

1. Definitions

Here are the important definitions and what they refer to in the Zephyr PM subsystem:

1. **Power Management (PM):** In Zephyr, this refers to the infrastructure that allows the OS, devices, and drivers to reduce power consumption by transitioning to lower-power states when idle, reducing clocking, gating power, suspending devices or the CPU/System etc.
 - **System Power Management:** The PM subsystem that deals with the entire system/SoC/CPU entering low-power states (sleep, standby, etc) when idle. It is concerned with transitions of the CPU/SoC.
 - **Device Power Management:** The part of PM that deals with devices (drivers/peripherals) being suspended, resumed, either when system enters low-power or independently of it. Enables devices to go into low power when idle.
 - **Device Runtime Power Management:** A variant of device PM where devices can autonomously (from the driver/subsystem usage) suspend/resume while the system (CPU) is still active — i.e., devices idle while CPU might be active.
 - **Power Domain:** A grouping of resources (devices/peripherals) that share a power domain (i.e., can be turned off together). The PM subsystem supports handling these domains.
 - **Power State / Device Power State:** A state that represents how much of the device's context is retained, whether clocks/power are gated, etc. Eg: ACTIVE, SUSPENDED, OFF.
 - **Usage Count / Reference Counting (for runtime PM):** In device runtime PM, usage count is maintained so that when no one is using a device it can be suspended; when someone needs it, it's resumed.
 - **Wake-up Source:** A capability of a device to wake the system (or other devices) from a low-power state (e.g., a GPIO interrupt). The PM subsystem tracks this in devicetree via wakeup-source property.

2. Architecture / Key Components

Here's how the PM subsystem is architected in Zephyr, and the major components.

Architecture Overview

- The PM subsystem in Zephyr is designed to be architecture and SoC independent — meaning it provides an abstraction so that the rest of the OS/application doesn't need to know the specifics of each SoC's power modes.
- The architecture separates **core PM infrastructure** (common code) from **SoC/architecture-specific implementations** (hardware backend).
- The subsystem is organised into two main categories: *System PM* and *Device PM*. Device PM itself has sub-models (system-managed vs runtime).

Key Components

Here are major components you should know:

1. System PM Engine/Policy

- Decides what system/SoC/CPU low-power state to enter when the system is idle. Eg: “sleep”, “deep sleep”, “shutdown” etc.
- Provides APIs like `pm_power_state_set()`, `pm_constraint_set()`, `pm_notifier_register()` etc (depending on version) that let other modules influence or hook into power-state transitions.

2. Device PM Infrastructure

- Tracks devices, their states, their capabilities (can they suspend, resume, are they wakeup capable?).
- Defines device power states (ACTIVE, LOW_POWER, SUSPENDED, OFF).
- Provides APIs for device drivers to implement their suspend/resume logic. Eg: `pm_device_state_set()`, `pm_device_state_get()` etc.
- Manages dependencies: if a device depends on another (bus driver, parent device) then power-domain or runtime dependencies must be handled by driver.

3. Device Runtime PM Engine

- Maintains a usage count per device. Drivers call `pm_device_runtime_get()` when they need the device, `pm_device_runtime_put()` when done. When usage count goes to zero, the engine may suspend the device.
- Supports asynchronous suspend/resume as well (so that driver suspend does not block system).
- Integrates with power domains so that suspending a device may also release the domain (if runtime PM of power domain enabled).

4. Power Domains

- A set of devices/peripherals grouped logically under a shared power supply/clock domain. Suspending the domain may require suspending all its devices. Drivers/devicetree may declare power-domains property.
- Provides guidelines for implementation in Zephyr.

1. Devicetree / Kconfig Bindings

- The PM subsystem uses devicetree properties (e.g., `zephyr,pm-device-runtime-auto`, `wakeup-source`, `power-states`) and Kconfig options (`CONFIG_PM`, `CONFIG_PM_DEVICE`, `CONFIG_PM_DEVICE_RUNTIME`) to enable or configure features.

2. Busy Status Indication

- For system PM, to avoid suspending while a device is busy, Zephyr provides APIs: `device_busy_set()`, `device_busy_clear()`, `device_any_busy_check()` etc.

Workflow / Interaction

- At run-time: When the system becomes idle, the System PM engine may evaluate constraints (from devices, subsystems) and enter a low-power state.
- Devices may be individually suspended/resumed via Device PM or Runtime PM.
- Driver developers implement suspend/resume callbacks (`pm_device_action_cb_t`) that handle hardware-specific shutdown of clocks/power, and restore on resume.
- Devicetree properties define which devices support runtime PM auto-enable or wakeup.
- Power domains may be requested and released as part of runtime PM operations.

- Subsystems or apps may hint at power constraints (e.g., “I need CPU active for X ms”) via policy APIs.

3. Workflows

Here are typical workflows illustrating how things happen in Zephyr PM:

3.1 System PM Workflow

1. CPU/Kernel enters idle.
2. The PM subsystem determines an appropriate low-power state (based on idle time estimate, constraints from devices or subsystems).
3. Before entering the low-power state, all relevant devices may need to be suspended (if system-managed device PM is enabled).
4. The system executes the low-power entry (SoC-specific).
5. On wakeup (interrupt, wake-source), the system resumes. Devices are resumed (if previously suspended).
6. Kernel returns to normal scheduling.

3.2 Device Runtime PM Workflow

1. A device driver or subsystem calls `pm_device_runtime_get(dev)`.
 - If the usage count was zero (device was suspended), then this call will trigger a *resume* operation for that device (and potentially its power domain).
2. The driver performs its work (e.g., transaction with peripheral).
3. The driver calls `pm_device_runtime_put(dev)`.
 - This decrements the usage count. If it becomes zero, the subsystem may suspend the device (either synchronously or asynchronously depending on configuration).
4. Meanwhile, the system (CPU) remains active; other devices may still be running.
5. If the device belongs to a power domain, the domain may be released when all child devices are idle, if configured.

3.3 Device PM (System-managed) Workflow

1. The system decides to enter a low-power state and inspects if devices are busy or flagged as always-active.
2. Suspend callbacks are invoked on devices (in init order) to transition them into low-power state (SUSPENDED or OFF).
3. After wake-up, devices are resumed in reverse order.
4. Note: if a device refuses suspend (returns -EBUSY), then system may refuse to enter that state.

4. Use Cases

Here are some typical use-cases for Zephyr’s PM subsystem:

- **Battery-powered IoT devices:** A sensor node that wakes up periodically, reads sensors, sends data via BLE, then sleeps. Here runtime PM helps to suspend unused peripherals

(e.g., the radio, sensor bus) while system PM puts CPU into deep sleep when idle.

- **Wearables / Smartwatch:** In the “Designing for Low Power Using Zephyr® Project: A Smartwatch Example” from NXP Semiconductor, Zephyr’s PM subsystem is used plus SoC features to optimize battery life.
- **Always-on edge devices with peripherals that aren’t always in use:** e.g., a network gateway where the CPU stays awake but some devices (unused serial ports, sensors) can be suspended using device runtime PM to save power.
- **Complex SoC with multiple power domains:** Some MCUs have multiple power domains; Zephyr’s power domain support allows turning off entire domains when not in use. Example: Silicon Labs Series 2 devices.
- **Low power wireless nodes:** A BLE node may sleep, wake on interrupt (GPIO wakeup), turn on radio, send/receive, then suspend radio and other peripherals again.

5. Advantages

Here are the benefits of using Zephyr’s PM subsystem:

- **Hardware/architecture-agnostic abstraction:** Developers write driver or application code without tailoring to each SoC’s power modes (to some extent).
- **Granular device-level control:** Via runtime PM, devices can individually suspend/resume, enabling power savings even when CPU is running.
- **Support for complex power domains:** Useful in modern MCUs which have multiple domains and autonomous peripherals.
- **Flexible policy & constraints:** Applications or subsystems can impose constraints (e.g., “Don’t sleep beyond X until operation Y completes”).
- **Improved battery life:** By suspending unused devices, gating clocks/power, and using deep sleep states.
- **Wake-up support:** Devices can be configured as wake-sources, enabling low-power states while still being responsive to external events.
- **Driver-based integration:** Drivers can implement power-state transitions themselves, leading to optimal hardware use.

7. Key Components / Concepts Summarised

Here is a list of the key components/concepts you should remember, with brief descriptions:

Component	Description
CONFIG_PM	Kconfig option to enable system power management.
CONFIG_PM_DEVICE	Kconfig option to enable device power management.
CONFIG_PM_DEVICE_RUNTIME	Kconfig option to enable device runtime PM.
Device power states (pm_device_state)	Enumerated states like PM_DEVICE_STATE_ACTIVE, SUSPENDED, OFF.
Device action callback (pm_device_action_cb_t)	The driver callback invoked when a PM action (SUSPEND/RESUME/TURN_OFF) is requested.
Runtime PM APIs	e.g., pm_device_runtime_get(), pm_device_runtime_put(), pm_device_runtime_put_async().
System PM policy APIs	e.g., pm_constraint_set(), pm_notifier_register(), pm_power_state_set().
Busy status APIs	e.g., device_busy_set(), device_busy_clear(), to prevent entering low-power when a device is busy.
Power Domain support	Enabling grouping of devices under a power domain and managing domain transitions. (Bridle)
Devicetree properties	e.g., zephyr,pm-device-runtime-auto, wakeup-source, power-state nodes.

8. Real-World Examples

Here are a few real-world usage scenarios:

- **Smartwatch reference example (NXP + Zephyr):** As mentioned above, NXP presented a wearable example built on Zephyr, leveraging its PM subsystem to maximise battery life in a UI-rich application. ([NXP Semiconductors](#))
- **Silicon Labs Series 2 devices:** Their Zephyr implementation uses runtime PM for peripherals, and uses the PM subsystem in Zephyr to integrate with their “Energy Modes”.

Eg: on Series 2, device runtime PM actions (SUSPEND/RESUME) handle peripheral clocks, pinctrl sleep state, etc. ([Silicon Labs Docs](#))

- **Embedded sensor node:** Suppose you have a BLE sensor node: When idle, the radio and sensor bus are suspended via runtime PM, the CPU enters deep sleep via system PM, and a GPIO interrupt wakes the system for a measurement cycle.
- **Network gateway with intermittent usage:** A gateway might have many peripherals (USB, sensors, display) but only some active at any time; devices not in use can be suspended to save power while the CPU stays active for network management.

9. Advantages & Limitations (Recap)

Advantages

- Flexible and granular power control (per-device, per-domain, system)
- Hardware-agnostic abstraction simplifies porting
- Enhanced battery life especially in embedded/IoT systems
- Supports wake-sources so low power doesn't mean dead-end
- Integration with devicetree/Kconfig makes configuration easier

10. Interview Questions & Answers

Q1. What is the difference between system power management and device power management in Zephyr?

A1.

System Power Management refers to putting the CPU/SoC into a low-power sleep/idle state when the system is idle. Device Power Management refers to suspending/resuming individual devices/peripherals (clocks off, power gating) either in response to system state changes or independently (runtime). The former affects the whole system/CPU, while the latter affects only specific peripherals.

Q2. What is device runtime power management and when would you use it?

A2.

Device Runtime PM allows devices to be suspended or resumed based on their usage, independent of the system's sleep state. Usage is tracked via a reference count: when no subsystem uses the device, it can be suspended; when an operation needs it, it is resumed. It is useful in cases where the CPU remains active but many devices are idle (e.g., network gateway, sensor hub).

Q3. What are the main device power states in Zephyr's device PM infrastructure, and what do they signify?

A3.

Typical states include:

- `PM_DEVICE_STATE_ACTIVE` — device is active, all context retained.

- `PM_DEVICE_STATE_LOW_POWER` — (some implementations) context retained but device clocks/power reduced.
- `PM_DEVICE_STATE_SUSPENDED` — most context lost, driver must restore.
- `PM_DEVICE_STATE_OFF` — device fully powered off, context lost, full reinitialization needed.

Q4. How does Zephyr handle device dependencies in device runtime PM (or what is a limitation in that area)?

A4.

In current Zephyr versions, device runtime PM **does not automatically manage dependencies** between devices and their parent/child devices or power domains. This means that if a sensor uses a bus device, the bus may get suspended/resumed on each transaction instead of staying active for a batch of transactions. Driver authors should manage dependencies manually.

Q5. What are wake-sources and why are they important in PM?

A5.

Wake-sources are hardware sources (e.g., `GPIO`, timer, RTC, external interrupt) that can wake the system (CPU) or device from a low-power state. They are critical because power management lowers power by suspending devices or entering sleep, but you often still need responsiveness (e.g., a button press or sensor event). In Zephyr, wake-capability is indicated in devicetree via `wakeup-source` property.

Q6. What are some of the configuration options you must enable in Zephyr to use the PM subsystem?

A6.

Key Kconfig options include:

- `CONFIG_PM` to enable system power management.
- `CONFIG_PM_DEVICE` to enable device power management.
- `CONFIG_PM_DEVICE_RUNTIME` to enable device runtime PM.

Also, devicetree properties like `zephyr,pm-device-runtime-auto` for automatic enabling of runtime PM on a device.

Q7. Describe the workflow of a device driver implementing runtime PM in Zephyr.

A7.

At device init: the driver may call `pm_device_runtime_enable(dev)` (or devicetree auto-enable) to enable runtime PM. During operation: when the driver or subsystem needs the device, it calls `pm_device_runtime_get(dev)` which increases a usage count and if the device was suspended will resume it. After usage is done, call `pm_device_runtime_put(dev)` (or `pm_device_runtime_put_async(dev, ...)`) which decreases usage count; if count goes to zero, then the subsystem may suspend the device (either synchronously or asynchronously). The driver must implement the suspend/resume logic via its action callback.

Q8. What are some of the advantages of using Zephyr's PM subsystem in embedded/IoT devices?

A8.

Advantages include: **better battery life** (by suspending unused devices and entering deep sleep), **more efficient use of idle time**, ability to respond to external events while asleep (via wake-sources), modularity and portability (because PM architecture is hardware-agnostic), and ability to control devices individually (granular control).

Q9. What are the limitations you should be aware of?

A9.

Limitations include: the need for correct driver implementation (suspend/resume logic, dependencies) which adds complexity; the fact that device dependency management in runtime PM is not fully automated; latency of suspend/resume may impact responsiveness; benefit is limited if CPU remains busy; hardware may not support fine-grained domains or autonomous peripherals; and configuration and wake-source setup needs careful design.

Q10. If you have a bus device that is used by multiple sensor devices, how would you avoid the bus being suspended/resumed too frequently in runtime PM?

A10.

Because runtime PM does not automatically manage dependencies, one approach is to manually manage usage counts for the bus device: each sensor driver that uses the bus should call `pm_device_runtime_get(bus_dev)` when starting and `pm_device_runtime_put(bus_dev)` when done, so that the bus stays active across multiple sensor operations. Alternatively, you might disable runtime PM for the bus (so it remains active) if frequent use is expected, and let the sensors suspend individually. Profiling may help determine the best approach.

11. Summary

In summary:

The Zephyr PM subsystem provides a **layered**, flexible means to manage power at system, device and runtime levels. With correct driver and application support, it enables embedded/IoT devices to reduce power consumption while maintaining responsiveness and functionality. The major work is in writing drivers and configuring wake-sources, domains and policies correctly. For battery-constrained systems or IoT endpoints this is a crucial capability. Knowing its architecture, key APIs, workflows and limitations makes you well-equipped to implement or evaluate PM in Zephyr-based systems.

Here's a deep dive into the core concepts from the **"Overview"** section of the Zephyr RTOS Power Management (PM) subsystem (per the link you shared:

Definitions

Here are the important terms and what they mean in this context:

- **Power Management (PM) subsystem:** In Zephyr, this is the framework that allows the system (CPU/SoC) and devices/peripherals to reduce power consumption by entering lower power states (sleep, idle, suspend) or by turning off/suspending devices when they're not in use.
- **Architecture / SoC independence:** The PM subsystem is designed so that the bulk of the PM logic is generic, and the architecture/SoC-specific parts are abstracted away. In Zephyr's words: "The interfaces and APIs ... are designed to be architecture and SOC independent."
- **System Power Management:** This refers to putting the CPU/SoC (and possibly many peripherals) into low power states when the system is idle. It is one category of PM in Zephyr.
- **Device Power Management:** This refers to managing the power state of individual devices (drivers/peripherals) – **suspending/resuming them, turning them off,** etc. It is another category of PM in Zephyr.
- **Power state / device power state:** A defined state that reflects how much of the device/system context is retained, what clocks/power are running, etc. For example, a device may be "active", or "suspended", or "off".
- **Wake-up source:** A hardware event (**interrupt, timer, GPIO, etc**) that can mark the system or device to wake up from a low-power state. Although more detailed in other sections, the overview section implies that one responsibility for system sleep is configuring wake-up events.
- **Policy / Constraints:** In system PM, a policy may decide which low-power state to use (based on idle time, residency, latency), and constraints may prevent deeper sleep if some devices or tasks require higher readiness. (More detailed later.)

Architecture / Key Components

Understanding how Zephyr's PM subsystem is structured is important.

High-level architecture

- Zephyr separates **core PM infrastructure** (generic code) from **SoC/architecture-specific implementations**. The overview states: **"The architecture and SOC independence is achieved by separating the core PM infrastructure and the SOC specific implementations."**
- The abstraction allows OS and application layers to use PM services without tying to particular hardware details. This design makes porting to new SoCs easier.
- Within PM, the features are classified into **System Power Management** and **Device Power Management**.

Key components (inferred from overview and expanded docs)

- **PM API Layer:** The public functions that drivers, the kernel, applications use to set constraints, notify busy states, request device suspend/resume, etc.
- **System PM Engine:** The part of the kernel idle thread that detects idle time, selects a power state, triggers the SoC/architecture specific sleep routine.
- **Device PM Engine:** The logic that handles device power state transitions (active → suspend/off, and back) either through system-managed device PM or runtime device PM.
- **SoC/Board Back-end:** The hardware implementation that actually executes entering into a low power state (e.g., system standby, deep sleep) or turning off device clocks, etc.
- **Policy / Constraint Manager:** Logic that decides which state to enter, based on idle time, device constraints, wake-up latency, minimum residence time, etc. (More detail in full docs but implied).

Workflows

Here's how things typically flow in Zephyr's PM subsystem (based on overview + additional documentation):

System PM Workflow

1. The Zephyr kernel scheduler finds that no threads are ready to run → it executes the "idle thread".
2. If CONFIG_PM is enabled, Zephyr's PM subsystem checks for possible low-power states that can be entered. (The policy logic may consider upcoming timer events, constraints from devices, minimum residency time, etc.)
3. If a low-power state is chosen, the system executes architecture/SoC specific routine to transition to that state. During this, clocks may be gated, CPU core may be put into "wait for interrupt" mode or deeper sleep.
4. A wake-up source (timer, external interrupt) triggers, the system resumes execution; the PM subsystem or back-end performs necessary restore operations.
5. Kernel resumes scheduling threads normally.

Device PM Workflow (system-managed)

1. Before the system enters a deep sleep state, the device PM subsystem may inspect all devices: are any busy? Are any wake-sources?
2. It may call the device's suspend callback (driver level) to put it into lower power state (or turn it off), if supported.
3. After system wakes up, resume callbacks are called.
(This is more detailed in device PM docs, but overview indicates the classification of device PM as part of the subsystem.)

Device Runtime PM Workflow

1. A device driver or client calls an API to "get" the device for use; if the device was suspended/off, a resume is triggered (so device becomes active).
2. After the usage is done, the driver/client calls API to "put" the device; if usage count goes to zero, the device may be suspended/off while system remains active.

3. This allows device-level power savings even when CPU is not idle.
(Again, more detail is in its section but overview references that device PM is a category.)

Use Cases

Here are scenarios where Zephyr's PM subsystem is valuable:

- **Battery-powered IoT sensor node:** A device that wakes periodically (say once a minute), reads sensors, sends data over BLE, then goes back to sleep. Using system PM → deep sleep during idle, device PM → suspend sensors/radio when inactive.
- **Wearables / smart devices:** A smartwatch or fitness tracker where the CPU may spend large time idle or waiting for user activity; devices (heartbeat sensor, display, BLE radio) can be suspended when not in use to prolong battery life.
- **Embedded gateway with sporadic peripheral use:** Example: a gateway that is mostly awake (system active) but many peripherals rarely used (USB, extra sensors). Using runtime device PM to suspend unused peripherals saves power even though CPU is active.
- **Low-power microcontroller in event-driven mode:** A controller that wakes on GPIO interrupt, performs a small task, then goes back to deep sleep. System PM handles the deep sleep; device PM handles turning off unused devices/power domains.
- **SoCs with multiple power domains:** Modern MCUs often support multiple domains (e.g., radio, sensors, CPU) that can be powered down individually. Zephyr's PM abstraction allows controlling at both system and device level.

Advantages

Some of the key benefits of using Zephyr's PM subsystem:

- **Hardware agnostic abstraction:** Because PM logic separates core infrastructure from SoC implementation, developers can reuse code across boards/SoCs. (From overview: architecture and SOC independence.)
- **Fine-grained control:** Both system-level and device-level power management allow for deep optimization of power consumption.
- **Better battery life / lower power usage:** Especially critical for IoT devices, wearables, sensor nodes: by suspending devices and entering idle/deep sleep, overall power draw goes down.
- **Flexibility / scalability:** The PM subsystem supports multiple policies (residency based, application-defined) and can adapt to simple or complex SoCs.
- **Wake-up support:** Devices and system can be configured to wake from low-power states on external events, enabling event-driven low duty-cycle operation.
- **Improved modularity:** Device drivers and applications can participate in PM (via APIs) rather than being always active; this aligns with modern embedded design.

Limitations

Recognising limitations is equally important:

- **Driver support needed:** To fully exploit device PM (especially runtime PM), device drivers must implement suspend/resume logic correctly (context save/restore, dependencies etc). Without that, benefit is limited.
- **Latency trade-offs:** Deeper power states often mean longer wake-latency. If system or device needs to wake quickly or frequently, the overhead may negate power savings.
- **Complexity:** Designing correct policies, tracking device dependencies, ensuring wake-sources are properly configured can add design effort.
- **Not automatically solving everything:** Although abstraction helps, hardware must support low-power states, clock gating, power domains. On less capable SoCs, benefit may be limited.
- **Idle time required:** For system PM to be effective, the system must actually be idle enough; if CPU is heavily loaded, less opportunity to sleep. Device PM still helps, but the gains are lower.
- **Wake-sources must be managed:** Since entering deep sleep disables many clocks/peripherals, making sure the correct sources are enabled and properly configured is the developer's responsibility.
- **Policy complexity for advanced usage:** For advanced policies (e.g., residency, upcoming event times, constraints) the logic may become non-trivial.

Real-World Examples

Here are a few real examples and how they tie in:

- On Silicon Labs Series 2 devices, Zephyr's PM subsystem is used: the system PM selects an energy mode requirement and passes it to the Silicon Labs Power Manager. The device runtime PM is used in drivers for peripherals (clock branches, pinctrl sleep states) to achieve low power during peripheral idle.
- On sensor-node platforms: Using Zephyr, developers implement a workflow where the CPU goes to deep sleep, sensors/radio are suspended, a timer awakens the system, then operations happen, then go back to sleep. The PM subsystem supports switching between idle states and managing device suspend/resume. (Seen in broader Zephyr documentation).
- A real measurement: On TI CC13x2 platform (ported to Zephyr), deep sleep ("standby" mode) drew ~0.85 μ A, and shutdown ~150 nA after leveraging Zephyr's PM and TI Power Manager backend.

Key Components / Concepts Summarised

Here is a bulleted list of the main components or concepts you should keep in mind (even though the overview section is brief, it refers to them implicitly and other docs expand them):

- **CONFIG_PM: Kconfig option to enable system power management.**
- **CONFIG_PM_DEVICE: Kconfig option to enable device power management.**
- **CONFIG_PM_DEVICE_RUNTIME: Kconfig option to enable runtime device PM.**
- **Power states:** For the system, multiple states like ACTIVE, IDLE, SUSPEND, OFF etc (varies by SoC). For devices: states like ACTIVE, SUSPENDED, OFF.

- **Policy manager:** Decides which system power state to enter (residency-based or application defined).
- **Constraints:** Components (drivers/subsystems) can prevent entering certain low power states if they need higher readiness.
- **Busy status indication:** Device PM infrastructure allows checking if a device is busy so that system PM won't try to suspend it while in use.
- **Wake-up capability / wake-source:** Devices may be configured to wake the system; in devicetree using wakeup-source.
- **SOC/Board backend:** The actual routines that enter low power states, restore context, implement clock gating, etc.
- **Runtime device PM engine:** Tracks usage counts of devices and triggers suspend/resume independent of system state.
- **Device tree and Kconfig bindings:** Specify device capability for power management, wake-sources, etc.

Interview Questions & Answers

Here are some likely interview-style questions (and suggested answers) around Zephyr's PM subsystem (overview + related concepts).

Q1. What is the significance of Zephyr's Power Management subsystem being "architecture and SoC independent"?

A1.

It means that the PM subsystem provides generic interfaces and APIs that are independent of the underlying CPU architecture or SoC vendor. The SoC/board-specific details (for entering low-power states, clock gating, power domains) are abstracted behind the PM backend. This allows application and OS code to use PM functions without being heavily tied to vendor specifics. The overview page states: "The interfaces and APIs ... are designed to be architecture and SOC independent."

Q2. What are the two main categories of power management features in Zephyr?

A2.

They are:

1. System Power Management – managing the whole system/SoC/CPU idle states.
2. Device Power Management – managing individual devices/peripherals (suspend/resume, off) when not in use.

Q3. Describe a simple workflow for system power management in Zephyr.

A3.

When the kernel finds no threads ready to schedule, it runs the idle thread. If CONFIG_PM is enabled, the PM subsystem determines an appropriate low power state (based on idle time, constraints). Then the board/SoC backend executes enter-sleep logic. On a wake-up event (timer, interrupt) the system resumes, kernel scheduling continues.

Q4. How does device power management differ from runtime device power management?

A4.

- Device Power Management (system-managed): Devices may be suspended/resumed as part of the system entering/exiting low power states.
- Runtime Device Power Management: Devices can be suspended/resumed individually *while the system (CPU) remains active*. For example, if the CPU is awake but a peripheral is idle, it can be turned off to save power.

Q5. What role does the policy logic play in system PM?

A5.

The policy logic decides *which* low-power state to enter based on factors such as next scheduled timer/event, minimum residency time required for each state, wake-latency, and any constraints set by devices or subsystems. One policy is “residency-based” (choose the deepest state whose minimum residency \leq predicted idle time).

Q6. What are constraints in Zephyr’s PM subsystem, and why are they useful?

A6.

Constraints are signals by subsystems or devices that prevent the system from entering certain deep sleep states because they need higher availability or shorter wake latency. They are useful to avoid entering a state where some component would malfunction or wake latency is too high. (Details in device PM docs, but referenced in overview expansions.)

Q7. List some advantages of using Zephyr’s PM subsystem.

A7.

- Improves power efficiency (especially in battery-powered/IoT devices) by idle/suspend mechanisms.
- Fine-grained control with both system and device level PM.
- Abstraction makes porting to different SoCs easier.
- Enables event-driven, low-duty-cycle operation via wake-sources and idle sleep.
- Improves modularity: devices and drivers can take part in PM rather than remain always active.

Q8. What are some of the limitations or challenges when using Zephyr PM?

A8.

- Driver/board support needed: Without proper suspend/resume logic, you don’t get full benefit.
- Wake latency vs power savings trade-off: if you sleep too deep, waking takes longer.
- Idle time requirement: If system is busy most of the time, less benefit from sleep modes.
- Complexity of implementing device dependencies, power domains, managing wake sources.
- Hardware limitations: Some SoCs may not support deep states or fine-grained power domains.

Q9. Give a real-world scenario where runtime device PM would be more beneficial than just system PM.

A9.

Consider a gateway that keeps its CPU awake to manage network traffic, but many peripherals (e.g., USB ports, additional sensors) are idle most of the time. Using runtime device PM, those idle peripherals can be suspended while the CPU remains active. System PM may not help much because the CPU isn't idle. So runtime device PM provides power savings in this scenario.

Q10. If you were porting Zephyr to a new SoC, what PM related tasks would you expect to do?

A10.

- Provide the SoC-specific backend implementations for entering/exiting sleep states (idle/suspend) and perhaps power domains.
- Map the SoC's power states into Zephyr's PM framework (defining pm_state values, their minimum residency, exit latency).
- Enable wake-sources (timer, GPIO, etc) and their integration.
- Ensure device drivers implement suspend/resume (or at least support them) and correctly interact with the PM subsystem.
- Possibly implement or tune policy logic (residency-based, application-defined).
- Configure devicetree/Kconfig so that devices wakeup or power domains are recognized.

Summary

In summary, the "Overview" section of Zephyr's PM subsystem tells us that Zephyr provides a generic, architecture-agnostic infrastructure for power management — covering both system (CPU/SoC) and device (peripheral) power states. By separating core PM logic from hardware specifics, Zephyr enables flexible power savings across embedded/IoT platforms. For the best results, device drivers, board support, and wake-up sources must be properly implemented and tuned. Understanding workflows, policies, device vs system PM, and trade-offs (latency vs savings) is crucial.

If you like, I can **pull in details from the *System* and *Device* sections (not just 'Overview')** of the same Zephyr PM documentation and provide a combined, richer summary (with diagrams, deeper workflows, and more interview questions) — would that be useful for you?

Here's a comprehensive breakdown of the **System Power Management** section of the Zephyr RTOS PM subsystem (via the link you shared:

1. Definitions

Here are the key definitions for system-level power management in Zephyr:

- **Kernel idle state:** When Zephyr's kernel has no threads ready to run, it enters the idle thread. In that state the system is idle and can potentially transition to lower power.
- **System Power Management (System PM):** The subsystem within Zephyr that, when enabled (via CONFIG_PM), allows the kernel to request the PM core to transition the system (SoC/CPU/peripherals) into one of the supported low-power states during idle time.
- **Power state (system-level):** An enumerated state (pm_state) representing a specific system/SoC/CPU low-power mode. Each state has defined power consumption, context retention, minimum residency (how long you should stay in it) and exit/wake latency.
- **Power management policy:** Logic which chooses *which* power state to enter (from the set supported on the platform) based on factors such as how long the system will be idle, constraints, latency, residency, etc.
- **Constraint:** A mechanism by which subsystems or devices can prevent transitions into certain power states (because maybe they need a peripheral active, or wake latency too high).
- **Wake-up event / wake-source:** An interrupt or peripheral event (timer, GPIO, etc) that will wake the system from the idle/sleep state. In system PM the application is responsible to set up a valid wake-up event before entering a low-power state.

2. Architecture / Key Components

Here's how the system-PM part of Zephyr is architected, and what its major components are.

Architecture overview

- When idle, the kernel requests the PM subsystem to choose a low-power state.
- The PM subsystem is divided into a **core infrastructure** (generic code) and a **SoC/board-specific back-end** which implements the actual state transitions (entering sleep, restoring context) for a given platform. This separation achieves architecture/SoC independence.
- The system PM logic includes:
 - The idle routine that triggers system suspend.
 - The policy manager (chooses state).
 - Device busy/constraint checking (ensuring devices are not busy and allowed states).
 - SoC interface call to execute the chosen state.
 - Wake-up / restore path.

Key components

- **Idle thread / k_cpu_idle():** The function that is executed by the kernel when no other threads are ready; idle processing happens here.
- **Policy Manager:** Determines which **pm_state** to use. It supports at least two policies: **residency-based** and **application-defined**.
- **State definitions (pm_state):** These are defined per platform (via devicetree binding zephyr,power-state) and include parameters like minimum residency and exit latency.
- **Constraints & busy status APIs:** Provide hooks for devices or subsystems to set constraints (prevent certain states) and to signal busy status to avoid entering low-power

when a device transaction is ongoing.

- **SoC/Board Back-end:** The implementation for `pm_power_state_set()` (or similar) that actually places the system into the requested low-power state, and `pm_system_resume()` for wake-up.
- **Wake-up configuration:** The application must configure a wake-event before sleeping; not all peripherals may remain active in all states.

3. Workflows

Here are typical workflows for how system PM works in Zephyr:

3.1 System idle → power state transition workflow

1. Kernel finds no threads to schedule → enters `k_cpu_idle()`.
2. If `CONFIG_PM` is enabled, the kernel calls into the PM subsystem (e.g., `pm_system_suspend()` internally) to determine what to do.
3. The policy manager evaluates idle time, constraints, and chooses a power state (e.g., `PM_STATE_SUSPEND_TO_IDLE`, or `PM_STATE_STANDBY`, depending on platform) via logic such as:
4. if (`time_to_next_event` \geq `state.min_residency` + `state.exit_latency`)
5. choose state
- 6.
7. Before entering that state, the subsystem checks any constraints or busy devices that would prevent entry. If a deeper state is disallowed, fallback to a shallower one (or stay active).
8. The SoC/board back-end is invoked (e.g., `pm_state_set(...)`) to actually put the system into the low-power state.
9. The system remains in that low-power mode until a wake-up event occurs (timer, GPIO interrupt). The application must set up such wake source beforehand.
10. Upon wake, the back-end executes `pm_system_resume()` or similar to restore context, enable interrupts, resume devices etc. Kernel scheduling resumes normal operation.

3.2 Example: Residency-based policy decision

- The policy manager computes the time until the next scheduled event (e.g., thread timer).
- It looks at available `pm_state` entries, each having `min_residency_us` and `exit_latency`. It picks the *deepest* state (highest power saving) such that:
$$\text{time_to_next_event} \geq \text{state.min_residency_us} + \text{state.exit_latency}$$
- If none satisfy that, it remains in `PM_STATE_ACTIVE` (or a shallow idle state) and returns to idle.

4. Use Cases

Here are some scenarios where system-level PM in Zephyr is particularly useful:

- **Battery-powered IoT node:** A sensor device wakes every minute, reads data, sends it, then remains idle for the rest of the minute. System PM can put the MCU into a deep sleep state for ~59 s, saving a lot of power.
- **Wearables / smart devices:** A smartwatch may have idle periods when user is inactive; system PM can reduce CPU/peripheral power during those idle periods.
- **Event-driven system:** A microcontroller that waits for a button press or sensor interrupt can sleep most of the time and wake only on the event. Setting up a GPIO wake-source and using system PM is an effective design.
- **Modern SoCs with multiple deep-sleep modes:** For example, a SoC may support light idle, standby, RAM-retention sleep, or full shutdown. Zephyr's system PM makes use of these states via policy manager and back-end implementation.
- **Low-duty-cycle communication devices:** A BLE beacon or LoRa node that transmits occasionally and spends most time idle; system PM ensures minimal power during idle except essential wake source.

5. Advantages

The benefits of using Zephyr's system power management include:

- **Lower system power consumption:** By leveraging idle time and entering deep sleep states, the CPU and many peripherals can be powered down, improving battery life.
- **Architecture/SoC independence:** The PM core is generic; only back-end implementation is SoC specific. This makes it easier to port across platforms.
- **Fine-grained control via policy:** The policy manager allows selecting the best state based on expected idle duration and latency/residency trade-offs.
- **Constraint and busy-status mechanisms:** These ensure that the system doesn't enter a low-power state when it's unsafe (e.g., a device is busy) or undesirable (e.g., latency too high).
- **Integration with device PM:** While system PM deals with CPU/SoC, it works together with device PM (if enabled) so devices can suspend in concert. This holistic view improves savings.
- **Flexible for a variety of use-cases:** From simple idle loops to deep shutdowns, system PM supports multiple power states defined on the platform.

6. Limitations

However, there are also limitations and trade-offs to be aware of:

- **Wake latency vs power savings trade-off:** The deeper the sleep state, the more power saved—but longer the wake-up time. If the system needs to respond quickly, deeper states may be unsuitable.
- **Idle time required:** For system PM to benefit, the system must have sufficiently long idle durations; if the CPU is busy almost all the time, system PM gains are small.
- **Hardware/SoC support required:** The available power states, their residency and exit latency are determined by the hardware. If the SoC doesn't support deep states, savings are limited.

- **Correct wake-source setup required:** It's the application's responsibility to set up a valid wake-up event. Without it, the system may stay in a sleep state indefinitely or may not wake.
- **Device dependencies:** If devices or peripherals need to remain active (or are busy), system PM may be prevented or reduced; proper coordination with device PM is necessary (though this belongs more to device PM).
- **Policy complexity / tuning:** Selecting the optimal policy (residency vs application defined) and tuning min_residency/latency values may be non-trivial to maximise power savings while maintaining performance.
-

8. Key Components / Concepts Summarised

Here is a summary of the main system-PM components and concepts to keep in mind:

- **CONFIG_PM:** Kconfig option to enable system power management.
- **Power states (pm_state):** Enumerated states representing system low-power modes, with associated parameters (min residency, exit latency).
- **Policy manager:** Determines the state to transition to; supports residency-based or application-defined policies.
- **Residency-based policy logic:** The core logic: choose deepest state such that $\text{time_to_next_event} \geq \text{state.min_residency} + \text{state.exit_latency}$.
- **Constraints / busy status:** Mechanisms to block certain states when devices/subsystems are active or higher responsiveness is required.
- **Wake-up setup:** Application must set up wake-source (timer, GPIO) appropriate for the chosen state. Some peripherals may be inactive in some states.
- **SoC/Board backend interface:** Functions like `pm_state_set()` / `pm_system_resume()` that implement the physical state transition.
- **Idle thread integration:** The system PM logic is invoked when the kernel's idle thread runs (`k_cpu_idle()`).

9. Interview Questions & Answers

Here are some likely interview questions (with sample answers) around Zephyr's system power management subsystem:

Q1. What does enabling CONFIG_PM do in Zephyr?

A1. When CONFIG_PM is enabled, Zephyr's kernel idle thread will, upon entering idle (no threads ready), invoke the power management subsystem to evaluate and potentially enter a low-power state. The system power management logic gets activated.

Q2. Explain how Zephyr chooses which system power state to enter.

A2. Zephyr uses the policy manager. If using the residency-based policy, it calculates how long the system will be idle (time until next scheduled event), and for each possible power state checks if $\text{time_to_next_event} \geq (\text{min_residency} + \text{exit_latency})$. It picks the deepest such

state (max power saving) that satisfies that. If none, it stays active or enters light idle. Foreign constraints (locks, busy devices) may prevent deeper states.

Q3. What are constraints in the system PM subsystem and why are they useful?

A3. Constraints allow subsystems or drivers to block transitions into certain low-power states when they require higher readiness or context retention. For example, if a peripheral must remain active or wake latency is critical, that subsystem can set a constraint to prevent entry into deeper sleep states where that peripheral would be disabled. This ensures safe operation.

Q4. What must an application do before the system enters a low-power state?

A4. The application must set up a valid wake-up event (timer, GPIO, interrupt) that will wake the system from the sleep state. Since in some low power states some peripherals/clocks may be disabled, only certain wake-sources may work—so the app must ensure choice of wake-source is compatible with the chosen state.

Q5. What are the trade-offs when choosing a deeper sleep state?

A5. Deeper sleep states offer greater power savings but have increased exit/wake latency (the time it takes to resume). Also the “minimum residency” may be larger (you must stay in the state for a minimum time to make the transition worthwhile). If the system needs to wake quickly, a shallow state may be preferable. Also some peripherals may become unavailable in deeper states.

Q6. How is system PM architecture in Zephyr made SoC independent?

A6. The architecture divides functionality into a generic core infrastructure (policy manager, idle integration, constraints, state definitions) and a SoC/board-specific backend that implements actual state transitions (`pm_state_set`, `pm_system_resume`) based on the platform's hardware. This separation allows the generic code to be reused across SoCs.

Q7. Give an example of a system power state enumeration and what it means.

A7. Examples (in some documentation) include: `PM_STATE_ACTIVE` (system fully powered and active), `PM_STATE_RUNTIME_IDLE` (all cores idle, waiting for interrupts), `PM_STATE_SUSPEND_TO_IDLE`, `PM_STATE_STANDBY`, `PM_STATE_SUSPEND_TO_RAM`, `PM_STATE_SOFT_OFF`. Each state has increasing depth of power saving but higher wake latency.

Q8. If your system has just 5 ms until next scheduled event, and a candidate power state has min_residency 10 ms and exit_latency 2 ms, will Zephyr choose that state under residency policy? Why or why not?

A8. No — because the rule is $\text{time_to_next_event} \geq (\text{min_residency} + \text{exit_latency})$. Here $5 \text{ ms} < (10 \text{ ms} + 2 \text{ ms}) = 12 \text{ ms}$, so the condition fails. Therefore Zephyr will not choose that deeper state; it will either stay in active or a lighter idle state. This prevents entering a state that won't pay off (i.e., you'd wake too soon).

Q9. What happens if a device is busy when the system tries to suspend?

A9. If a device signals it is busy (via busy-status API) then the power management subsystem will treat this as a blocking condition and may prevent or delay entering a deeper sleep state that would power down that device or context. Using busy status ensures the system doesn't suspend while hardware transactions are in progress.

Q10. What are some real-world considerations when using system power management in Zephyr in a battery-powered IoT device?

A10. Some considerations:

- Ensure wake-up interval is long enough to justify entering a deep state (i.e., overhead of transition is amortised).
- Choose wake-sources that remain functional in the targeted low-power state.
- Ensure device drivers/devices either support being suspended or will not block the transition.
- Tune policy: a residency-based policy may be fine but for some applications a custom (application-defined) policy might give better results (e.g., anticipating network traffic).
- Measure wake latency and context restore time to ensure system responsiveness is acceptable.
- Hardware: ensure the SoC supports deep sleep/ RAM retention etc or else you may not get anticipated savings.

10. Summary

In summary:

The system power management part of Zephyr gives you the infrastructure to reduce system (CPU/SoC) power by transitioning into idle/sleep states during inactivity. It relies on a policy manager to choose an appropriate state, constraints and busy status check to ensure safe transitions, and a backend to implement the state transition for each SoC. With correct configuration (wake-sources, device readiness, policy) you can achieve significant power savings—especially in idle/low-duty-cycle devices. However, you must account for wake latency, ensure hardware support, and properly integrate wake-sources and application logic.

Here is a detailed breakdown of the **Device Power Management (PM)** section from Zephyr

Definitions

- **Device Power Management (Device PM):** The portion of Zephyr's power-management infrastructure that deals with controlling the power states of individual devices/peripherals (drivers) — enabling them to suspend, resume, turn off, or be placed into lower-power modes.
- **Device Runtime Power Management:** A mode of device PM where a device may be suspended or resumed dynamically *while the system (CPU) remains active*. It uses

reference counting (usage count) to determine when a device is needed and when it can be suspended.

- **System-Managed Device Power Management:** A mode where devices are suspended/resumed in conjunction with the system entering/exiting a low power state (via system PM). i.e., when the CPU/SoC sleeps, devices too are suspended.
- **Device Power State:** An enumeration of the device's power status, e.g., ACTIVE, SUSPENDED, OFF etc. The PM subsystem tracks the state, drivers implement actions.
- **Device Action (`pm_device_action`):** The type of action requested of a device driver: e.g., SUSPEND, RESUME, TURN_OFF, etc. The driver must implement a callback to handle these.
- **Busy Status Indication:** A mechanism for a driver or subsystem to mark a device as "busy" (e.g., in the middle of a transaction) so that PM logic will not suspend it prematurely.
- **Wakeup Capability / Wake-source:** A property of a device that it can wake the system (or itself) from a low-power state. Device PM knows about it and the devicetree may include wakeup-source.

Architecture / Key Components

Architecture Overview

- The Device PM subsystem is part of Zephyr's broader PM infrastructure. It interacts with the Device Driver Model: each device driver that supports power management must register a callback (`pm_device_action_cb_t`) for handling power-state transitions.
- Two distinct modes (runtime vs system-managed) are supported, giving flexibility depending on how "power-aware" the device drivers/subsystems are.
- The infrastructure maintains per-device state, usage counts (for runtime), busy flags, and wakeup capability flags. It also interacts with system PM when required (i.e., devices must coordinate with system low-power transitions).
- The devicetree (and Kconfig) bindings allow specifying whether a device supports PM, whether runtime PM is enabled, wakeup-capability etc.

Key Components

- **`pm_device_action_cb_t`:** The device driver callback function invoked by the PM subsystem when a state transition is requested (action = SUSPEND / RESUME / TURN_OFF).
- **`enum pm_device_state`:** Defines the device states supported (ACTIVE, SUSPENDED, OFF; sometimes LOW_POWER) which drivers must handle.
- **Runtime API:** For runtime PM: `pm_device_runtime_get()`, `pm_device_runtime_put()`, `pm_device_runtime_put_async()`, `pm_device_runtime_enable()`, etc. These are used by drivers/subsystems to indicate usage and allow the PM subsystem to decide suspends/resumes.
- **System-Managed Device PM API:** For system-managed: `pm_device_state_set()`, `pm_device_state_get()`, `pm_device_wakeup_enable()`, `device_busy_set()`, `device_busy_clear()`, etc.

- **Busy-status API:** `device_busy_set()` / `device_busy_clear()` / `device_any_busy_check()` so that system PM will not transition devices while busy.
- **Devicetree/Driver Macros:** Macros like `PM_DEVICE_DEFINE`, `PM_DEVICE_DT_DEFINE`, `DEVICE_DT_DEFINE` etc. to register device PM contexts.

Workflows

Device Runtime PM Workflow

1. At driver initialization, the driver marks itself for runtime PM by calling `pm_device_runtime_enable()` or via devicetree flag `zephyr,pm-device-runtime-auto`.
2. When the driver or subsystem needs the device, it calls `pm_device_runtime_get(dev)`. This increments the usage count for that device. If the count was zero (device suspended/off), a resume action is triggered.
3. The driver performs its operation(s) on the device (e.g., communication, sensor read).
4. When done, the driver/subsystem calls `pm_device_runtime_put(dev)` (or asynchronously via `pm_device_runtime_put_async()`). This decrements the usage count; if it reaches zero then the PM subsystem may suspend (or turn off) the device.
5. While the system is active (not sleeping) the device can go into low-power states if idle. When the system itself goes to sleep, since runtime PM is enabled, the device is likely already in a low state (so system PM entry is faster).

System-Managed Device PM Workflow

1. System idle happens; the system PM subsystem decides to enter a lower power state (via system PM). Before doing so, it checks if the chosen state requires devices to be suspended (depending on devicetree property `zephyr,pm-device-disabled`).
2. If devices need suspension, the PM subsystem iterates through devices (in initialization order) and calls the driver suspend callbacks (`PM_DEVICE_ACTION_SUSPEND`). Dependencies are respected (parents before children).
3. The SoC/board backend executes the low-power transition.
4. On wake-up, the PM subsystem will resume devices in the reverse order (children first) via `PM_DEVICE_ACTION_RESUME`.
5. If any device cannot suspend (returns `-EBUSY`) or if `CONFIG_PM_NEED_ALL_DEVICES_IDLE` is set and a device is busy, the system will be prevented from entering the low state.

Use Cases

- **Peripheral idle in otherwise active system:** For example, a microcontroller controlling multiple sensors where only some are active at a time. Runtime PM allows suspending the unused sensors/peripherals while the CPU remains active.
- **Sensor bus or communications device:** A sensor on I2C bus which is only used occasionally; enabling runtime PM lets the driver suspend the sensor and possibly the bus when unused.
- **Wake-on-GPIO/interrupt device:** A GPIO controller or sensor that acts as a wake-source; device PM ensures that the device remains capable of wake-up while system is asleep.

- **System entering deep sleep:** When the CPU/SoC is about to enter standby or RAM-retention mode, using system-managed device PM ensures that all devices are in a consistent, low power state before system sleep.
- **Power domains and grouping:** Devices grouped under a power domain (e.g., radio, sensor domain) can be suspended as a group via device PM context.

Advantages

- **Granular power control:** Device PM allows individual peripherals to be suspended, saving power beyond system idle states.
- **Independence from system state** (runtime PM): Devices can be put into low-power while the system remains active, enabling dynamic power saving.
- **Better system PM performance:** With runtime PM, when system PM triggers, many devices may already be in low power, making system suspend faster.
- **Improved battery life in embedded/IoT devices:** By shutting off unused peripherals or domains, overall power consumption drops.
- **Driver-level integration:** Device drivers can implement suspend/resume logic to optimize hardware-specific transitions.
- **Wake-source handling:** Ensures wake-capable devices are properly managed and not inadvertently suspended.

Limitations

- **Driver implementation complexity:** Device drivers must implement the suspend/resume logic correctly, handle dependencies, and manage wake-sources.
- **Dependency management overhead:** For runtime PM, automatic handling of device dependencies is not fully supported: “as of today ... the device runtime power management API does not manage device dependencies.” Drivers must handle parent/child relationships themselves.
- **Overhead of transitions:** Frequent suspend/resume cycles may degrade performance or nullify power savings if transitions are too frequent.
- **Busy-status coordination required:** If a device is in the middle of a transaction (e.g., flash write) and is suspended, data corruption or unpredictable behavior may occur — developers must carefully call `device_busy_set()/clear()`.
- **Wake-capability configuration burden:** Wake-source devices must be properly configured (devicetree, driver) else suspension may cause lost wake events.
- **Not all devices may support PM:** Some devices may be always-on (legacy, critical) and thus cannot be suspended—they may block deeper system power states unless managed.

Real-World Examples

- In the Zephyr sample directory `samples/subsys/pm/device_pm/` you will find a demonstration of device PM features: showing how a device moves between ACTIVE and SUSPENDED states.
- On a TI CC1352R1 platform, Zephyr’s Device PM architecture combined with system PM allowed the device to achieve ~0.85 μ A in standby mode by turning off not just the CPU but many peripherals (device PM) as well. ([Linaro](#))

- A custom driver snippet from the docs for a “dummy” device: shows implementing `dummy_driver_pm_suspend()` and `dummy_driver_pm_resume()` with calls to `pm_device_runtime_get()` / `put()` for dependencies.

Key Concepts Summary

- Device states: `PM_DEVICE_STATE_ACTIVE`, `SUSPENDED`, `OFF`, (sometimes `LOW_POWER`). (zephyr-docs.listenai.com)
- Actions: `PM_DEVICE_ACTION_SUSPEND`, `RESUME`, `TURN_OFF`.
- Runtime vs System-Managed PM: which mode is used for a given device.
- Usage count / reference counting (runtime mode).
- Busy status indication so that devices aren’t suspended during operations.
- Wakeup capability and wake-source management.
- Devicetree and macros for device PM support.
- Handling dependencies: devices needing other devices/power domains must manage usage accordingly.

Interview Questions & Answers

Q1. What is the difference between Device Runtime Power Management and System-Managed Device Power Management in Zephyr?

A1. Runtime Device PM allows devices to be suspended/resumed while the CPU/system is still active (based on usage count). It gives fine-grained control and enables savings even when system isn’t sleeping. System-Managed Device PM, on the other hand, ties device suspend/resume to the system entering/exiting a deeper low-power state — when the CPU/SoC sleeps, devices are suspended as part of that transition.

Q2. What device states are defined in Zephyr’s device PM infrastructure?

A2. Key states include:

- `PM_DEVICE_STATE_ACTIVE` — device fully active, context retained.
- `PM_DEVICE_STATE_SUSPENDED` — device is suspended; context may be lost or must be restored.
- `PM_DEVICE_STATE_OFF` — device powered off, context lost.
(Some documentation also lists `PM_DEVICE_STATE_LOW_POWER`, where context is preserved but device clocks/power reduced.)

Q3. How does a device driver implement support for Device PM in Zephyr?

A3. The driver registers a power-management callback (`pm_device_action_cb_t`) which handles actions like `SUSPEND`, `RESUME`, `TURN_OFF`. At init, the driver uses macros (e.g., `PM_DEVICE_DEFINE` or `PM_DEVICE_DT_DEFINE`) to declare its PM context and pointer to it must be passed to `DEVICE_DEFINE` / `DEVICE_DT_DEFINE`. In runtime PM mode, the driver enables runtime PM via `pm_device_runtime_enable()` or devicetree flag `zephyr,pm-device-runtime-auto`. Then the driver uses `pm_device_runtime_get()` / `put()` in its operations to signal usage.

Q4. Why is busy-status indication important in device PM?

A4. Because if a device is in the middle of a hardware transaction (e.g., writing to flash, doing a DMA transfer) and the system or device PM logic suspends it, that might corrupt data or leave hardware in inconsistent state. By marking the device busy (`device_busy_set()`), the PM subsystem will avoid suspending it (or prevent system sleep) until `device_busy_clear()` is called.

Q5. What are the advantages of device PM for battery-powered embedded systems?

A5. Advantages include:

- Ability to reduce power consumption by shutting off or lowering power for unused peripherals.
- Savings even when the system CPU remains active (runtime PM).
- Faster system-sleep entry because devices may already be in low-power state.
- Better overall power/energy efficiency, which is critical for IoT or wearable devices.

Q6. What limitations or challenges should you be aware of when using device PM in Zephyr?

A6. Challenges include:

- The need for proper driver implementation (suspend/resume logic, dependencies, wake sources).
- Runtime PM does *not automatically handle dependencies* (e.g., child device and bus/parent device) — manual management may be required.
- Frequent transitions (suspend/resume) could introduce latency or performance overhead that offset power savings.
- Wake-source devices must be correctly configured or you may miss wake events.
- Some devices may not support suspend/resume safely or may need always-on, which complicates system PM or device PM usage.

Q7. In runtime PM mode, what triggers a device resume and what triggers a suspend?

A7. A resume is triggered when `pm_device_runtime_get()` is called and the usage count goes from zero to one (device was suspended/off). A suspend is triggered when `pm_device_runtime_put()` (or its async variant) is called and the usage count drops to zero — then the PM subsystem may schedule the suspend (or turn off) of the device.

Q8. How do wake-capable devices work with device PM?

A8. Devices that are wake-capable (devicetree property `wakeup-source`) are marked as such. The driver or application must call `pm_device_wakeup_enable(dev, true)` to enable their wake capability. Such devices typically should not be suspended during system sleep (or their wake capability might be disabled). Device PM helps ensure these devices remain capable of waking up the system when necessary.

Q9. Give an example of code snippet implementing suspend/resume in a driver for device PM.

A9. From the Zephyr docs (dummy driver example):

```
static int dummy_driver_pm_suspend(const struct device *dev) {
```

```

/* Request devices needed by this device */
(void)pm_device_runtime_get(config->enable_pin.port);
/* Disable interrupt pin */
gpio_pin_interrupt_configure_dt(&config->int_pin, GPIO_INT_DISABLED);
/* Turn device off */
gpio_pin_set_dt(&config->enable_pin, 0);
/* Release dependencies */
(void)pm_device_runtime_put(config->enable_pin.port);
(void)pm_device_runtime_put(config->int_pin.port);
return 0;
}

static int dummy_driver_pm_resume(const struct device *dev)
{
    (void)pm_device_runtime_get(config->enable_pin.port);
    (void)pm_device_runtime_get(config->int_pin.port);
    gpio_pin_set_dt(&config->enable_pin, 1);
    gpio_add_callback(...);
    gpio_pin_interrupt_configure_dt(&config->int_pin, GPIO_INT_EDGE_TO_ACTIVE);
    (void)pm_device_runtime_put(config->enable_pin.port);
    return 0;
}

```

Q10. If a device driver returns -EBUSY during PM_DEVICE_ACTION_SUSPEND, what happens?

A10. In system-managed device PM mode, if a device returns -EBUSY during its suspend callback, the system PM will abort entering that low power state (or fallback) because the device cannot be suspended safely. This prevents inconsistent states or data corruption.

Summary

The Device PM subsystem in Zephyr provides a structured, flexible way to manage the power states of individual devices/peripherals. By supporting both runtime (device-driven) and system-managed (system-driven) modes, it caters to a variety of use-cases: from peripherals that only need occasional activity, to systems entering deep sleep. For embedded/IoT applications, using device PM well can result in significant power savings—but it demands that drivers correctly implement suspend/resume logic, handle dependencies and wake-capability, and that usage patterns are tuned so that suspend/resume transitions are meaningfully amortised.

Definitions

- **Device Power Management (Device PM):** A framework in Zephyr that provides mechanisms to control the power states of individual devices (drivers/peripherals) in a coherent way, based on defined expectations (by drivers/subsystems/applications) and device-to-device dependencies.
- **Device Runtime Power Management:** One of the two supported methods. It allows devices to be suspended or resumed independently of the system sleep state (i.e., while CPU/system may still be active). It uses reference-counting to track usage.
- **System-Managed Device Power Management:** The second supported method, where devices are suspended/resumed in coordination with system power state transitions (i.e., when system/SoC goes to low-power).
- **Device Power State:** Internal representation of the device's mode (active, suspended, off) tracked by the PM subsystem. Transition requests are made via "actions" (e.g., SUSPEND, RESUME) and drivers implement hardware-specific logic.
- **Device Busy Status:** A flag or indication that a device is in the middle of a hardware transaction and should not be suspended yet (to avoid data corruption or inconsistent state when entering low-power).

Architecture / Key Components

Architecture Overview

- The Device PM subsystem is layered on top of the Zephyr device driver model: each driver optionally supports power management and registers a callback to handle transitions.
- It provides two methods (runtime vs system-managed) so system integrators can choose based on application/driver capabilities and system demands.
- Dependencies between devices (for example, a sensor depending on a bus or power domain) are recognized. While runtime PM currently has limited automatic dependency support, the infrastructure allows drivers/subsystems to manage dependencies.
- The subsystem abstracts the mechanism (generic code) from hardware specifics (drivers implement the actions). So it's hardware-agnostic to a large extent.

Key Components

- `pm_device_action_cb_t`: The driver-provided callback function that the PM subsystem calls to perform actions: suspend, resume, turn-off, etc.
- `enum pm_device_state`: Enumeration of device states (e.g., ACTIVE, SUSPENDED, OFF).
- **Runtime PM APIs:** For runtime mode — `pm_device_runtime_get()`, `pm_device_runtime_put()`, `pm_device_runtime_enable()`, etc. (More detailed in the runtime PM section, but this is introduced here.)
- **System-Managed Device PM APIs:** For the system mode — `pm_device_state_set()`, `pm_device_state_get()`, busy status functions (`device_busy_set()`, `device_busy_clear()`), wake-capability functions (`pm_device_wakeup_enable()`).

- Device registration macros: Drivers declare their PM context with macros such as `PM_DEVICE_DEFINE` or `PM_DEVICE_DT_DEFINE`.

Workflows

From the introduction section we can outline two high-level workflows:

1. Runtime Device PM Workflow (Introduction Stage):

- Device driver, subsystem or application indicates that a device will be used (via runtime PM get) → the device is resumed (if needed) and usage count increments.
- User uses the device (operations occur).
- When no longer needed, runtime PM put is called → usage count drops; if zero then the device may be suspended or turned off by the PM subsystem.
- Because this happens while the system is active, this allows saving power at device level even when CPU remains active.
- Since devices may already be in low power state, when the system enters a sleep mode the system PM doesn't need to spend as much time suspending devices. (As noted in the doc: "When using this Device Runtime Power Management, the System Power Management subsystem is able to change power states quickly because it does not need to spend time suspending and resuming devices that are runtime enabled.")

2. System-Managed Device PM Workflow (Introduction Stage):

- System decides to enter a low-power state (CPU/SoC).
- Before entering, the Device PM subsystem checks if the selected state triggers device suspend (depending on devicetree property `zephyr,pm-device-disabled`). If yes: then devices are suspended (in initialization order), ensuring dependencies are respected.
- Device drivers implement suspend/resume callbacks and respond accordingly.
- On wake-up, devices are resumed in reverse order.
- Note: This mode is simpler but has drawbacks (less flexibility) and runtime mode is preferred when possible.

Use Cases

- In an embedded IoT device with many peripheral devices: using runtime PM allows idle peripherals (e.g., sensor bus, extra modules) to be suspended even though the CPU remains active (in e.g., network gateway mode).
- In a wearable device (smartwatch): when display, sensors or radio are inactive, device PM can suspend those peripherals to save power.
- In a system where the application is not power-aware or cannot manage each device individually: system-managed mode allows devices to be suspended when the system goes into deep sleep (without detailed runtime logic).
- In a scenario where devices act as wake-sources (GPIO, interrupts): device PM ensures those devices remain configured to wake the system, while other devices are suspended.

Advantages

- More granular control of power consumption (device-level) rather than only system-level sleep.
- Runtime device PM allows power saving especially when CPU is active but many peripherals are idle.
- Allows the PM subsystem to speed up system sleep transitions because many devices already in low power.
- Unified framework abstracts hardware specifics; driver authors use standardized interfaces.
- Dependencies and wake-capability support help manage complex peripheral interactions.

Key Concepts / Features

- Two modes: **Runtime** and **System-Managed**. Runtime is preferred when device usage can be tracked; system mode is fallback.
- Usage count/reference counting in runtime mode.
- Device state transitions: ACTIVE → SUSPENDING → SUSPENDED → OFF (with corresponding actions).
- Device busy-status API to prevent suspends while in transaction.
- Wake-capable devices: marking as wake source so that suspension doesn't break wake functionality.
- Device registration and devicetree integration: drivers declare their PM support in DT and macros.
- Awareness of device dependencies (though runtime PM currently has limitations in automatic handling).

Real-World Examples

- The documentation shows a “dummy driver” example where the driver implements `dummy_driver_pm_suspend()` and `dummy_driver_pm_resume()` listed in the introduction section. (See the code snippet in “Device Model with Device Power Management Support” in the introduction.)
- The documentation lists sample directories: `samples/subsys/pm/device_pm/` and tests such as `tests/subsys/pm/device_wakeup_api/`. These serve as concrete implementations of device PM functions.
- Driver example from Nordic Developer Academy: their driver uses `pm_device_runtime_get()` before SPI transaction, and `pm_device_runtime_put()` afterwards to manage the peripheral runtime PM.

Limitations (as implied)

- Runtime PM does *not* automatically manage device dependencies (e.g., bus device for multiple sensors) – drivers may need to handle manually.
- System-managed mode: if a device cannot be suspended (returns `-EBUSY`), then system sleep may be blocked. So system mode has risks of being blocked by one device.
- Device drivers must implement correct suspend/resume logic; otherwise power savings will not be achieved or may cause errors.

- Device PM benefits depend on device idle periods; if devices are used continuously, little benefit.

Interview Questions & Answers

Q1. What are the two methods of Device Power Management supported in Zephyr's Device PM infrastructure?

A1. They are:

- Device Runtime Power Management — devices suspend/resume based on usage while the system is active.
- System-Managed Device Power Management — devices are suspended/resumed when the system enters/exits low-power states.

Q2. How does Runtime Device PM decide when to suspend a device?

A2. In runtime mode, the PM subsystem maintains a usage count (reference count) for each device. When a driver or subsystem calls `pm_device_runtime_get()`, usage count increases (device is brought to ACTIVE if needed). When they call `pm_device_runtime_put()`, usage count decreases; if it reaches zero, the PM subsystem may trigger a suspend.

Q3. In System-Managed Device PM mode, what happens if a device driver's suspend callback returns -EBUSY?

A3. If a device is busy (returns -EBUSY) in its suspend callback, then the system will not enter the chosen low power state (or may fall back). In other words, a single device's inability to suspend can block system power management transition.

Q4. Why is busy-status indication important for device PM?

A4. Busy-status ensures that devices which are in the middle of hardware transactions (e.g., writing flash, DMA in progress) are not suspended prematurely. Suspending such a device could lead to data corruption or undefined behavior. The busy-status API allows the PM subsystem or SOC backend to check if any device is busy before entering a low-power state. ([zephyr-docs.listenai.com](https://zephyr-docs.listenable.com))

Q5. When would you prefer Runtime Device PM over System-Managed Device PM?

A5. You would prefer Runtime Device PM when your application or subsystem is device-usage aware (knows when devices will be used/not used), and you want fine-grained power savings even when the system is active. Runtime mode allows devices to enter low power independently of system sleep, making it suitable for peripherals that are idle while CPU/network remains active. The documentation states that runtime PM is *preferred* when possible.

Q6. What role do devicetree properties and driver macros play in Device PM?

A6. Drivers must declare their PM support via macros like `PM_DEVICE_DEFINE` or `PM_DEVICE_DT_DEFINE` which allocate context for the PM subsystem. The devicetree node may include flags such as `zephyr,pm-device-runtime-auto` to auto-enable runtime PM or

wakeup-source to mark a device as capable of waking the system. This integration links the hardware descriptor to the PM infrastructure.

Q7. How does Device PM help the System PM subsystem?

A7. When devices are already suspended via runtime PM (i.e., device usage count is zero), then when the system decides to enter low power, the system PM engine needs to do less work to suspend devices. This results in faster entry into low power, reducing wake-latency overhead and enabling deeper states more efficiently. The introduction section notes this benefit.

Q8. What does the device state diagram look like in Zephyr's Device PM?

A8. The state diagram is roughly:

ACTIVE → (PM_DEVICE_ACTION_SUSPEND) → SUSPENDING → SUSPENDED →
(PM_DEVICE_ACTION_TURN_OFF) → OFF

SUSPENDED → (PM_DEVICE_ACTION_RESUME) → ACTIVE

OFF → (PM_DEVICE_ACTION_TURN_ON) → SUSPENDED

This diagram is shown in the introduction docs.

Q9. What limitations should a developer keep in mind when implementing Device PM?

A9. Some limitations:

- Runtime PM currently does not automatically manage device dependencies (e.g., parent/child device, bus). Drivers may need to manually manage dependencies.
- If a device is busy, system-managed mode may block system sleep transition.
- Drivers must implement correct suspend/resume logic; otherwise power savings may not materialize or system stability could be affected.
- The benefit depends on device idle periods; if devices are constantly used, PM benefits are limited.

Q10. Give a real-world scenario where a device driver uses Runtime Device PM APIs.

A10. For example, a SPI sensor driver: before beginning a transaction it calls `pm_device_runtime_get(dev)` to bring the device out of suspended state (if needed). After finishing the transaction, it calls `pm_device_runtime_put(dev)` to allow the device to be suspended. If multiple sensors use the bus device, the bus may stay active until all users call `put`. This pattern allows the bus (and sensor) to be suspended between uses, saving power even though the CPU remains active managing other tasks.

Summary

In short: The introduction section of Zephyr's Device PM documentation defines how Zephyr allows devices/peripherals to be managed with respect to power, defines two operation modes (runtime & system-managed), describes device states and driver responsibilities, and explains how this feature supports consistent, dependency-aware, granular device power

control. For embedded/IoT systems where peripherals are not always active, leveraging this framework can significantly reduce power consumption—but drivers and subsystems must implement it correctly and account for dependencies and wake-sources.

If you like, I can **pull out all the devicetree bindings, Kconfig options, and full API list** specifically for the *introduction + the rest of device PM section*, and compile them into a reference sheet for you. Do you want that?

The **Zephyr Device Power Management** documentation includes detailed API, Kconfig, and Devicetree configuration references. Here's a consolidated **reference sheet**:

✿ Key Devicetree Bindings

- `zephyr,pm-device-disabled`: disables PM for specific power states.
- `zephyr,pm-device-runtime-auto`: enables automatic runtime PM.
- `wakeup-source`: marks device as capable of waking the system.
- `zephyr,power-state`: defines CPU/SoC power states and latency properties.

⚙️ Core Kconfig Options

- `CONFIG_PM_DEVICE`: enable Device PM subsystem.
- `CONFIG_PM_DEVICE_RUNTIME`: enable runtime PM.
- `CONFIG_PM_DEVICE_SYSTEM_MANAGED`: enable system-managed PM.
- `CONFIG_PM_DEVICE_DEBUG`: adds debug tracing/logs.
- `CONFIG_PM_DEVICE_BROADCAST`: notify dependent devices of transitions.

🧠 Main API Functions

- `pm_device_state_set()`, `pm_device_state_get()`
- `pm_device_runtime_get()`, `pm_device_runtime_put()`
- `pm_device_runtime_enable()`, `pm_device_runtime_disable()`
- `device_busy_set()`, `device_busy_clear()`
- `pm_device_wakeup_enable()`, `pm_device_wakeup_disable()`
- `PM_DEVICE_DEFINE()` / `PM_DEVICE_DT_DEFINE()` — register device PM context.

Would you like me to compile these into a **tabular PDF or Markdown quick-reference guide** (with sections for API + Kconfig + Devicetree + examples)?

Here's a deep dive into **Zephyr RTOS's Device Runtime Power Management** subsystem (via the link: [Device Runtime PM](#)) — covering definitions, architecture, workflows, use-cases, advantages, key components, real-world examples, and interview-style Q&A.

1. Definitions

- **Device Runtime Power Management (Device Runtime PM):** A method in Zephyr's PM infrastructure where individual devices (peripherals, drivers) can be suspended or resumed independently of whether the CPU/system is in idle or active state — i.e., while the system remains active the device can still be powered down when not in use. ([\(\)](#))
- **Usage count / reference counting:** In runtime PM, each device has a usage count; when a driver/subsystem requests the device, the usage count increments (via `pm_device_runtime_get()`), and when done using it, the count decrements (via `pm_device_runtime_put()`). When it drops to 0, the device can be suspended. ([\(\)](#))
- **`pm_device_runtime_enable()`:** API call to enable runtime PM for a device. After enabling, if the device is currently active, it may immediately be suspended to save power. ([\(\)](#))
- **`pm_device_runtime_put_async()`:** An asynchronous variant of the put operation. After usage finishes, schedule suspension after a delay, allowing non-blocking behavior. ([\(\)](#))
- **Suspend/Resume callbacks:** The driver must implement actions for device suspend/resume via its PM callback. Although this is part of the general device PM infrastructure, runtime PM triggers these based on usage count. ([\(\)](#))

2. Architecture / Key Components

Architecture Overview

- Runtime PM sits atop the device driver model and integrates with Zephyr's PM subsystem. The device driver exposes a callback (`pm_device_action_cb_t`) handling actions like SUSPEND or RESUME. ([\(\)](#))
- When runtime PM is enabled for a device, the subsystem tracks the usage count and device state (ACTIVE vs SUSPENDED/...). When usage count drops to zero, it may trigger a suspend; when a get is requested, a resume is triggered. ([\(\)](#))
- The subsystem works independent of system sleep: Devices are handled while CPU may still be running. This enables savings even when the system is active. ([\(\)](#))
- Devicetree/Kconfig integration: A device node in devicetree may include the `zephyr,pm-device-runtime-auto` property to enable runtime PM automatically after init. ([\(\)](#))

Key Components

- **APIs for runtime PM:**
 - `pm_device_runtime_enable(dev)` — enable runtime PM on device. ([\(\)](#))
 - `pm_device_runtime_get(dev)` — increment usage count and possibly resume device. ([\(\)](#))
 - `pm_device_runtime_put(dev)` — decrement usage count and possibly suspend device when zero. ([\(\)](#))
 - `pm_device_runtime_put_async(dev, delay)` — schedule asynchronous suspension. ([\(\)](#))
 - `pm_device_runtime_is_enabled(dev)` — query if runtime PM is enabled for device. ([\(\)](#))
- **Driver PM action callback:** The driver must implement logic for actions like `PM_DEVICE_ACTION_SUSPEND` and `PM_DEVICE_ACTION_RESUME`. Example in docs. ([\(\)](#))
- **Usage count tracking:** Internal to PM subsystem, counts how many users currently hold the device; upon drop to zero, trigger suspend.
- **Devicetree auto-enable:** With `zephyr,pm-device-runtime-auto`, runtime PM is enabled automatically post-init with no explicit call. ([\(\)](#))

- **Suspend/Resume handling:** The driver may implement suspend/resume operations (clock gating, disabling hardware, saving state) triggered by PM subsystem. Seen in example in Nordic Developer Academy. ([Nordic Developer Academy](#))

3. Workflow

Here is a step-by-step typical workflow for runtime device PM:

1. Initialization:

- In driver init (mydev_init()), optionally mark device as initially suspended if hardware is already off (pm_device_init_suspended(dev)). ([link](#))
- Enable runtime PM for the device either by calling pm_device_runtime_enable(dev) or via the devicetree auto flag. The function may immediately suspend the device if it is active. ([link](#))

2. Usage of Device:

- When a driver or subsystem wants to use the device, call pm_device_runtime_get(dev). If the device is suspended (usage count 0), this triggers a resume action (driver callback). Then usage count increments. ([link](#))
- Perform device operations (read, write, communicate).

3. Release of Device:

- After operations complete, call pm_device_runtime_put(dev) (or pm_device_runtime_put_async(dev, delay) if asynchronous).
- This decrements the usage count; if it reaches zero, the PM subsystem triggers the driver suspend callback to suspend the device. ([link](#))

4. Repeat usage cycles:

Device may be used many times; each get/put manages the state. While CPU remains active, unused devices can stay suspended, saving power.

5. System sleep interaction:

Because devices are already in low-power state (thanks to runtime PM), when system PM invokes system sleep, less work is needed to suspend devices, making transitions faster. The docs say: “When using this Device Runtime Power Management, the System Power Management subsystem is able to change power states quickly because it does not need to spend time suspending and resuming devices that are runtime enabled.” ([link](#))

4. Use Cases

Here are typical scenarios where device runtime PM shines:

- An IoT sensor hub where the CPU remains active managing network stack, but many sensors/peripherals are idle most of the time. Runtime PM allows those idle peripherals to be suspended while CPU handles other work.
- A wireless gateway device: The CPU and radio stack remain active, but seldom-used peripherals (USB ports, I/O expansions, extra sensors) can be suspended via runtime PM to reduce power draw.
- A wearable device: Display is off, sensor modules not active, but CPU waits for user input; sensors and modules that aren't used can be suspended individually.
- A bus device (e.g., SPI, I2C) shared across multiple sensors: When not in use by any sensor, the bus driver can decrease usage count to zero, suspend the bus, saving power

from clock gating or power domain shutdown.

5. Advantages

- **Fine-granular power savings:** Devices can be individually suspended even while the system remains active, enabling power savings beyond system idle.
- **Improved system sleep transitions:** With devices already suspended, system PM can enter deeper sleep faster and with lower latency.
- **Driver/subsystem independence:** The reference counting model allows drivers, subsystems, and applications to interact with device PM without tightly coupling logic across layers. Drivers don't need to know about higher-level app usage; they just expose suspend/resume mechanics. (.)
- **Better battery efficiency:** For embedded/IoT systems, shutting off unused peripherals is major for battery-life extension.
- **Flexibility:** Works even when system remains active; good for devices with mixed usage patterns.

6. Limitations (Things to watch)

- **Driver implementation required:** Devices must implement suspend/resume logic; without that you may not get benefits or may have issues.
- **Dependency management:** If devices depend on other devices (bus, power domain), runtime PM may require explicit management of those dependencies because automatic handling is limited. (The documentation says dependencies currently not automatically managed). (.)
- **Overhead of transitions:** Frequent get/put cycles may add overhead and reduce benefit if devices are used very frequently with small idle windows.
- **Wake-capability and context loss:** If device suspension causes context loss, the driver must restore state on resume; this may add latency or complexity.
- **Hardware support required:** The hardware must support proper suspend/resume and have low-power features to gain real savings.
- **Usage count imbalance risk:** If get/put calls are mismatched, device may stay active (wasting power) or be suspended too early (causing failure).

7. Real-World Examples

- From the Zephyr docs: Example code snippet for a device driver implementing runtime PM:
- ```
static int mydev_init(const struct device *dev)
```
- ```
{
```
- ```
 /* optionally mark device suspended if already off */
```
- ```
    pm_device_init_suspended(dev);
```
- ```
 /* enable device runtime PM */
```
- ```
    ret = pm_device_runtime_enable(dev);
```
- ```
 if ((ret < 0) && (ret != -ENOSYS)) {
```
- ```
        return ret;
```
- ```
}
```



- ...
- }
- 
- static int mydev\_operation(const struct device \*dev)
- {
- ret = pm\_device\_runtime\_get(dev);
- if (ret < 0) {
- return ret;
- }
- /\* do work \*/
- ...
- return pm\_device\_runtime\_put(dev);
- }
- ( )
- From Nordic Developer Academy – the nRF SPI driver shows runtime PM usage: Before SPI transaction it calls pm\_device\_runtime\_get(dev), after completes pm\_device\_runtime\_put(dev).
- In Zephyr tests: there are tests/subsys/pm/device\_runtime\_api/ to verify runtime PM behavior. ( )

## 8. Key Components / Concepts Summary

- **Enable/Disable runtime PM:** pm\_device\_runtime\_enable(), pm\_device\_runtime\_disable()
- **Usage count ops:** pm\_device\_runtime\_get(), pm\_device\_runtime\_put(), pm\_device\_runtime\_put\_async()
- **Check status:** pm\_device\_runtime\_is\_enabled()
- **Auto-enable via devicetree:** zephyr,pm-device-runtime-auto
- **Driver callback:** pm\_device\_action\_cb\_t handles SCENARIOS like PM\_DEVICE\_ACTION\_SUSPEND / RESUME
- **Succinct state model:** Device states tracked internally (ACTIVE, SUSPENDED, OFF) even though driver implements actions.
- **Independent from system sleep:** Runtime PM allows device power control while system is active.
- **Modular layering:** Drivers, subsystems, applications can all request gets/puts without knowing each other.

## 9. Interview Questions & Answers

### Q1. What is device runtime power management in Zephyr, and how does it differ from system-managed device power management?

**A1.** Device Runtime PM allows individual devices to be suspended or resumed based on usage while the system (CPU) is still active. In contrast, system-managed device PM occurs when the system itself enters a low-power state and devices are suspended in coordination with that. Runtime mode therefore gives finer control and can save power even when CPU is doing other work. ( )

## Q2. What steps must a driver perform to support runtime PM?

**A2.** The driver must:

1. Implement the PM action callback (`pm_device_action_cb_t`) handling `PM_DEVICE_ACTION_SUSPEND`, `RESUME`, etc. ( )
2. In initialization: optionally mark device as initially suspended via `pm_device_init_suspended(dev)` if hardware is already off, then call `pm_device_runtime_enable(dev)` (or set `zephyr,pm-device-runtime-auto` in devicetree). ( )
3. In operational code: before using the device call `pm_device_runtime_get(dev)`, after usage call `pm_device_runtime_put(dev)` (or async variant). ( )

## Q3. How does the usage count mechanism work in runtime PM?

**A3.** Every call to `pm_device_runtime_get(dev)` increments the usage count; if it was zero, a resume is triggered. Each `pm_device_runtime_put(dev)` decrements the usage count; when it drops to zero, the subsystem triggers the driver's suspend callback (or schedules it if `put_async`). This count ensures the device stays active while someone is using it, and suspends only when no users remain. ( )

## Q4. What is `pm_device_runtime_put_async()` and when would you use it?

**A4.** `pm_device_runtime_put_async(dev, delay)` schedules a suspend operation for the device after the given delay if usage count is zero. It is used when suspend might be slow or blocking, or when you want to defer suspension to avoid immediate suspend/resume churn. ( )

## Q5. What are some advantages of using device runtime PM in an embedded system?

**A5.** Advantages include:

- More granular power savings by suspending idle devices while system remains active
- Improved battery life
- Faster system low-power transitions because devices may already be suspended
- Modular driver/subsystem design: drivers don't need to know about higher levels, just use gets/puts.

## Q6. What limitations or pitfalls should you watch out for with runtime PM?

**A6.** Some pitfalls:

- Drivers must implement correct suspend/resume logic; issues here impact stability or savings
- Dependencies between devices (bus, power domain) may need manual handling since runtime PM doesn't automatically manage them fully.
- Frequent get/put cycles may cause overhead, reducing benefit
- Hardware must support suspend/resume and context restore
- Mis-balanced get/put may leave device active or suspended incorrectly

## Q7. If you have a sensor on an I2C bus used sporadically, how would runtime PM help? Describe how you would use it.

**A7.** I would enable runtime PM for the sensor device (and possibly the I2C bus if it supports it). In the sensor driver's operation routine: before starting a read, call

`pm_device_runtime_get(sensor_dev)` (and if needed `pm_device_runtime_get(i2c_bus_dev)`). After reading, call `pm_device_runtime_put(sensor_dev)` (and bus dev). Usage count logic will allow the sensor (and bus) to suspend when no one uses them. During the long idle periods between sensor reads, the bus and sensor will be suspended, saving power even though CPU remains active handling other tasks.

#### **Q8. What devicetree property allows automatic runtime PM enabling and how is it used?**

**A8.** The `zephyr,pm-device-runtime-auto` flag in the devicetree device node enables runtime PM automatically; the subsystem will invoke `pm_device_runtime_enable()` right after the device init function returns successfully, without explicit calls in driver init. ([\(\)](#))

#### **Q9. How does device runtime PM integrate with system PM to speed up system low-power transitions?**

**A9.** Because runtime PM suspends devices when they're unused even while system is active, by the time the system (CPU/SoC) goes idle and system PM initiates a low-power transition, many devices may already be in suspended state. This reduces the suspend/resume work needed for those devices during system PM, making the system transition faster and allowing deeper states with less wake latency. The docs state: "When using this Device Runtime Power Management, the System Power Management subsystem is able to change power states quickly..." ([\(\)](#))

#### **Q10. In code, if `pm_device_runtime_get()` returns a negative error (other than `-ENOSYS/-ENOTSUP`), what should you do?**



**A10.** You should treat it as a failure to resume the device and handle accordingly (e.g., abort the operation or fall back). According to commit notes, when runtime PM is disabled or unsupported the get/put are no-ops returning 0, but if they return a negative error (not `ENOSYS/NOTSUP`) then it indicates a resume failure and should not be ignored. ([juju.nz](#))

## 10. Summary

The Device Runtime Power Management feature in Zephyr provides a powerful mechanism to manage peripheral power use at a fine granularity – allowing devices to be suspended when idle even while the system remains active. By properly implementing driver support (suspend/resume actions, usage count gets/puts) and enabling runtime PM, embedded systems can significantly reduce peripheral power draw, extend battery life, and improve system low-power transitions. However, success depends on correct driver logic, dependency management, and hardware capability.

Perfect 

Let's make this **complete Zephyr Runtime Power Management (PM) Cheat Sheet** – tailored for **embedded systems and interview preparation**.

Below is the **structured outline** I'll include – please confirm if you want it as a  **PDF**,  **Markdown (.md)**, or both.

# ⚙️ Zephyr Device Runtime Power Management – Comprehensive Cheat Sheet

## 🧩 1. Overview

- **Purpose:** Save power by suspending/resuming devices *independently* of system sleep.
- **Scope:** Works while CPU/system is active.
- **Mechanism:** Reference counting (get/put) + driver suspend/resume callbacks.
- **Auto-enable:** via `zephyr,pm-device-runtime-auto` Devicetree property.

## 🏗️ 2. Architecture

### Layers:

Application

↓

Driver / Subsystem

↓

Device Runtime PM Subsystem

↓

Driver PM Callbacks (Suspend/Resume)

↓

Hardware Device

### States Managed:

- ACTIVE → SUSPENDED → OFF
- Driven by usage count transitions.

## 🔑 3. Core APIs

| Function                                             | Purpose                                         |
|------------------------------------------------------|-------------------------------------------------|
| <code>pm_device_runtime_enable(dev)</code>           | Enable runtime PM for device.                   |
| <code>pm_device_runtime_disable(dev)</code>          | Disable runtime PM for device.                  |
| <code>pm_device_runtime_get(dev)</code>              | Increment usage count, resume device if needed. |
| <code>pm_device_runtime_put(dev)</code>              | Decrement usage count, suspend when zero.       |
| <code>pm_device_runtime_put_async(dev, delay)</code> | Schedule suspend after delay (non-blocking).    |
| <code>pm_device_runtime_is_enabled(dev)</code>       | Query runtime PM enable status.                 |
| <code>pm_device_init_suspended(dev)</code>           | Mark device as initially suspended at init.     |

#### 4. Devicetree Properties

| Property                                   | Meaning                                |
|--------------------------------------------|----------------------------------------|
| <code>zephyr,pm-device-runtime-auto</code> | Auto-enable runtime PM post-init.      |
| <code>wakeup-source</code>                 | Device can wake system from low power. |
| <code>zephyr,pm-device-disabled</code>     | Disable device PM for specific states. |

#### 5. Kconfig Options

| Option                   | Description                             |
|--------------------------|-----------------------------------------|
| CONFIG_PM_DEVICE_RUNTIME | Enable runtime PM support globally.     |
| CONFIG_PM_DEVICE         | Enable general device power management. |
| CONFIG_PM_DEVICE_DEBUG   | Add debug logging/tracing.              |

## 6. Typical Driver Integration Flow

```
static int mydev_init(const struct device *dev)
{
 pm_device_init_suspended(dev);
 return pm_device_runtime_enable(dev);
}
```

```
static int mydev_read(const struct device *dev)
{
 pm_device_runtime_get(dev);
 /* Perform hardware read */
 ...
 pm_device_runtime_put(dev);
 return 0;
}
```

```
static int mydev_pm_ctrl(const struct device *dev,
 enum pm_device_action action)
{
 switch (action) {
 case PM_DEVICE_ACTION_RESUME:
 /* Enable clocks, power, restore state */
 break;
 case PM_DEVICE_ACTION_SUSPEND:
 /* Disable clocks, power down */
 break;
 default:
 return -ENOTSUP;
 }
 return 0;
}
```

}

## 7. Workflow Summary

1. Device driver enables runtime PM.
2. On each use → `pm_device_runtime_get()` → Device resumes.
3. After use → `pm_device_runtime_put()` → Device suspends when count = 0.
4. Optional async suspend → non-blocking.
5. System PM benefits from devices already being suspended.

## 8. Use Cases

- **IoT Nodes:** Sensor or modem idle most of the time.
- **Wearables:** Display and sensors off between events.
- **Gateways:** USB / SPI peripherals used intermittently.
- **Bus Devices:** Shared I<sup>2</sup>C/SPI bus suspended when idle.

## 9. Advantages vs Limitations

| Advantages                  | Limitations                           |
|-----------------------------|---------------------------------------|
| Fine-grained power control  | Drivers must implement suspend/resume |
| Independent of system sleep | Dependency handling not automatic     |
| Better battery efficiency   | Risk of usage-count imbalance         |
| Faster sleep transitions    | Overhead if used too frequently       |

## 10. Real-World Examples

- Nordic nRF SPI driver (uses get/put before/after transfers).
- Zephyr sample: `tests/subsys/pm/device_runtime_api/`.
- Sensors (like LIS2DH12 accelerometer) using runtime PM for periodic reads.

## 11. Interview Q&A Highlights

**Q:** What's the difference between runtime and system device PM?

**A:** Runtime PM works while system is active; system PM occurs during sleep transitions.

**Q:** Why use `pm_device_runtime_put_async()`?

**A:** To delay suspension, avoiding immediate resume/suspend churn.

**Q:** How does usage counting work?

**A:** Tracks how many users hold the device. Suspends only when count hits 0.

**Q:** What's needed in a driver to support runtime PM?

**A:** PM callback for suspend/resume + get/put logic around device use.

## ⚙️ 12. Debugging & Tips

- Enable `CONFIG_PM_DEVICE_DEBUG` to trace state transitions.
- Verify usage count symmetry (same number of get/put).
- Check if device supports suspend/resume in hardware.
- Combine with system PM for maximal battery gain.

---

Here is a detailed breakdown of the **Zephyr RTOS Power-Domain subsystem**

## 1. Definitions

- **Power Domain:** In Zephyr's PM subsystem, a *power domain* is an abstraction representing a grouping of devices that share a common power source (for example, a regulator, clock/clock domain, or SoC region). The subsystem allows the domain to notify all member devices when the domain's power state changes (e.g., turned on/off) in a generic way. ([\(\)](#))
- **Internal Power Domains:** These refer to power-domains within a SoC (integrated circuits) where many device blocks/peripherals share a domain that can be powered off (to reduce leakage or idle power) when unused. ([\(\)](#))
- **External Power Domains:** These refer to power sources external to the SoC — e.g., a regulator or power-switch controlling a group of peripherals on a board, typically external devices. Zephyr supports these via the power-domain abstraction. ([\(\)](#))
- **Domain Notification / Device Action:** When a power domain changes its state (turn on/off), it must notify its member devices by invoking their PM action callbacks with actions like `PM_DEVICE_ACTION_TURN_ON` or `PM_DEVICE_ACTION_TURN_OFF`. ([\(\)](#))

## 2. Architecture / Key Components

### Architecture Overview

- The power-domain infrastructure in Zephyr is an optional layer (enabled via `CONFIG_PM_DEVICE_POWER_DOMAIN`). ([\(\)](#))
- A power-domain is defined as a device node (in devicetree) with compatible "power-domain". That node implements the PM action callback to handle domain state transitions, e.g., `TURN_ON`, `TURN_OFF`, `SUSPEND`, `RESUME`. ([\(\)](#))
- Devices (peripherals) can specify a `power-domains = <domain_node>` property in their devicetree node to indicate which domain they belong to. When the domain changes state, all member devices will be notified via their driver callbacks. ([\(\)](#))
- The workflow typically: when usage of one of the devices calls `pm_device_get()` (or similar) the domain may get powered ON, then devices resume. When usage drops, the domain may be turned OFF and devices are notified accordingly.
- The abstraction allows drivers/applications to treat groups of devices collectively (via their domain) without needing to know which devices share the same power source.



## Key Components

- **Power Domain Device Node:** Declared in Device Tree. Example:
  - gpio\_domain: gpio\_domain@4 {
  - compatible = "power-domain";
  - ...
  - };
  - ()
- **PM Action Callback for Domain:** e.g., mydomain\_pm\_action(const struct device \*dev, enum pm\_device\_action action) that handles PM\_DEVICE\_ACTION\_TURN\_ON, TURN\_OFF, SUSPEND, etc. It must notify children devices: pm\_device\_children\_action\_run(dev, PM\_DEVICE\_ACTION\_TURN\_ON, NULL); etc. ()
- **Device Declaration with Power Domain:** E.g.:
  - &gpio0 {
  - compatible = "zephyr,gpio-emul";
  - power-domains = <&gpio\_domain>;
  - };
  - ()
- **Notification API:** pm\_device\_children\_action\_run(...) is used to propagate actions from domain to devices. ()
- **Kconfig Option:** CONFIG\_PM\_DEVICE\_POWER\_DOMAIN to enable power-domain support. ()

## 3. Workflows

Here is how the typical workflow of a power-domain plays out:

1. A device in a domain transitions state (e.g., driver calls pm\_device\_get() or usage count rise) → this may trigger the domain to TURN\_ON if not already powered.
2. The domain driver executes its PM action callback with PM\_DEVICE\_ACTION\_TURN\_ON. Inside that callback it powers up the domain hardware (regulator, clock, etc) and then calls pm\_device\_children\_action\_run(dev, PM\_DEVICE\_ACTION\_TURN\_ON, NULL) to notify all child devices. ()
3. Devices receive the TURN\_ON action via their driver callbacks — they resume or configure themselves.
4. At idle, when no device is using the domain, the domain driver may accept a PM\_DEVICE\_ACTION\_TURN\_OFF action. It then notifies children via pm\_device\_children\_action\_run(..., PM\_DEVICE\_ACTION\_TURN\_OFF, NULL) then disables/regulates the domain power. ()
5. If a wake-source device is in the domain, or the domain is required for part of system activity, domain may remain powered or be notified accordingly.
6. Devices in the domain may also be suspended/resumed individually via their own device PM logic; domain logic remains orthogonal but coordinated.

## 4. Use Cases

Here are practical use-cases for power domains:

- **SoC internal power regions:** On many modern SoCs, groups of peripherals share a region that can be shut off when not needed (e.g., audio domain, sensor domain). Using Zephyr's power-domain abstraction, the region can be treated as a domain, and when all peripherals are inactive, the region is powered off, saving leakage current.
- **External board power rail:** Suppose USB devices on a board all share a 5 V/3.3 V regulator that can be disabled when not in use. Defining a power-domain for that rail allows Zephyr drivers of those devices to implicitly handle the power-rail enable/disable and notify devices.
- **Wake-on-Interrupt domain:** A domain containing peripherals (e.g., accelerometer + GPIO) that acts as a wake-source: Define domain such that its power rail stays enabled when wake-capability devices are active, and disabled otherwise — enabling ultra low power in idle.
- **Shared bus with multiple devices:** Devices on a shared bus (SPI, I2C) may share a power domain. When none of them are active, the domain can be turned off, disabling the bus entirely (clocks/power), saving power.

## 5. Advantages

- **Grouping Efficiency:** Instead of managing each device individually, you can manage a whole group of devices via one domain abstraction — simpler for power rail/regulator/clock-domain control.
- **Leakage Power Reduction:** By powering off entire domains when unused, you reduce static/leakage current (especially for internal SoC domains).
- **Scalability:** Multiple devices share the abstraction; drivers don't need to know which other devices share the domain.
- **Generic Notification:** Devices automatically notified of domain changes via PM action callback, enabling consistent state changes.
- **Flexibility:** Works for internal SoC domains and external power domains alike (regulators, board rails).

## 6. Limitations & Considerations

- **Driver Complexity:** Domain driver must correctly implement the PM action logic (TURN\_ON, TURN\_OFF) and notify children; device drivers must handle domain-actions accordingly.
- **Correct Device Tree Setup Required:** Devices must be correctly assigned to their domain via power-domains property; misconfiguration may lead to devices powered when they shouldn't or domain left on unnecessarily.
- **Wake-Source Handling:** If a device in domain is a wake source, domain must stay enabled or be configured to allow wake; otherwise wake may be lost. Developers must handle that. ([link](#))
- **Dependency Handling:** Devices may depend not only on their own domain but other resources (e.g., clocks, regulators); domain abstraction reduces but does not eliminate complexity.
- **Not Always Applicable:** On simple microcontrollers without distinct power domains, the abstraction might not yield significant benefit and may add overhead.

- **Latency / Transition Cost:** Turning off a domain may introduce latency for devices to resume; for devices with frequent use you may lose benefit.

## 7. Real-World Examples

- In the Zephyr documentation example (under “Implementation guidelines”) a domain called `gpio_domain` is defined in devicetree; devices `gpio0` and `gpio1` are assigned to that domain; when the domain powers on or off, both GPIO devices are notified. ( )
- In tests: Zephyr has `tests/subsys/pm/power_domain/` demonstrating power-domain usage. ( )
- Example board design: On a battery-powered IoT board, the sensor and amplifier rails might be grouped in a domain. When sensors sleep, the domain is turned off to save quiescent current — Zephyr’s power-domain infrastructure supports this pattern.

## 8. Interview Questions & Answers

### Q1. What is a power domain in Zephyr and why is it useful?

**A1.** A power domain is an abstraction representing a group of devices that share a common power source (regulator, clock domain, or SoC region). It allows the system to notify all member devices when the domain’s power state changes (ON/OFF) without each device needing to know about the others. This is useful because it simplifies managing power for groups of peripherals, enables turning off whole domains to save leakage power, and improves scalability of driver/board power management. ( )

### Q2. How do you define a power domain in Zephyr’s devicetree?

**A2.** You declare a node compatible with "power-domain", for example:

```
gpio_domain: gpio_domain@4 {
 compatible = "power-domain";
 ...
};
```

Then devices that belong to this domain specify:

```
&gpio0 {
 compatible = "zephyr,gpio-emul";
 power-domains = <&gpio_domain>;
};
```

This sets up the domain and associates devices with it. ( )

### Q3. Describe the workflow when a power domain is turned on.

**A3.** When a domain is requested (e.g., a device in it gets used), the domain driver receives `PM_DEVICE_ACTION_TURN_ON` via its PM action callback. Inside that callback, the domain hardware is enabled (e.g., regulator turned on), then the driver calls `pm_device_children_action_run(dev, PM_DEVICE_ACTION_TURN_ON, NULL)` to notify all

member devices of the turn-on event. These devices can then resume or configure themselves. ( )

**Q4. What happens if a device in a domain is a wake-source and the domain is turned off incorrectly?**

**A4.** If the device is a wake-source (meaning it must be able to generate a wake interrupt or event), turning off the domain (power rail) may disable that device, causing the wake event to be lost. In Zephyr it is the responsibility of the driver or application to mark the domain or the device as wake-capable and ensure the domain remains powered or properly configured to allow wake. ( )

**Q5. What Kconfig option controls enabling power domain support in Zephyr?**

**A5.** The option `CONFIG_PM_DEVICE_POWER_DOMAIN` enables the power-domain abstraction support in Zephyr's PM subsystem. ( )

**Q6. Give an example use-case where using a power domain helps reduce power consumption.**

**A6.** On a wearable sensor board, suppose all sensors (accelerometer, gyroscope, magnetometer) share a 1.8 V regulator that can be turned off when sensors are idle. Defining a power domain for the sensor rail means when no sensor is active, the domain can be shut off (regulator disabled), thus saving all leakage current of the entire sensor block, not just individual devices. When a sensor wakes for a measurement, the domain is turned on and all devices are notified.

**Q7. What must a power-domain driver implement in Zephyr?**

**A7.** The driver must implement a PM control callback (e.g., `mydomain_pm_action()`) that handles actions like `PM_DEVICE_ACTION_TURN_ON`, `PM_DEVICE_ACTION_TURN_OFF`, `PM_DEVICE_ACTION_RESUME`, `PM_DEVICE_ACTION_SUSPEND`. In `TURN_ON` it should power up the domain and notify children (`pm_device_children_action_run(...)`). In `TURN_OFF` it should notify children and then shut down the domain. It needs to be defined as a device in the devicetree. ( )

**Q8. What are the limitations or things to watch when using power domains in Zephyr?**

**A8.** Some considerations:

- Devices must be correctly declared under the domain; mis-assignments lead to domains staying on unnecessarily or devices mis-behaving.
- If wake-sources are involved, domain must remain powered or configured accordingly; failure leads to wake failure.
- Turning off domains introduces latency when turning them on again; for frequent use the transition cost may outweigh savings.
- Complexity: driver must handle domain actions, children devices must handle notifications. So there is development overhead.
- On simple hardware without distinct power domains, abstraction may not add much benefit and may complicate design.

### Q9. How does Zephyr notify devices when their power domain changes state?

**A9.** The domain driver uses `pm_device_children_action_run(dev, ACTION, NULL)` to notify all child devices of a domain about the action (`TURN_ON` or `TURN_OFF`). The devices receive a PM action callback (in their driver) with the action and can respond accordingly (resume/suspend). ([↗](#))

### Q10. Can power domains be nested (hierarchical) in Zephyr?

**A10.** Yes. In SoCs many internal power domains are hierarchical (domains may contain sub-domains) and Zephyr's abstraction supports nesting in devicetree (internal domains can be nested). For example, a SoC "sensor domain" may include sub-domains for accelerometer, magnetometer, etc. ([↗](#))

## 9. Summary

The power-domain feature in Zephyr's PM subsystem provides an extra level of abstraction for power management: grouping devices that share a power source, enabling on/off of entire domains rather than just individual devices. It aids power savings (especially leakage/static current) in more complex hardware (SoCs with multiple power rails or domains). The approach requires correct driver and devicetree support but offers scalable and unified management of device groups. If used correctly, it complements device PM and system PM to maximise overall energy efficiency.

If you like, I can **create a cheat-sheet** (one-page reference) for the Zephyr Power Domain APIs, devicetree bindings and design guidelines (with sample code) — would you like that?

Here's a **complete list of key components, Kconfig options, devicetree bindings, and APIs** for **Zephyr Power Domain** from the official docs:

## 1. Key Components

- **Power Domain Device:** Device node with `compatible = "power-domain"`.
- **Child Devices:** Devices belonging to a power domain (referenced via `power-domains` property).
- **PM Action Callback:** Implemented in domain or device driver to handle actions like `TURN_ON`, `TURN_OFF`, `SUSPEND`, `RESUME`.
- **Notification Mechanism:** `pm_device_children_action_run()` to notify all child devices of domain state changes.

## 2. Kconfig Options

| Option                        | Description                                                    |
|-------------------------------|----------------------------------------------------------------|
| CONFIG_PM_DEVICE_POWER_DOMAIN | Enable power domain support in Zephyr.                         |
| CONFIG_PM_DEVICE              | Enable general device PM support (required for power domains). |
| CONFIG_PM_DEVICE_DEBUG        | Enable debug logging for PM devices.                           |

### 3. Devicetree Bindings

| Binding                        | Description                                           |
|--------------------------------|-------------------------------------------------------|
| compatible = "power-domain"    | Declares a node as a power domain.                    |
| power-domains = <&domain_node> | Assigns a device to a power domain.                   |
| wakeup-source                  | Marks the domain/device as a wake source if required. |

**Example Devicetree:**

```

gpio_domain: gpio_domain@4 {
 compatible = "power-domain";
};

&gpio0 {
 compatible = "zephyr,gpio-emul";
 gpio-controller;
 power-domains = <&gpio_domain>;
};

&gpio1 {
 compatible = "zephyr,gpio-emul";
 gpio-controller;
 power-domains = <&gpio_domain>;
};

```

```
};
```

## 4. Core APIs / Functions

| API                                                           | Purpose                                                          |
|---------------------------------------------------------------|------------------------------------------------------------------|
| <code>pm_device_children_action_run(dev, action, NULL)</code> | Notify all child devices of a domain about a PM action.          |
| <code>pm_device_get(dev)</code>                               | Increment usage count and ensure device/domain is powered.       |
| <code>pm_device_put(dev)</code>                               | Decrement usage count; suspend device/domain if count reaches 0. |
| <code>pm_device_runtime_enable(dev)</code>                    | Enable runtime PM for device.                                    |
| <code>pm_device_runtime_disable(dev)</code>                   | Disable runtime PM.                                              |

## 5. PM Action Callback Example

```
static int mydomain_pm_action(const struct device *dev,
 enum pm_device_action action)
{
 switch (action) {
 case PM_DEVICE_ACTION_RESUME:
 pm_device_children_action_run(dev, PM_DEVICE_ACTION_TURN_ON, NULL);
 break;
 case PM_DEVICE_ACTION_SUSPEND:
 pm_device_children_action_run(dev, PM_DEVICE_ACTION_TURN_OFF, NULL);
 break;
 case PM_DEVICE_ACTION_TURN_ON:
 /* power up domain */
 break;
 case PM_DEVICE_ACTION_TURN_OFF:
 /* power down domain */
 break;
 default:
 return -ENOTSUP;
 }
 return 0;
}
```