Zephyr OS

Adscripto: Agustín Zuliani - Adscripción 2025
Director: Prof. Ing. Daniel Márquez

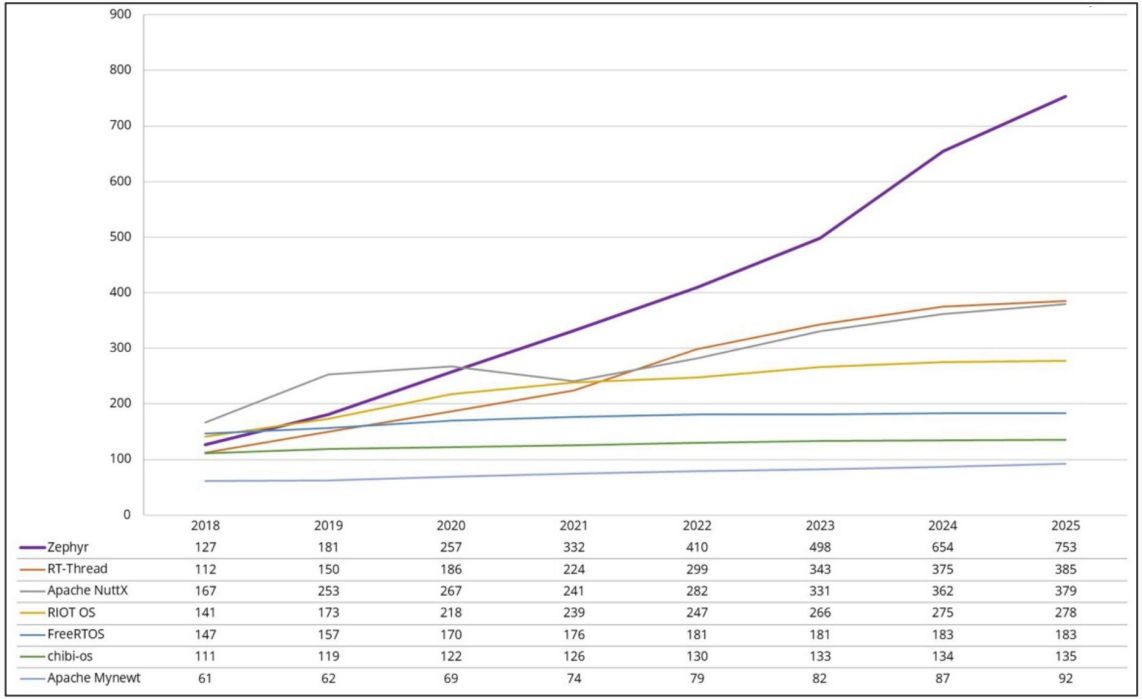
Departamento de Sistemas e Informática (DSI) - FCEIA

Sistemas Embebidos Avanzados II

Introducción

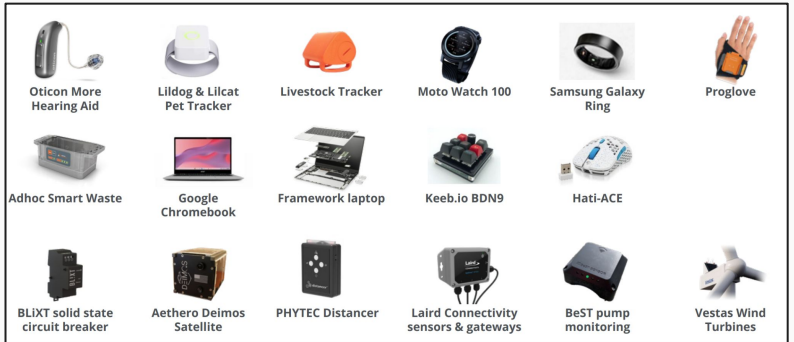
[Link - Pagina oficial]

¿Qué es Zephyr OS? Es un sistema operativo en tiempo real (RTOS) de código abierto, ligero y escalable, diseñado principalmente para dispositivos con recursos limitados, como los del Internet de las cosas (IoT). Soporta múltiples arquitecturas de hardware, ofrece una arquitectura modular, incluye un subsistema de red para protocolos como Bluetooth y Zigbee, y está desarrollado bajo la Fundación Linux. En la siguiente imagen se observa una comparación del soporte de placas de Zephyr respecto a otros RTOS's.



	2018	2019	2020	2021	2022	2023	2024	2025
Zephyr	127	181	257	332	410	498	654	753
RT-Thread	112	150	186	224	299	343	375	385
Apache NuttX	167	253	267	241	282	331	362	379
RIOT OS	141	173	218	239	247	266	275	278
FreeRTOS	147	157	170	176	181	181	183	183
chibi-os	111	119	122	126	130	133	134	135
Apache Mynewt	61	62	69	74	79	82	87	92

- Sistema operativo liviano para sistemas embebidos de pocos recursos.
- Bajo consumo, eficiente y seguro.
- Enfocado a la conectividad:
  - Bluetooth 5.0 & BLE
  - Wi-Fi, Ethernet, CANbus, ...
  - IoT protocols: CoAP, LwM2M, MQTT, OpenThread, ...
  - USB & USB-C
- Entorno completo para el desarrollador
  - Toolchain and HAL management
  - Logging, tracing, debugging
  - Emulation/Simulation
  - Testing framework



Productos que usan Zephyr

Posee poderosas herramientas de configuración:

- Kconfig
- Device Tree

Cmake

CMake es un sistema de configuración y generación de builds multiplataforma.

No compila el código directamente, sino que genera los archivos de proyecto (Makefiles, Ninja build files, Visual Studio solutions, etc.) que luego se usan para compilar.

Página oficial

¿Cómo se integra con Zephyr? Dado que Zephyr es modular de alguna manera necesita una herramienta que le permita administrar las siguientes configuraciones:

- Arquitectura (ARM, RISC-V, x86, etc)
- Incluir controladores
- Configurar la compilación cruzada (PC → Micro)
- Integrarse con Kconfig
- Generar build para Linux, Mac, Windows

Se dará una guía inicial de ciertos comandos de cmake a modo de orientación para ser utilizados junto con Zephyr. La guía está basada en la provista por la página oficial: Guia de ejercicios para utilizar Cmake

Nota: se siguieron el Step 1 y 2 para esta guía básica.

Todos los proyectos que utilizan Cmake, deben incluir un archivo: “CmakeLists.txt” ahi es donde se pondrán los comandos necesarios.

Luego debe establecerse el nombre del proyecto eso se realiza con el siguiente comando:

```
project(
    Tutorial
)
```

Cualquier proyecto debe comenzar especificando una versión mínima de CMake utilizando el comando:

```
cmake_minimum_required(VERSION 3.10)
```

Necesitamos crear un ejecutable especificando el nombre del target y el código fuente:

```
add_executable(
    Tutorial
    tutorial.cpp
)
```

Recursos:

- add_library()
- add_subdirectory()
- target_include_directories()
- target_link_libraries()
- PROJECT_SOURCE_DIR

Añade la librería anterior al proyecto:

```
add_subdirectory(
    MathFunctions
)
```

Le dice al compilador que agregue los siguiente archivos fuentes:

```
add_library(
    MathFunctions
    MathFunctions.cxx
    mysqrt.cxx
)
```

Necesitamos decir que se enlace el ejecutable con la librería de antes:

```
target_link_libraries(
    Tutorial
    PUBLIC
    MathFunctions
)
```

Hilos

Un hilo es un objeto del núcleo que se utiliza para el procesamiento de aplicaciones que son demasiado largas o complejas para ser realizadas por un ISR.

Threads - Zephyr

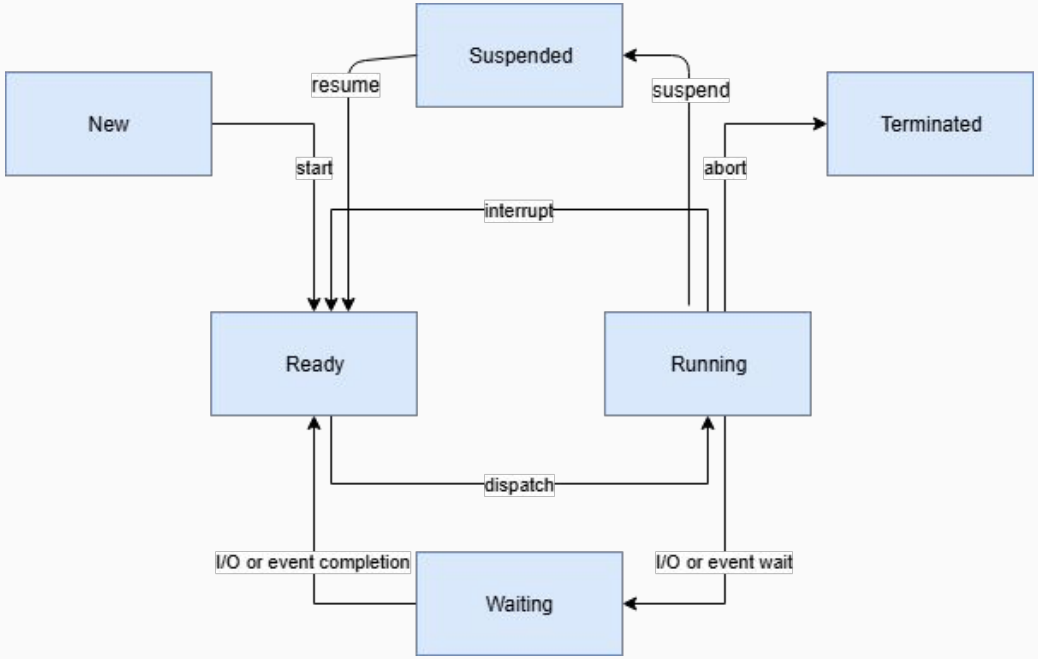
Teoria/Ejemplo - Github

Los hilos tienen las siguiente propiedades:

- **Stack Area**: región de memoria utilizada para la pila del hilo.
- **Thread Control Block**: se utiliza para la gestión de los metadatos del subproceso por parte del núcleo. Se define cuando se crea una variable del tipo "*k_thread*".
- **Entry Point Function**: se pueden pasar hasta tres valores como argumento de la función.
- **Scheduling Priority**: indica al scheduling como asignar el tiempo del CPU a cada hilo.
- **Thread Options**: permite que el hilo reciba un tratamiento especial. Por ejemplo se puede decir que un hilo trabaja con punto flotante, o decirle que es un hilo esencial para el funcionamiento con lo cual si deja de ejecutarse se detiene todo el sistema.
- **Start Delay**: especifica cuánto tiempo debe esperar el núcleo antes de iniciar el hilo.
- **Execution Mode**: indica si el hilo está en modo de superusuario o modo usuario.

Funciones de utilidad:

- **Borrar un hilo**:
  k_thread_join()
- **Abortar un hilo**:
  k_thread_abort()
- **Suspender un hilo**:
  k_thread_suspend()
- **Delay de un hilo**:
  k_sleep()



Para crear un hilo se debe definir el área de stack, el thread control block, y luego llamar a la función de creación.

Recursos:

- k_thread_create() : crea un hilo de manera dinámica.
- K_THREAD_DEFINE : crea un hilo de manera estática en tiempo de compilación.
- K_THREAD_STACK_DEFINE : macro que define un área de stack de forma estática.
- k_thread : define el thread control block.
- k_tid_t : define thread id.

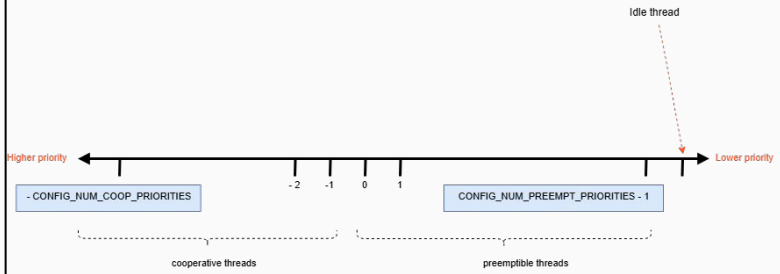
```c
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

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);
```

```c
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

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

K_THREAD_DEFINE(my_tid, MY_STACK_SIZE,
                my_entry_point, NULL, NULL, NULL,
                MY_PRIORITY, 0, 0);
```

PRIORIDADES:

La **prioridad** de un hilo es un **valor entero** y puede ser **negativa o no negativa**. Las prioridades numéricamente **más bajas prevalecen sobre las más altas**.

- Un **hilo cooperativo** tiene una prioridad negativa. Una vez que se convierte en el hilo actual, permanece como tal hasta que realiza una acción que lo deja fuera de servicio.
- Un **hilo preemptible** tiene una prioridad no negativa. Una vez que se convierte en el hilo actual, puede ser reemplazado en cualquier momento si un hilo cooperativo, o un hilo preemptible de mayor o igual prioridad, está listo.



OPCIONES:

El núcleo admite un pequeño conjunto de opciones de hilo que permiten que un hilo reciba un tratamiento especial en circunstancias específicas. El conjunto de opciones asociadas a un hilo se especifica al generarlo.

Un hilo que no requiere ninguna opción tiene un valor de opción de cero. Un hilo que requiere una opción la específica por nombre o utilizando el carácter '|' como separador si se necesitan varias opciones (es decir, combinar opciones mediante el operador OR bit a bit).

- K_ESSENTIAL: Esto indica al núcleo que trate la terminación o aportación del hilo como un error fatal del sistema.
- K_SSE_REGS
- K_FP_REGS: Esta opción indica que el hilo utiliza los registros de coma flotante de la CPU. Esto indica al núcleo que tome medidas adicionales para guardar y restaurar el contenido de estos registros al programar el hilo.
- K_USER
- K_INHERIT_PERMS

Semáforo

Un semáforo tiene las siguientes propiedades clave:

- Un **recuento** que indica el número de veces que se puede usar el semáforo. Un recuento de cero indica que el semáforo no está disponible.
- Un **límite** que indica el valor máximo que puede alcanzar el recuento del semáforo.

Semaphore - Zephyr

Teoria/Ejemplo - Github

- Un **semáforo** puede ser proporcionado por un **hilo** o un **ISR**. Al proporcionar el **semáforo**, se **incrementa su conteo**, a menos que este ya sea igual al **límite**.

- Un **hilo** puede **tomar un semáforo**. Tomarlo **disminuye su contador**, a menos que no esté disponible (es decir, esté en cero). Cuando un semáforo no está disponible, un hilo puede esperar a que se le proporcione. Cualquier número de hilos puede esperar simultáneamente en un semáforo no disponible. Cuando se proporciona el semáforo, lo toma el hilo con mayor prioridad que haya esperado más tiempo.

Recursos:

- k_sem : variable de tipo semáforo.
- k_sem_init() : define un semáforo de forma dinámica en tiempo de ejecución.
- K_SEM_DEFINE : define un semáforo de forma estática en tiempo de compilación.

```
struct k_sem my_sem;

k_sem_init(&my_sem, 0, 1);
```

```
K_SEM_DEFINE(my_sem, 0, 1);
```

Give semáforo:

Recursos:

- k_sem_give()

```
void input_data_interrupt_handler(void *arg)
{
    /* notify thread that data is available */
    k_sem_give(&my_sem);

    ...
}
```

Take semáforo:

Recursos:

- k_sem_take()

```
void consumer_thread(void)
{
    ...

    if (k_sem_take(&my_sem, K_MSEC(50)) != 0) {
        printk("Input data not available!");
    } else {
        /* fetch available data */
        ...
    }
    ...
}
```

Timers

Un temporizador es un objeto del núcleo que mide el paso del tiempo utilizando el reloj del sistema del núcleo. Cuando se alcanza el límite de tiempo especificado, un temporizador puede realizar una acción definida por la aplicación o simplemente registrar la expiración y esperar a que la aplicación lea su estado.

Timers - Zephyr

Teoria/Ejemplo - Github

Un temporizador tiene las siguientes propiedades clave:

- Una **duración** que especifica el intervalo de tiempo antes de que el temporizador expire por primera vez.
- Un **período** que especifica el intervalo de tiempo entre todas las expiraciones del temporizador después de la primera, también conocido como **k_timeout_t**. Debe ser **positivo**. Un **período** de **K_NO_WAIT** (es decir, cero) o **K_FOREVER** significa que el temporizador es de **un solo uso** y se **detiene** tras una única expiración. (Por ejemplo, si un temporizador se inicia con una duración de 200 y un período de 75, expirará primero después de 200 ms y, a partir de entonces, cada 75 ms).
- Una **función de expiración** que se ejecuta cada vez que expira el temporizador.
- Una **función de detención** que se ejecuta si el temporizador se detiene prematuramente durante su ejecución.
- Un **valor de estado** que indica cuántas veces ha expirado el temporizador desde que se leyó el valor de estado por última vez.

Recursos:

- k_timer
- k_timer_init()
- K_TIMER_DEFINE
- k_timer_start()
- k_timer_stop()

```
struct k_timer my_timer;
extern void my_expiry_function(struct k_timer *timer_id);

k_timer_init(&my_timer, my_expiry_function, NULL);
```

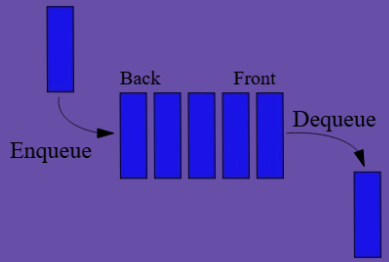
```
K_TIMER_DEFINE(my_timer, my_expiry_function, NULL);
```

```
void my_timer_handler(struct k_timer *dummy)
{
    k_work_submit(&my_work);
}

K_TIMER_DEFINE(my_timer, my_timer_handler, NULL);

...

/* start a periodic timer that expires once every second */
k_timer_start(&my_timer, K_SECONDS(1), K_SECONDS(1));
```

Cola de datos (FIFOs o LIFOs)

Una cola en Zephyr es un objeto del kernel que implementa una cola tradicional, permitiendo que los subprocesos y los ISR agreguen y eliminen elementos de datos de cualquier tamaño. La cola es similar a un FIFO y sirve como implementación subyacente tanto para k_fifo como para k_lifo . Para más información sobre su uso, consulte k_fifo .

FIFOs - Zephyr



Ejemplo - Github

Una **cola de mensajes** tiene las siguientes **propiedades** claves:

- Un **ring buffer** de elementos de datos que se han enviado pero aún no se han recibido.
- Un **tamaño de elemento de datos** , medido en bytes.
- Una **cantidad máxima** de elementos de datos que se pueden poner en cola en el **ring buffer**.

Un **hilo** o un **ISR** pueden **enviar** un dato a una **cola de mensajes**. El dato al que apunta el hilo emisor se copia a un hilo en espera, si existe; de lo contrario, se copia al **buffer circular** de la cola de mensajes, si hay espacio disponible. En cualquier caso, el tamaño del área de datos enviada debe ser igual al tamaño del dato de la cola de mensajes.

Un **hilo** puede **recibir** un dato de una **cola de mensajes**. El dato se copia al área especificada por el **hilo receptor**; el tamaño del área receptora debe ser igual al tamaño del dato de la cola de mensajes.

Recursos:

- k_queue
- k_queue_init() : inicializa una cola de datos de forma dinámica en tiempo de ejecución.
- K_QUEUE_DEFINE : define una cola de datos en tiempo de compilación.
- k_queue_append() : añade un elemento a la cola de datos al final.
- k_queue_get() : obtiene un elemento y lo elimina.
- k_queue_insert() : inserta un elemento en una ubicación específica.
- k_queue_is_empty() : determina si la cola de datos no está vacía.

```
K_QUEUE_DEFINE(queueMessage);
```

```
struct data_item_t {
    void* queue_reserved;   // Reservado para uso interno de la cola
    int value;              // Dato que se transmite
};
```

```
void threadProducer(void *p1, void *p2, void *p3)
{
    struct data_item_t txData;

    for (;;) {
        txData.value = k_uptime_get_32();  // Obtiene el tiempo actual
        k_queue_append(&queueMessage, &txData);  // Envía a la cola
        k_msleep(1000);  // Espera 1 segundo
    }
}
```

```
void threadConsumer(void *p1, void *p2, void *p3)
{
    for (;;) {
        struct data_item_t *rxData = k_queue_get(&queueMessage, K_FOREVER);

        if (rxData != NULL)
            printk("[CONSUMER] Dato recibido: %d\n", rxData->value);
    }
}
```

Mensaje de Registro (Log)

La API de registro proporciona una interfaz común para procesar los mensajes emitidos por los desarrolladores. Los mensajes se transmiten a través de un frontend y luego son procesados por backends activos. Se pueden usar frontends y backends personalizados si es necesario.

Logging - Zephyr

El sistema dispone de cuatro niveles de gravedad: **error**, **advertencia**, **información** y **depuración**. Para cada nivel de gravedad, la API de registro cuenta con un conjunto de macros dedicadas. La API de registro también incluye macros para el registro de datos.

Resumen de las funciones de registro:

- Filtrado en tiempo de compilación a nivel de módulo.
- API dedicada para volcado de datos.
- Soporte para registrar variables de punto flotante y argumentos largos.
- Soporte de Printk: el mensaje de Printk se puede redirigir al registro.

Ventajas principales sobre Printk:

1. **Menor impacto en tiempo real:**
   a. Los logs se manejan en background por lo que las interrupciones y tareas no se ven afectadas.
   b. printk() puede generar jitter o bloqueos si se usa dentro de ISR o contextos críticos.
2. **Clasificación por nivel y módulo:**
   a. Podés elegir qué mostrar sin cambiar el código.
   b. Ejemplo: mostrar sólo errores del ADC, pero mensajes informativos del sensor.
3. **Soporte para múltiples backends:**
   a. UART, RTT, shell, flash, red, RAM circular buffer, etc.
   b. Ideal para proyectos donde la salida serie no está siempre disponible.
4. **Control dinámico:**
   a. Podés cambiar el nivel de log en tiempo de ejecución desde la shell.

```
uart:~$ log enable adc_module
uart:~$ log set-level adc_module 4
```

5. **Formato y metadatos automáticos:**
   a. Incluye timestamp, hilo, nivel, y módulo automáticamente.
   b. Ejemplo de salida:

```
[00:12:34.123,456] <inf> main: ADC value: 512
```

Recursos:

- LOG_X para mensajes estándar similares a printf, por ejemplo *LOG_ERR*.
- LOG_ERR
- LOG_WRN
- LOG_DGB
- LOG_MODULE_REGISTER
- LOG_LEVEL_SET

KConfig:

- CONFIG_LOG=y
- CONFIG_LOG_DEFAULT_LEVEL=3
- CONFIG_LOG_BACKEND_UART=y

```
LOG_MODULE_REGISTER(main, LOG_LEVEL_DBG);

void main(void) {
    LOG_INF("Inicio del sistema");
    LOG_WRN("Temperatura fuera de rango");
    LOG_ERR("Error al inicializar el sensor");
}
```

Ejemplo de Uso

KConfig y DeviceTree

KConfig — Configuración del Sistema:

Sistema jerárquico de configuración (inspirado en Linux) que permite habilitar, deshabilitar o ajustar opciones del kernel, drivers y librerías de Zephyr sin modificar el código fuente.

DeviceTree — Descripción del Hardware:

Es una representación estructurada del hardware, independiente del código. Permite describir qué periféricos existen, cómo están conectados y qué recursos usan (pines, interrupciones, direcciones, etc.).

KConfig

- Se define mediante archivos Kconfig y prj.conf.
- Permite configurar:
  - Frecuencia del tick del kernel
  - Tamaño de pila de los hilos
  - Drivers habilitados (ADC, UART, GPIO, etc.)
  - Nivel de logging y optimizaciones
  - Entre otros

```
# prj.conf
CONFIG_GPIO=y
CONFIG_ADC=y
CONFIG_LOG=y
CONFIG_MAIN_STACK_SIZE=1024
```

```
&adc0 {
    status = "okay";
    channel@0 {
        reg = <0>;
        zephyr,gain = "ADC_GAIN_1";
        zephyr,reference = "ADC_REF_INTERNAL";
    };
};
```

DeviceTree

Se tienen dos archivos “.dts” que son las configuraciones base del SoC principal y luego los archivos “.overlay” que configuraciones o modificaciones propias del proyecto. Un sufijo “.dtsi” indica un archivo DTS que se incluye en otro archivo DTS. Esto puede verse en el ejemplo del ADC cargado en el repositorio.

Ventajas:

- No es necesario modificar el código para cambiar hardware.
- Evita errores de configuración manual.
- Facilita la portabilidad entre placas.
- Nos permite configurar el módulo directamente desde el archivo .dts para luego exportarlo al .c y cargar la configuración.

Las dos principales herramientas de compilación de Zephyr son el KConfig y el Device Tree.
Resumiendo lo antes mencionado podemos decir:

- El **device tree** se utiliza para describir el **hardware** y su **configuración en el momento del arranque**.
- **Kconfig** se utiliza para **describir qué características del software se integrarán** en la imagen final y su configuración.

Referencia

Gracias

SISTEMAS EMBEBIDOS AVANZADOS II