Zephyr RTOS GPIO driver system

```
#include
#include
const struct device *get_gpio_device(const char *label) {
return device_get_binding(label);
}
```

Zephyr Device Model

Zephyr uses a device model where hardware peripherals (like GPIOs, UARTs, etc.) are represented as struct device objects. These are registered during system initialization.

device_get_binding(label)

This function looks up a device by its **label**, which is defined in the **Device Tree** or board configuration files. For example, "GPIO_0" or "GPIOA".

- It returns a pointer to the struct device representing the GPIO controller.
- If the label is incorrect or the device isn't initialized, it returns NULL.

GPIO Driver Interaction

Once you have the device pointer, you can use it with GPIO driver APIs like:

gpio_pin_configure(dev, pin, GPIO_OUTPUT);

gpio_pin_set(dev, pin, value);

These functions use the device pointer to interact with the actual hardware registers via the GPIO driver.

gpio_setup() → get_gpio_device(label) → device_get_binding(label)

→ DEVICE_DT_DEFINE(...) → gpio_emul_driver (via DEVICE_API)

- **get_gpio_device()** is a helper to fetch the GPIO device.
- It enables your code to be **hardware-agnostic**, relying on labels from the Device Tree.
- Once you have the device, you can **configure pins**, **read/write values**, and **handle interrupts** using Zephyr's GPIO API.

1. Device Tree Source (DTS) Path for QEMU x86

```
vi ../../../boards/qemu/x86/qemu_x86.dts
```

```
    gpio0: gpio_sim {
        compatible = "zephyr,gpio-emul";
        gpio-controller;
        #gpio-cells = <2>;
        ngpios = <32>;
        label = "GPIO_0";
    };
```

```
static void gpio_setup(void *fixture){
gpio_dev = get_gpio_device(GPIO_LABEL);
zassert_not_null(gpio_dev, "Failed to get GPIO device");
}
```

- This function is part of a test setup.
- It calls **get_gpio_device(**) with a label like "GPIO_0".
-
- This wraps device_get_binding(), which looks up a device by its label.

device_get_binding(label)

- This searches the global device list for a device with a matching label (from the Device Tree).
- If found, it returns a pointer to the struct device.

Device Tree + Driver Binding

- The label (e.g., "GPIO_0") is defined in the Device Tree (.dts or .overlay).
- The corresponding node is associated with a driver using DEVICE_DT_DEFINE() or DEVICE_DEFINE().

```
	DEVICE_DT_INST_DEFINE(_num, gpio_emul_init,			\
			    PM_DEVICE_DT_INST_GET(_num),		\
			    &gpio_emul_data_##_num,			\
			    &gpio_emul_config_##_num, POST_KERNEL,	\
			    CONFIG_GPIO_INIT_PRIORITY,			\
			    &gpio_emul_driver);
```

"../../../drivers/gpio/gpio_emul.c

```
static DEVICE_API(gpio, gpio_emul_driver) = {
	.pin_configure = gpio_emul_pin_configure,
#ifdef CONFIG_GPIO_GET_CONFIG
	.pin_get_config = gpio_emul_pin_get_config,
#endif
	.port_get_raw = gpio_emul_port_get_raw,
	.port_set_masked_raw = gpio_emul_port_set_masked_raw,
	.port_set_bits_raw = gpio_emul_port_set_bits_raw,
	.port_clear_bits_raw = gpio_emul_port_clear_bits_raw,
	.port_toggle_bits = gpio_emul_port_toggle_bits,
	.pin_interrupt_configure = gpio_emul_pin_interrupt_configure,
	.manage_callback = gpio_emul_manage_callback,
	.get_pending_int = gpio_emul_get_pending_int,
#ifdef CONFIG_GPIO_GET_DIRECTION
	.port_get_direction = gpio_emul_port_get_direction,
#endif /* CONFIG_GPIO_GET_DIRECTION */
```

- This struct defines the function pointers for the GPIO emulator's API.
- It is passed to DEVICE_DT_DEFINE() to register the device.

Final Mapping

- When device_get_binding("GPIO_0") is called, it returns a struct device whose .api field points to gpio_emul_driver.
- So when you later call gpio_pin_configure(gpio_dev, ...), it internally calls gpio_emul_pin_configure().

========

How gpio_pin_set() Maps to the Emulator Driver

gpio_pin_set(gpio_dev, TEST_PIN, 1);

This is a public Zephyr API defined in include/zephyr/drivers/gpio.h.

Device API Dispatch

Internally, this calls the function pointer from the device's driver API:

gpio_dev->api->port_set_bits_raw(...)

This pointer is set to the emulator's implementation in gpio_emul_driver:

```
static const struct gpio_driver_api gpio_emul_driver = {
...
.port_set_bits_raw = gpio_emul_port_set_bits_raw,
.port_clear_bits_raw = gpio_emul_port_clear_bits_raw,
...
};
```

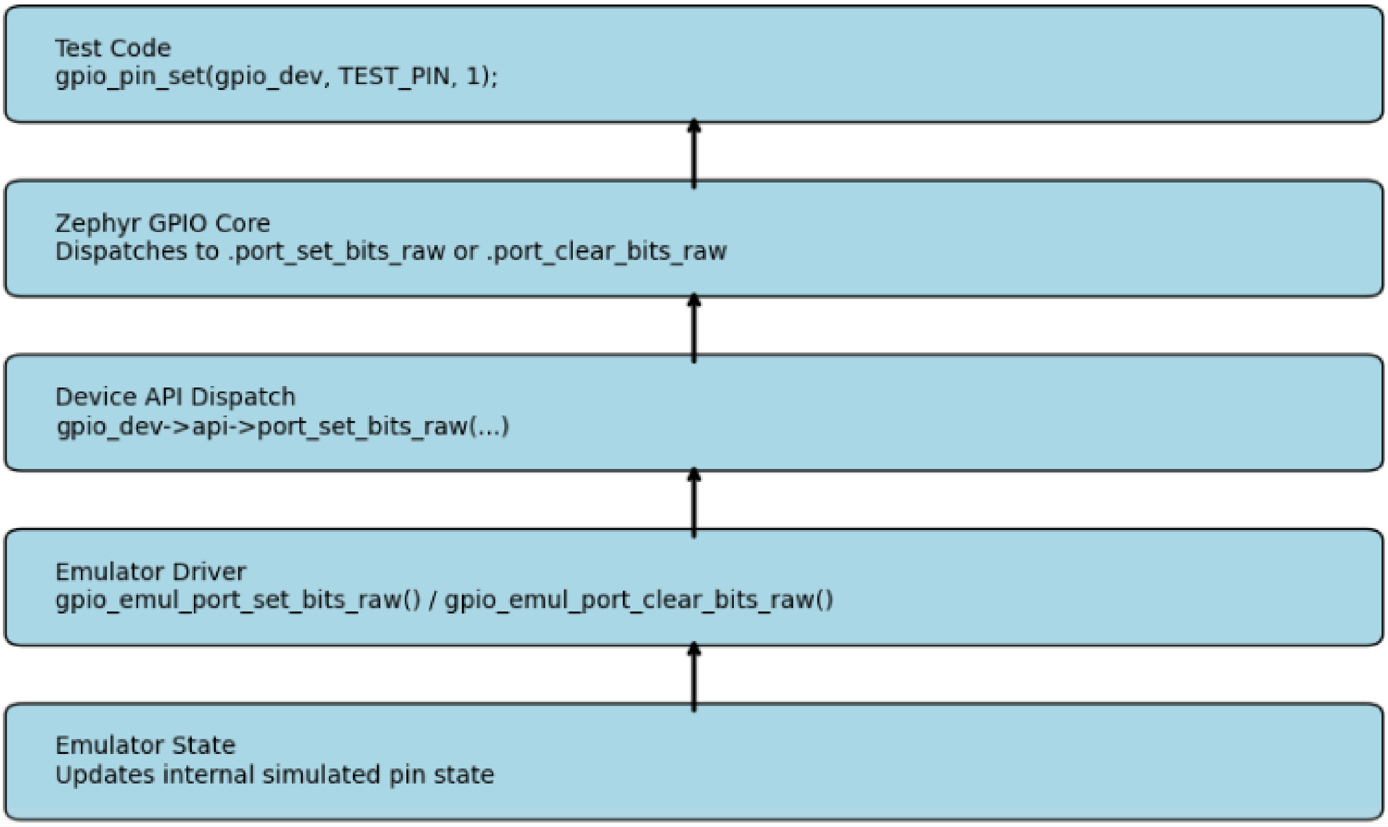

Diagram Layers Explained

1. **Test Code**
   Calls gpio_pin_set(gpio_dev, TEST_PIN, 1); in your test suite.

2. **Zephyr GPIO Core**
   Routes the call to the appropriate function pointer in the GPIO driver API.
3. **Device API Dispatch**
   Internally calls gpio_dev->api->port_set_bits_raw(...) or port_clear_bits_raw(...).
4. **Emulator Driver**
   Executes gpio_emul_port_set_bits_raw() or gpio_emul_port_clear_bits_raw() from gpio_emul.c.
5. **Emulator State**
   Updates the internal simulated pin state (e.g., a bitfield or array) to reflect the pin value.

`==========================================================================`

/zephyrproject/zephyr/samples/Zaphyr_app/my_zephry_app_threadPrio/

my_zephry_app_threadPrio

Threads Overview

✅ Thread 1: Sleeping & Suspended

- **Function**: thread1_fn
- **Behavior**:
  - Sleeps for 2 seconds repeatedly using k_sleep(), which puts it in the **sleeping state**.
  - Can be **suspended** by Thread 3 using k_thread_suspend(), which halts its execution regardless of its sleep state.
  - Later **resumed** by Thread 3 using k_thread_resume().

✅ Thread 2: Pending

- **Function**: thread2_fn
- **Behavior**:
  - Waits indefinitely on a semaphore using k_sem_take(&sync_sem, K_FOREVER).
  - This puts it in the **pending state** until another thread gives the semaphore.
  - Once the semaphore is given by Thread 3, it prints a message and loops back to wait again.

✅ Thread 3: Running & Controller

- **Function**: thread3_fn
- **Behavior**:
  - Runs continuously and **controls the other two threads**.
  - Suspends Thread 1, waits 3 seconds, then resumes it.
  - Gives the semaphore to Thread 2 to unblock it.
  - Sleeps for 5 seconds before repeating.
- This thread is always **running** when active and demonstrates how a thread can manage others.

🧠 Thread Priorities

- **Thread 1**: Priority 1 (highest)
- **Thread 3**: Priority 4
- **Thread 2**: Priority 5 (lowest)

In Zephyr, **lower numbers mean higher priority**. So Thread 1 will preempt others unless it's sleeping or suspended.

🔄 Execution Flow Summary

1. **Startup**:
   - All three threads are created and start running.
2. **Thread 1**:
   - Sleeps for 2 seconds → enters **sleeping state**.
   - May be **suspended** by Thread 3 during this time.
3. **Thread 2**:
   - Waits on a semaphore → enters **pending state**.
   - Remains blocked until Thread 3 gives the semaphore.
4. **Thread 3**:
   - Suspends Thread 1 → Thread 1 enters **suspended state**.
   - Sleeps for 3 seconds.
   - Resumes Thread 1 → Thread 1 can now run again.
   - Gives semaphore to Thread 2 → Thread 2 wakes up.
   - Sleeps for 5 seconds and repeats the cycle.

📊 Thread States Demonstrated

Thread	State(s) Demonstrated	How?
Thread 1	Sleeping, Suspended	k_sleep(), k_thread_suspend()
Thread 2	Pending	k_sem_take()
Thread 3	Running	Actively controls other threads

Core Concepts of Zephyr RTOS

Zephyr is a **lightweight, scalable, real-time operating system (RTOS)** designed for embedded systems. It supports multiple architectures and is highly modular, allowing developers to include only the components they need.

1. Kernel Architecture

- **Small-footprint kernel**: Designed for resource-constrained devices.
- **Supports multiple CPU architectures**: ARM, x86, RISC-V, ARC, MIPS, and more
-

2. Threading Model

Zephyr supports:

- **Cooperative threads**: Run until they yield or block.
- **Preemptive threads**: Can be interrupted by higher-priority threads.
- **Round-robin scheduling**: Among threads of equal priority.
- **POSIX pthreads API**: Optional compatibility layer.

3. Scheduling Algorithms

- **Cooperative and Preemptive Scheduling**
- **Earliest Deadline First (EDF)**
- **Meta IRQ scheduling**: For deferred interrupt handling.
- **Timeslicing**: Among equal-priority preemptible threads.

4. Memory Management

- **Heap and slab allocators**
- **Thread stacks**: Configurable per thread.
- **Custom memory regions**: For advanced use cases.

5. Inter-thread Communication

- **Semaphores**: Binary and counting.
- **Mutexes**: For mutual exclusion.
- **Message Queues and Pipes**: For data passing.

6. Power Management

- **System Power Management**: Application-defined policies.
- **Device Power Management**: Driver-level control.

7. Modularity

- Zephyr is **highly configurable** via Kconfig and device tree.

- Developers can enable/disable features to optimize footprint.

🧵 System Threads in Zephyr

System threads are **automatically spawned by the kernel** during system initialization. They are essential for basic OS operation.

Types of System Threads

1. Main Thread

- **Purpose**: Executes kernel initialization and then calls the user-defined main() function.
- **Priority**:
  - Highest preemptible priority (0) if preemption is enabled.
  - Lowest cooperative priority (-1) if not.
- **Behavior**:
  - If main() is defined and returns normally, the thread terminates without error.
  - If main() is missing or the thread aborts, a **fatal error** is raised.

2. Idle Thread

- **Purpose**: Runs when no other thread is ready.
- **Behavior**:
  - May invoke power-saving routines.
  - Otherwise, runs a “do-nothing” loop.
- **Priority**: Always the **lowest**.
- **Essential**: Cannot terminate; if it aborts, a fatal error occurs.

3. Workqueue Threads

- Spawned if the **system workqueue** is enabled.
- Handle deferred work items submitted by the application.

Writing a main() Function

Example: