. You don’t need to do anything else. The device tree binding ` semtech,sx1262` initialize undocumented register to produce a continuous wave, but not the sx1261.

For loraWan use `lorawan\_start()` , then `lorawan\_join()` with all the key needed then you can do downlink or send data to the server. For the join, the Network key is the same as the application key.

## SystemView

Install [Systemview](https://www.segger.com/downloads/systemview/) and [j-link](https://www.segger.com/downloads/jlink/).

Build your zephyr project with the flag `-S rtt-tracing` and flash it

Ex: west build -p always -b nxg\_court -S rtt-tracing zephyr\_nxg\_court

west flash

Open system view and start recording. Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Le contenu généré par l’IA peut être incorrect.

In the time line you can see the switch between all the tasks in real time, as well as the interrupt and scheluder, and idle thread.

Une image contenant capture d’écran, texte, logiciel, ligne

Le contenu généré par l’IA peut être incorrect.

The event list show all the event track by zephyr by default and the one you may have had with the timestamp and contextual information.

Une image contenant texte, logiciel, Icône d’ordinateur, Page web

Le contenu généré par l’IA peut être incorrect.

In system view system you can chose the event you want to track or not.

Une image contenant texte, logiciel, Icône d’ordinateur, Page web

Le contenu généré par l’IA peut être incorrect.

In the windows terminal you can track messages print via the function ` SEGGER\_SYSVIEW\_PrintfHost()`, ` SEGGER\_SYSVIEW\_WarnfHost ()`, ` SEGGER\_SYSVIEW\_ErrorfHost()`

For more detail the complete doc : https://www.segger.com/downloads/jlink/UM08027\_SystemView.pdf

To add custom event, you need to edit the "C:\Program Files\SEGGER\SystemView\Description\SYSVIEW\_Zephyr.txt". At the end of the file you can add event, and each time increment the event number, change the event name and description. In your code source you can add ` SEGGER\_SYSVIEW\_RecordXX(id\_event, XX)` or `SEGGER\_SYSVIEW\_RecordEndCallXX(id\_event, XX)` to track the beginning and the end of the event if needed. With XX you can choose the number of argument you want to record. When installing SystemView by default the file "C:\Program Files\SEGGER\SystemView\Description\SYSVIEW\_Zephyr.txt" is already created, but if it’s not up to date, Zephyr provides the version updated in “.\zephyr\subsys\tracing\sysview\SYSVIEW\_Zephyr.txt” that need to be move inside "C:\Program Files\SEGGER\SystemView\Description\”

## RTT

Install [j-link](https://www.segger.com/downloads/jlink/).

In the Kconfig add the configuration :

CONFIG\_USE\_SEGGER\_RTT=y

CONFIG\_LOG\_BACKEND\_RTT=y ->log using rtt, can use rtt and uart at the same time

CONFIG\_RTT\_CONSOLE=y -> work only if uart is disable for the console

CONFIG\_UART\_CONSOLE=n

## Low Power STM32 (nucleo f411re)

I use the sample: `zephyr\samples\boards\st\power\_mgmt\blinky` as a base. On this sample, the PM is already activated. That means when using k\_sleep() for a long time, it will put the MCU into stop mode. And the following line:

DT\_FOREACH\_STATUS\_OKAY(st\_stm32\_gpio, STM32\_GPIO\_PM\_ENABLE)

will put all the gpio in suspended mode before entering stop mode to reduce power consumption.

Without modifying the sample, in stop mode, the board consumes 80 μA.

To reduce the consumption, it’s possible to use a more efficient stop mode by modifying the source code. In the file `zephyr\soc\st\stm32\stm32f4x\power.c` I changed the argument of the function LL\_PWR\_SetPowerMode(); to LL\_PWR\_MODE\_STOP\_LPREGU\_DEEPSLEEP. It puts the flash in deep power-down mode, using a low-power, low-voltage regulator. I now consume 40 μA.

To reduce it even more, it’s possible to put all the unused pins in analog input. But zephyr doesn’t have an easy way to do so.

1. The first option is to use the hal or ll of stm32 to put all the pins in analog input before the main execute, as the default configuration for the board.

Either in the application or in the board directory, create a c file that contains the function to set all the pins. At the end of the file add:

SYS\_INIT(function, PRE\_KERNEL\_1, 0);

It calls the function before the main, you can choose the level of priority, and level of execution

1. The second option, in the devicetree, add a node zephyr, user listing all the pins to set in analog.Une image contenant capture d’écran, texte, Police

   Le contenu généré par l’IA peut être incorrect.

Then, in the code in C add the following to configure the pinctrl described in the device tree.

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.