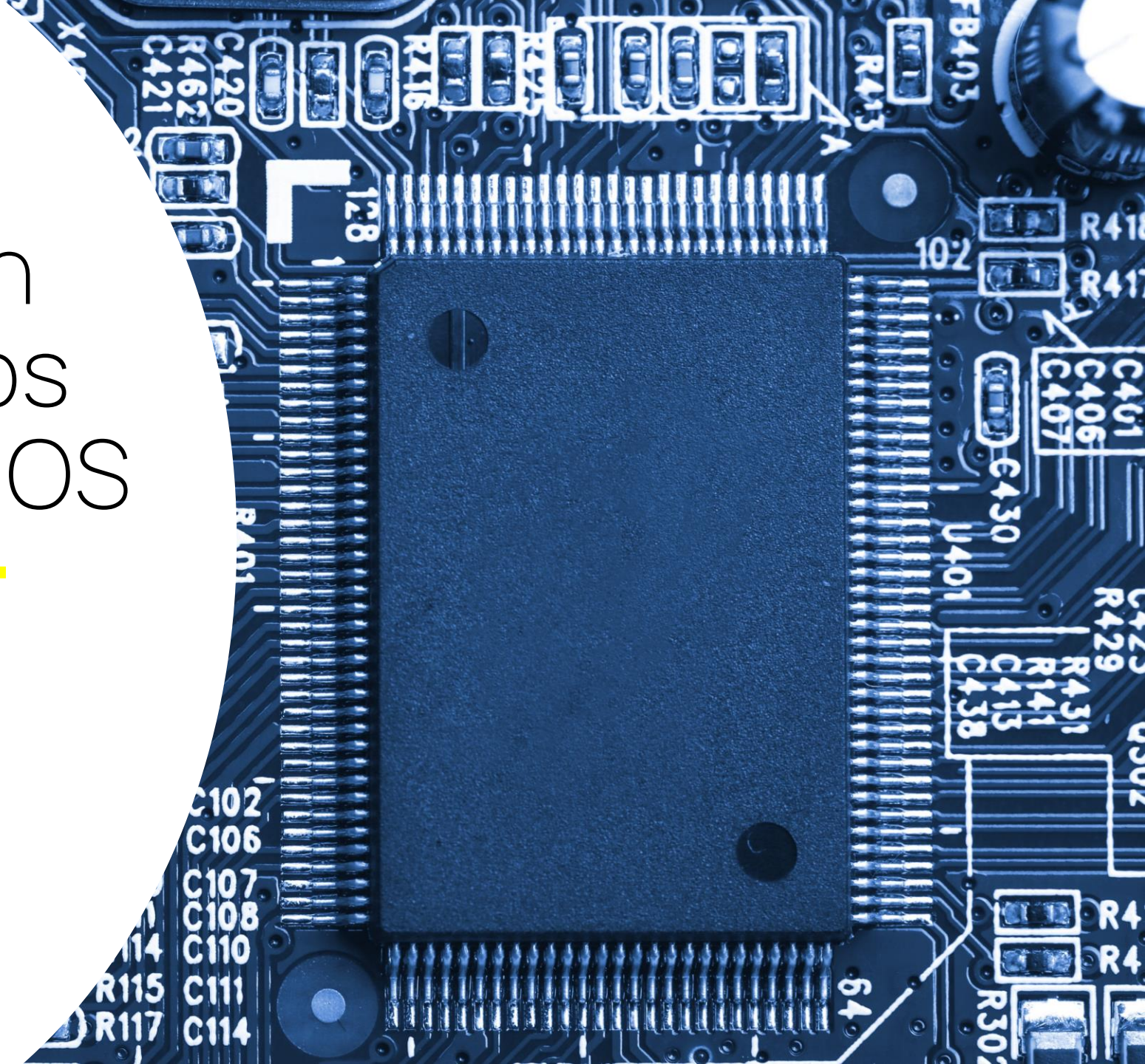
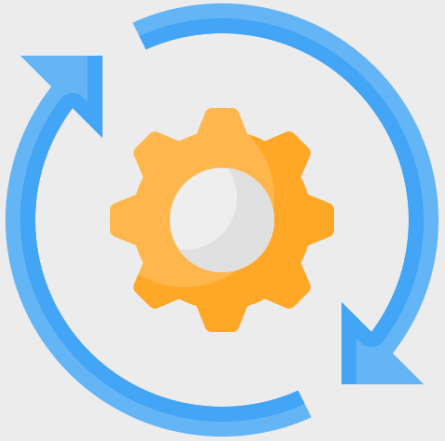
Programmation embarqué temps réel avec Zephyr OS

SNIOT – SEANCE 3



Objectifs

- Comprendre ce qu’est un système temps réel
- Savoir choisir une solution adaptée au projet
- Programmer avec l’environnement de développement Zephyr



Système temps réel - définition

- Un système temps réel est un système qui doit respecter des contraintes de temps. Autrement dit il doit réagir à un processus extérieur et produire un résultat juste dans le temps imparti.

Aéronautique

Aérospatiale

Médicale

Industrielle

Système temps réel - conséquences

- Le système doit réagir à un événement avant un délai fixé
- Le système ne doit pas rater des événements
- Le non-respect de la contrainte de temps peut être
  - Catastrophique => temps réel dure
  - Tolérable => Temps réel mou
- Système temps réel != système rapide

Système temps réel - application

Quelques exemples

- Guidage d’un missile
- Système de freinage de voiture
- Calculateur dans les avions
- Fermeture des portes du métro
- …

Bare metal vs Système temps réel

Bare metal

Léger

Temps d’exécution très rapide

Contrôle total

Application complexe

Fiabilité

Temps réel

Fiabilité

Thread, timer, fifo, lifo, stack …

Application complexe

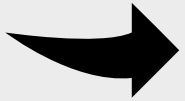
Temps contraint

Légèrement moins efficace

Debugging (programmation concurrente)

Quand utiliser un système temps réel

- Application nécessitant des contraintes de temps strictes
- Application critique



- Application embarqué complexe (le temps réel permettra une meilleure montée en complexité du système)
- Préférer une utilisation bare metal pour des utilisations simples

Les systèmes temps réel

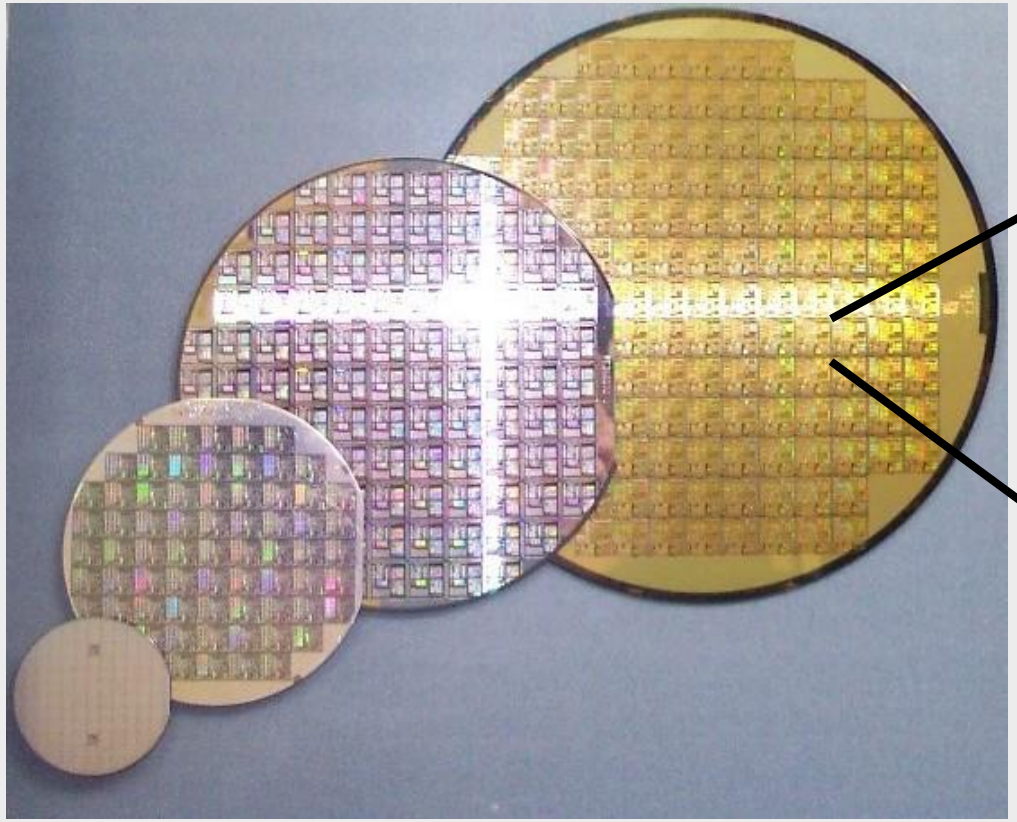
- RTOS
- ZephyrOS
- Linux temps réel
- Java temps réel

Real-Time Project

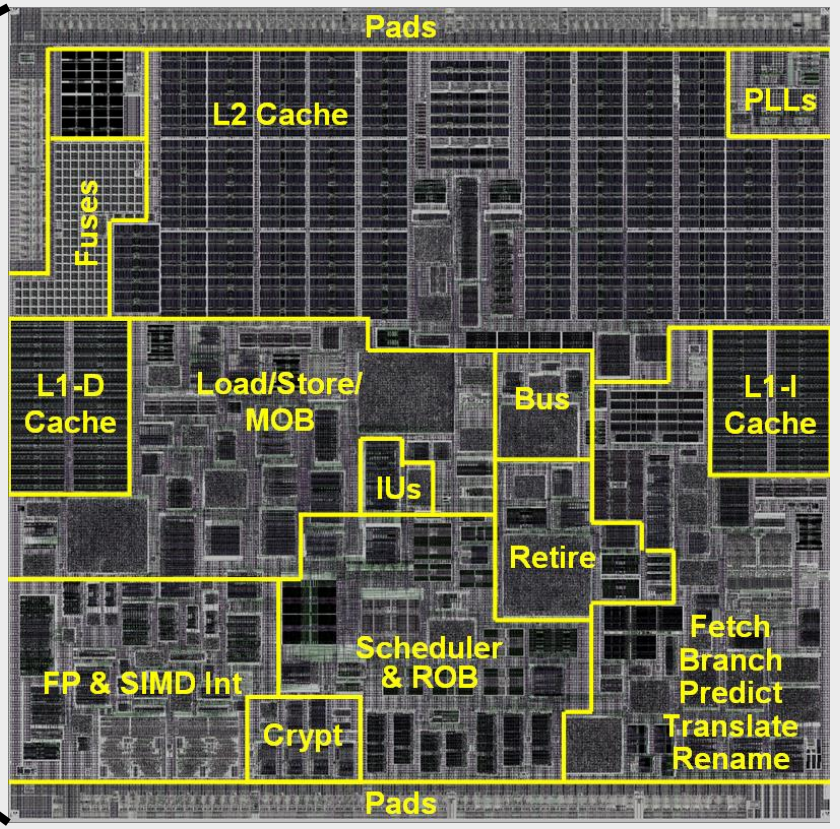
Architecture des microcontrolleurs

Fabrication

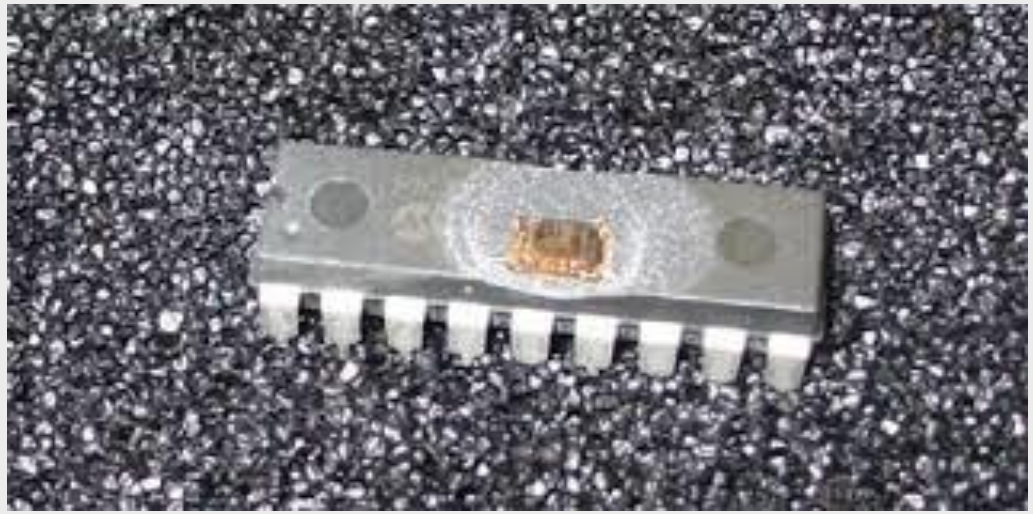
Wafer de silicium

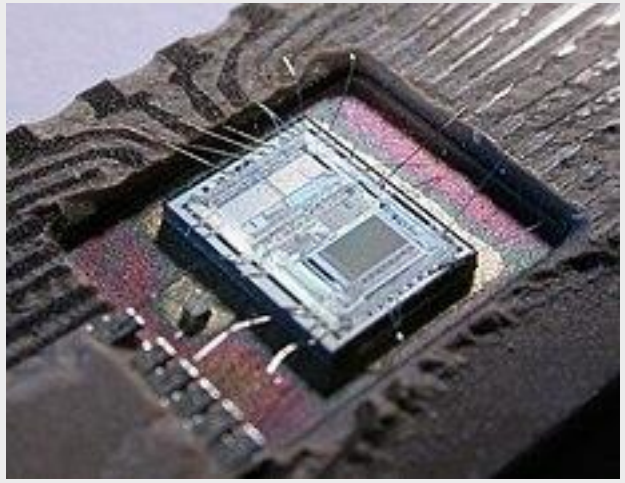


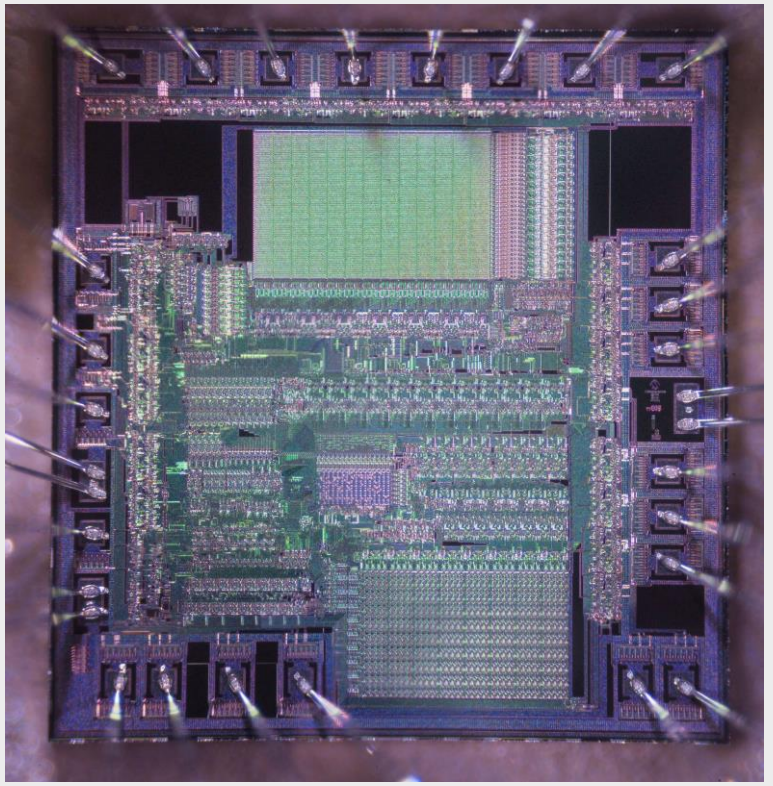
Zoom sur la puce



Microcontrôleur

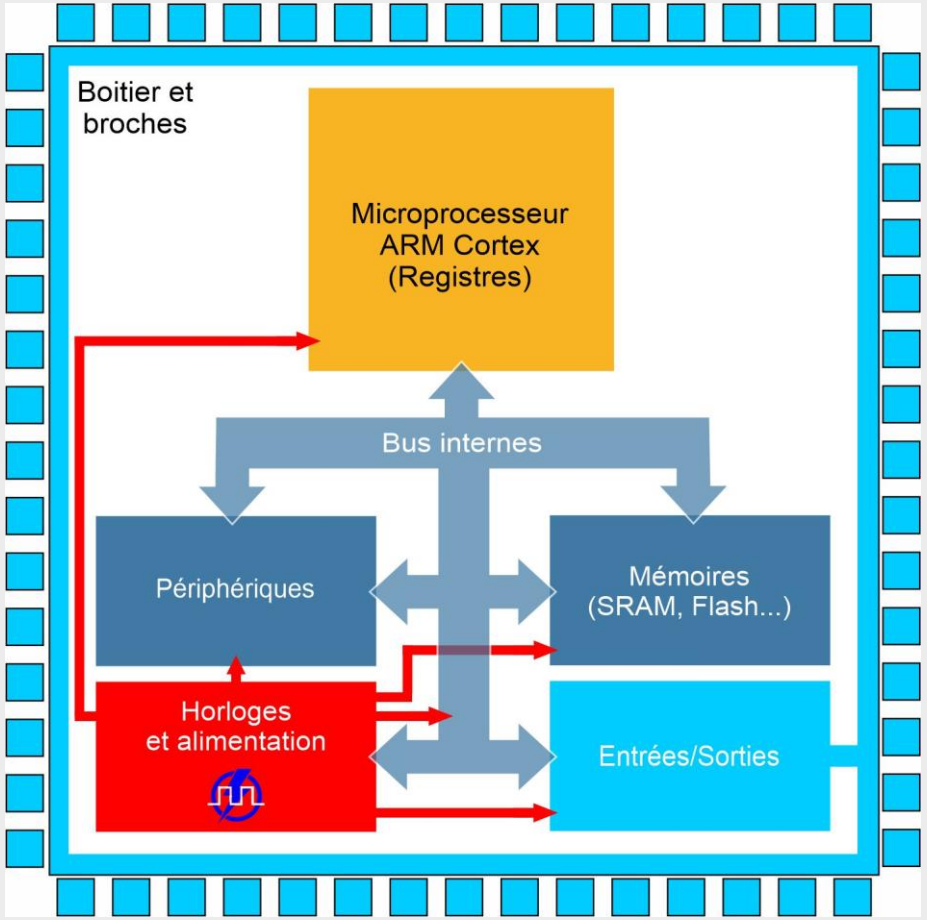






Architecture

Périphériques:
- I2C
- UART
- CAN
- ADC / DAC
- …



Mémoires ROM et RAM

GPIO

Les fondamentaux d'un OS temps réel

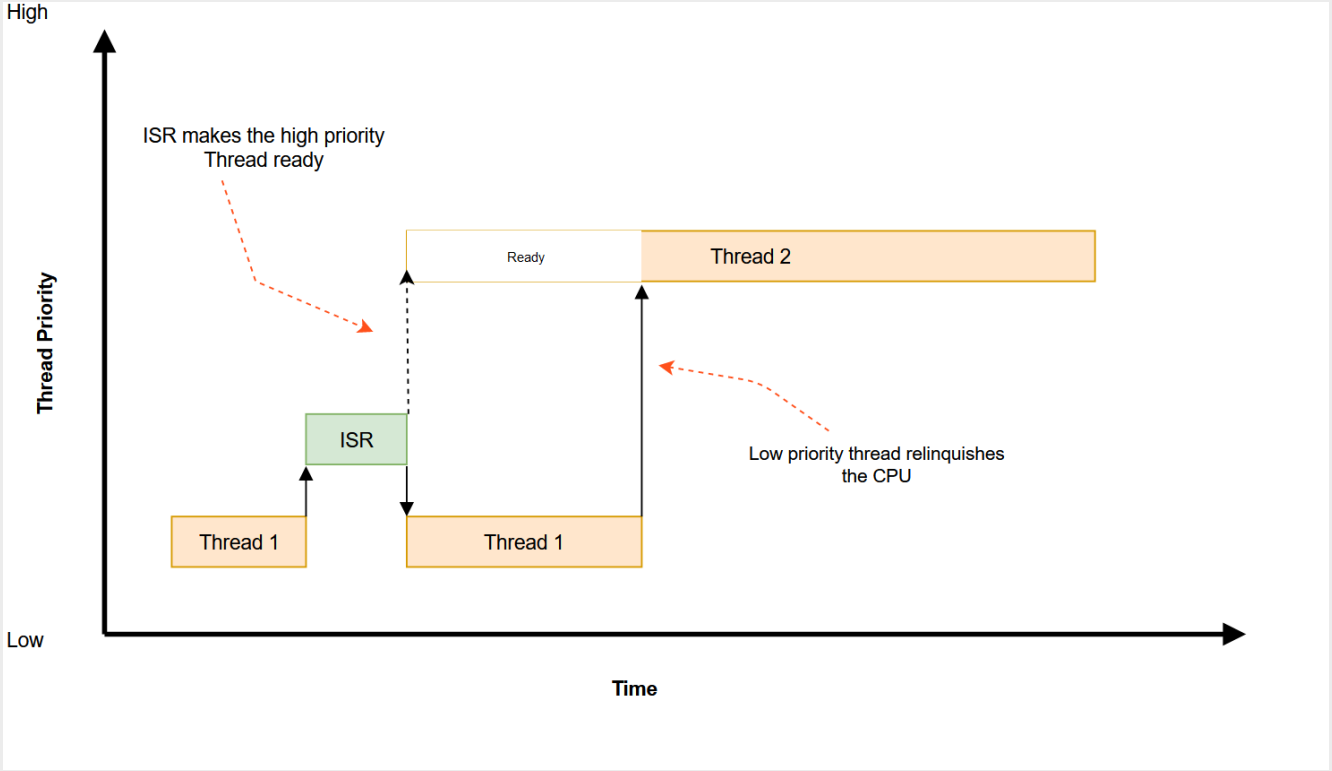
Ordonnanceur

- L'ordonnanceur permet de cadencer l'ordre d'exécution des instructions par le CPU. Il coordonne l'exécution des différentes tâches du système (=> des différentes threads)
- Il décide de l'ordre d'exécution des différents processus en fonction des priorités
- Pour donner un ordre d'idée sur un processeur NRF5340 en Zephyr le changement d'une tâche par une autre de l'ordonnanceur est réalisé en quelques dizaines de microsecondes

https://www.techniques-ingenieur.fr/base-documentaire/technologies-de-l-information-th9/systemes-embarques-42588210/ordonnancement-temps-reel-s8055/

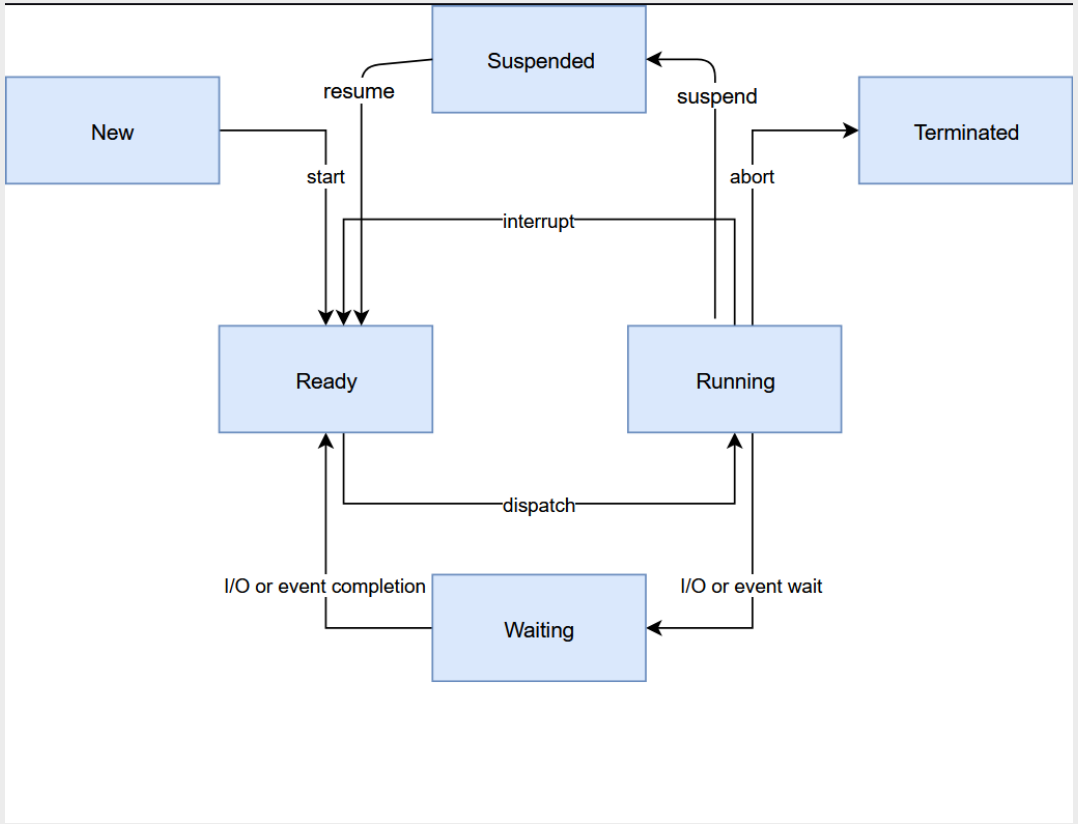
Thread

- Un thread permet l'exécution d'une tâche spécifique
- Il dispose de sa propre **Stack**



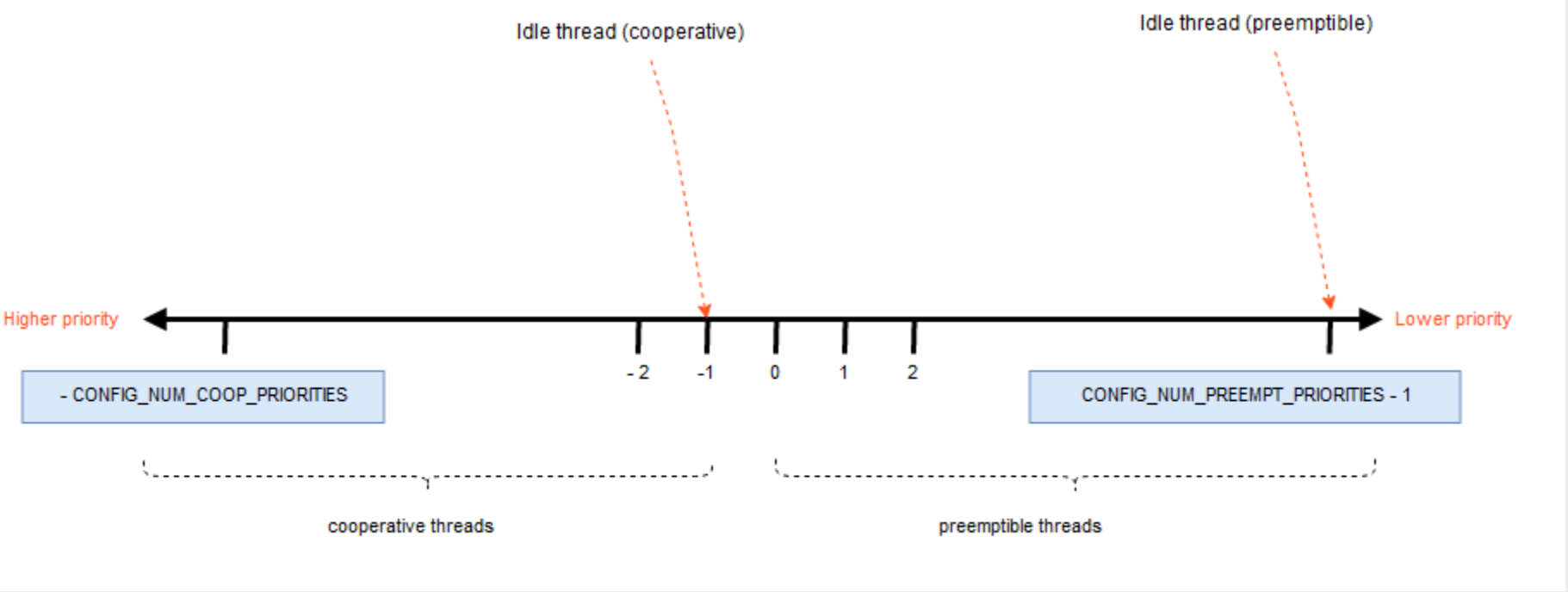
Thread - Lifecycle

- Un Thread à un cycle de vie en Zephyr



Thread - Priority

- Zephyr défini une priorité de thread
  - A *cooperative thread* : qui s'arrête seulement quand il a terminé sa tâche
  - A *preemptible thread:* qui peut être suspendu par une interruption ou un cooperative thread

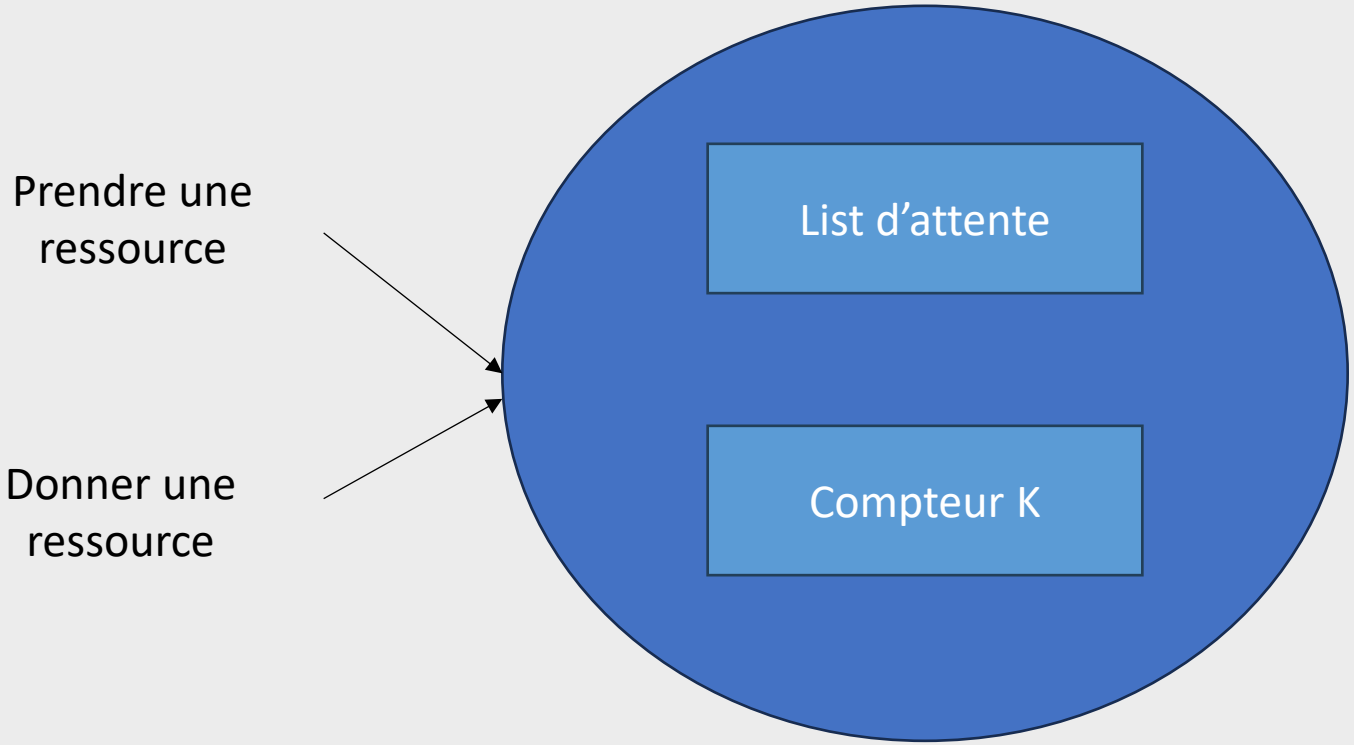


Mutex

- Permet de protéger une ressource de la lecture et de l'écriture concurrente
- Permet d'assurer l'intégrité de la donnée lors de l'accès concurrent à une variable par exemple

Sémaphore

- Permet de restreindre l’accès à une ressource partager et de synchroniser des processus



Procédure prendre une ressource (P)
Si K > 0
    Décrémenter K
    Exécuter le processus
Sinon (K = 0)
    Suspendre le processus
    Placer le processus dans la file d'attente

Procédure donner une ressource (V)
Si la file est vide
    Incrémenter k
Sinon
    Prendre le processus élément de la file d'attente
    Réveiller le processus

ZephyrOS

Structure du projet

```
zephyrproject/
├── zephyr_smart_home/
│   ├── app.overlay
│   ├── CMakeLists.txt
│   ├── prj.conf
│   ├── README.rst
│   ├── inc/
│   │   └── foo.h
│   └── src/
│       ├── main.c
│       └── foo.c
├── zephyr
└── build
```

app.overlay : Overlay pour la description matérielle du device tree

CMakeLists.txt : Fichier pour la compilation avec make. Il faut renseigner les fichiers sources .c ici.

prj.conf : Fichier de configuration pour la compilation

README.rst : Fichier de description du projet. Ne pas oublier de bien rédiger son README avec les informations utiles ! (non spécifique à Zephyr)

Description matérielle: Devicetree

Permet de:

- décrire le hardware disponible sur la *board*
- Décrire la configuration initiale

=> C'est donc un langage de description matérielle et de configuration

```
/ {

    chosen {
        zephyr,console = &uart0;
    };

    leds {
        compatible = "gpio-leds";
        led0: led_0 {
            gpios = <&gpio0 28 GPIO_ACTIVE_LOW>;
            label = "Green LED 0";
        };
        led1: led_1 {
            gpios = <&gpio0 29 GPIO_ACTIVE_LOW>;
            label = "Green LED 1";
        };
    };

    pwmleds {
        compatible = "pwm-leds";
        pwm_led0: pwm_led_0 {
            pwms = <&pwm0 0 PWM_MSEC(20) PWM_POLARITY_INVERTED>;
        };
    };

    buttons {
        compatible = "gpio-keys";
        button0: button_0 {
            gpios = <&gpio0 23 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
            label = "Push button 1";
        };
    };
};
```

Description matérielle: Overlay

- Permet d'override la configuration du ***devicetree***
- Doit être défini dans le projet avec la nomenclature ***[nom_fichier].overlay***

```
/ {

    gpio_keys {
        compatible = "gpio-keys";
        ledyellow: led_yellow {
            gpios = <&gpio0 12 GPIO_ACTIVE_HIGH>;
            label = "LED 1";
        };
    };

    aliases {
        led-yellow = &ledyellow;
    };
};


&wifi {
    status = "okay";
};
```

Exemple: définition d'un GPIO et activation du driver wifi

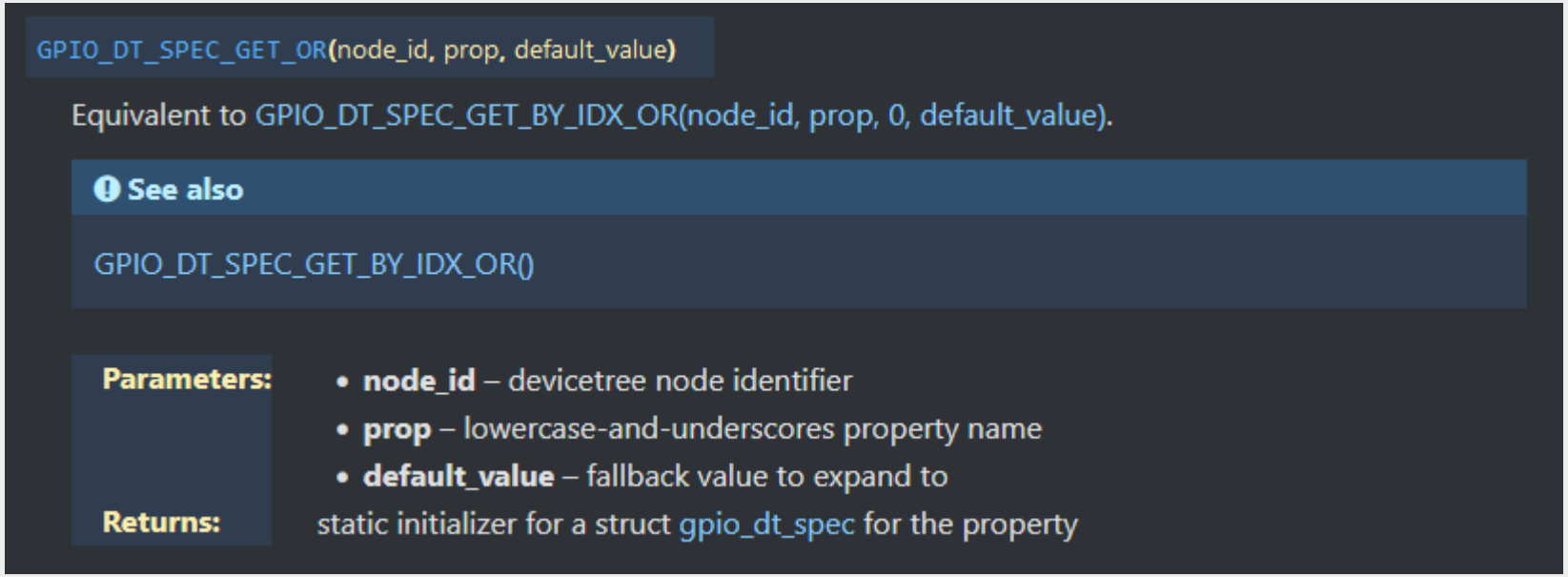
Devicetree: Récupération de la configuration matérielle

- Utilisation de ***macro*** pour récupérer les éléments du devicetree

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

#define LED_YELLOW_NODE DT_ALIAS(led_yellow)

const struct gpio_dt_spec led_yellow_gpio = GPIO_DT_SPEC_GET_OR(LED_YELLOW_NODE, gpios, {0});

int main(void) {

    gpio_pin_configure_dt(&led_yellow_gpio, GPIO_OUTPUT_HIGH);
}
```

Devicetree: Récupération de la configuration matérielle

- Macro `GPIO_DT_SPEC_GET_OR`



`GPIO_DT_SPEC_GET_OR`**(node_id, prop, default_value)**

Equivalent to GPIO_DT_SPEC_GET_BY_IDX_OR(node_id, prop, 0, default_value).

See also

GPIO_DT_SPEC_GET_BY_IDX_OR()

Parameters:

- **node_id** – devicetree node identifier
- **prop** – lowercase-and-underscores property name
- **default_value** – fallback value to expand to

Returns: static initializer for a struct gpio_dt_spec for the property

Thread

- Définir un thread avec une macro

```
void thread_function() {
        // Do something
}


K_THREAD_DEFINE(thread_id, 521, thread_function, NULL, NULL, NULL, 9, 0, 0);
```

Thread

- Définir un thread sans macro

```
extern void my_entry_point(void *, void *, void *);

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);
struct k_thread my_thread_data;

k_tid_t my_tid = k_thread_create(&my_thread_data, my_stack_area,
                                 K_THREAD_STACK_SIZEOF(my_stack_area),
                                 my_entry_point,
                                 NULL, NULL, NULL,
                                 MY_PRIORITY, 0, K_NO_WAIT);
```

Mutex

- Voir la doc

Configuration: prj.conf

- Configuration du projet à partir des configs définies dans les fichiers ***Kconfig***

```
# Sensor
CONFIG_I2C=y
CONFIG_SENSOR=y
CONFIG_ADC=y
```

Exemple: configuration de l'I²C, de l'ADC et de l'activation du driver sensor

Fichier CMakeLists.txt

- Fichier pour la compilation des sources

```
cmake_minimum_required(VERSION 3.20.0)

find_package(Zephyr REQUIRED HINTS
$ENV{ZEPHYR_BASE})
project(zephyr_smart_home)

target_sources(app PRIVATE
    src/main.c
)
```

Installation de l’environnement Zephyr

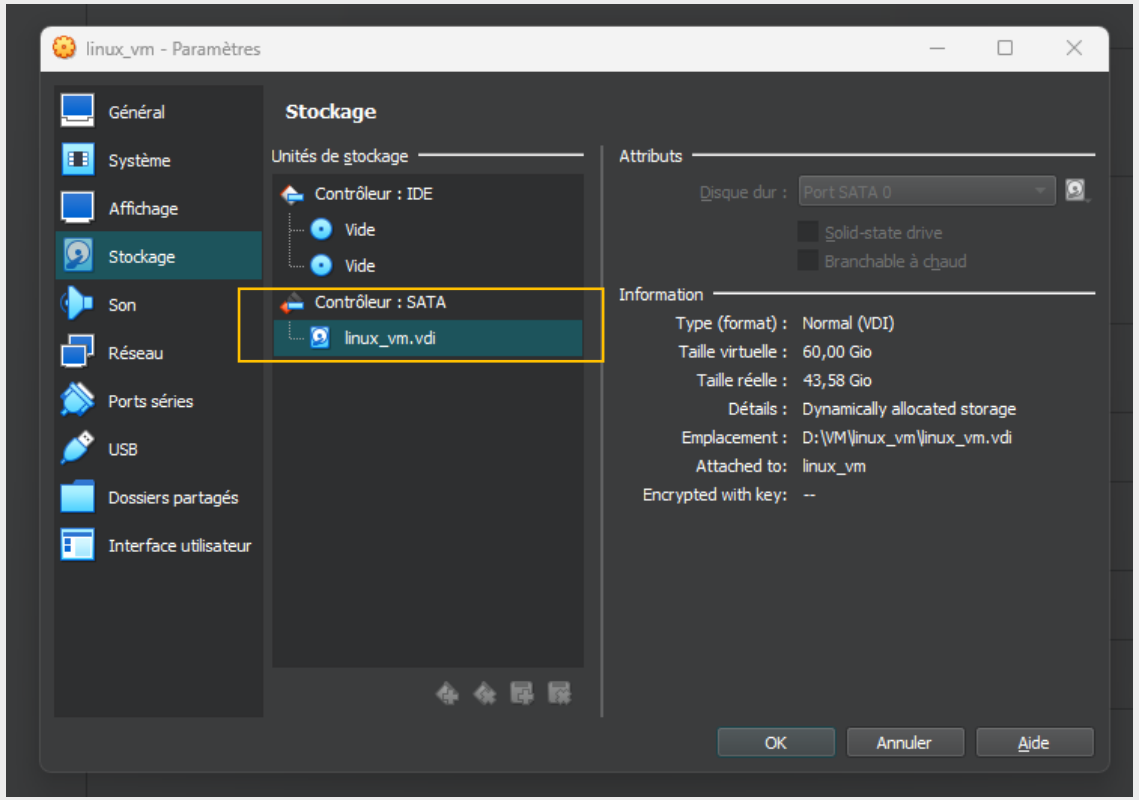
Installer Zephyr et West

Nous allons coder dans un environnement Linux

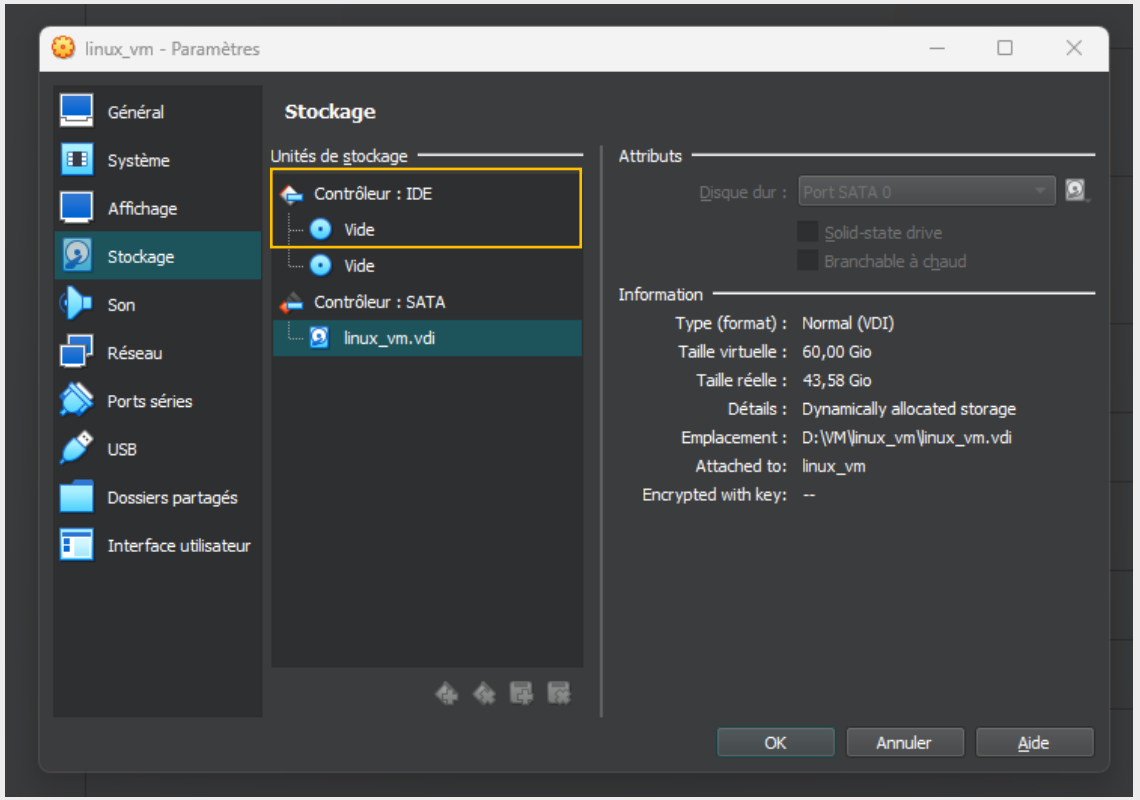
1. Si utilisation de la VM Seatech => augmenter l'espace disque disponible
2. Installer Zephyr et West https://docs.zephyrproject.org/latest/develop/getting_started/index.html
3. Installer VSCode est les extensions
4. Créer un repertoire git zephyr_smart_home dans votre workspace
5. Initialiser le repo git et le push sur un remote (github)

1. Utilisation de la VM Seatech

1. Augmenter l’espace disque disponible sur le support vdi



2. Monter l’iso gparted live et allouer l’espace disponible au disque système



2. Installation de Zephyr et West

Suivre le guide Getting Started du site de Zephyr:

https://docs.zephyrproject.org/latest/develop/getting_started/index.html

3. Installation de VScode

https://code.visualstudio.com/download

+

Installer les extensions utiles (C/C++)

4. Initialisation du repo git et github

- Installation de git

```
> sudo apt install git-all
```

- Installation de l’outil gh

```
> sudo apt install gh
```

```
> gh auth login
```

Flasher la carte ESP32

Pour flasher sur ESP32:

- Supprimer le package brltty: https://stackoverflow.com/questions/73487141/esp32-flahing-from-ubuntu

```
> sudo apt remove brltty
```

- Rajouter les permissions sur le port usb (à faire à chaque démarrage !) https://stackoverflow.com/questions/73487141/esp32-flahing-from-ubuntu

```
> chmod 0777 /dev/ttyUSB0
```

Flasher la carte ESP32

source ~/zephyrproject/.venv/bin/activate

- Dans zephyrproject

```
> source ~/zephyrproject/.venv/bin/activate
```

```
> west build -b esp32_devkitc_wroom zephyr_smart_home/ --pristine
```

- Dans zephyrproject

```
> west flash
```

```
> west espressif monitor -p /dev/ttyUSB0
```

Projet – Smart Home

Consignes

- Rendus:
  - Repo github
  - Rapport de projet rempli
  - Démonstration
- En monôme dans la limite du matériel disponible

Github du projet

https://github.com/Vivoulia/sniot-zephyr

Description des inputs / outputs

Interface	Type	Port ESP32
LED Orange	GPIO	12
Buzzer	GPIO	25
PIR Sensor	GPIO	14
DTH11	GPIO	17
BUTTON 0 (Gauche)	GPIO	16
BUTTON 1 (Droite)	GPIO	27
STEAM SENSOR	ANALOG	AIN 06
LCD Screen	I2C	I2C0