📝 Notes: ESP32, FreeRTOS

1. ESP32 cores and FreeRTOS

- ESP32 has **two CPU cores** (PRO_CPU = core 0, APP_CPU = core 1).
- FreeRTOS runs on both cores → two tasks can **run in parallel**.
- **Task = thread** in FreeRTOS (not a "part of a thread").
- A task has its own stack, state (ready, running, blocked, suspended) and priority level
- Tasks can be:
  - **Pinned** to a core → always run there.
  - **Unpinned** → scheduler may move them between cores.

2. Running on one core

- **ESP-IDF option:** CONFIG_FREERTOS_UNICORE → disables one core, scheduler only on core 0.
- **Otherwise:** pin all tasks to the same core with xTaskCreatePinnedToCore().
- Arduino core: your code usually runs on core 1, system tasks (Wi-Fi/BT) on core 0.

3. BaseType_t

- FreeRTOS standard signed integer type.
- Defined per architecture:
  - ESP32 (32-bit): typedef int BaseType_t;
  - 8-bit MCU: could be char, etc.
- Ensures portability.

Related types

- UBaseType_t → unsigned version (used for priorities, counters).
- TickType_t → type for time/delay values (depends on configTICK_RATE_HZ).

4. Return values (pdPASS, pdFAIL, pdTRUE, pdFALSE)

- Just integer macros for readability + consistency:
- #define pdPASS (BaseType_t) 1

- #define pdFAIL (BaseType_t) 0
- #define pdTRUE (BaseType_t) 1
- #define pdFALSE (BaseType_t) 0
- Used by many APIs:
  - xTaskCreate() → returns pdPASS if the task was created.
  - xQueueSend() → returns pdTRUE on success.
- Benefit: clear meaning across different MCUs.

#if CONFIG_FREERTOS_UNICORE

static const BaseType_t app_cpu = 0;

#else

static const BaseType_t app_cpu = 1;

#endif

In that code snippet:

- **app_cpu** (lowercase) → just a **variable name** the author chose. It means *"the core I'll use for my demo task."*
- In **unicore mode**, app_cpu = 0, which points to the hardware **PRO_CPU (core 0)**.
- In **dual-core mode**, app_cpu = 1, which points to the hardware **APP_CPU (core 1)**.

So:

Code variable	Value	Hardware core it maps to
app_cpu = 0	0	**PRO_CPU (core 0)**
app_cpu = 1	1	**APP_CPU (core 1)**

---

1. Function prefixes

- **v** → function returns **void**
  - Example: vTaskDelay(), vTaskDelete()

- **x** → function returns a **status (BaseType_t), can also be for TickType_t and TaskHandle_t.**
    - Example: xTaskCreate() returns pdPASS or errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY.
- **ux** → function returns **unsigned BaseType_t**
    - Example: uxTaskPriorityGet() (returns an unsigned priority value).
- **pv** → function returns a *pointer (void)**
    - Example: pvPortMalloc() returns a void* pointer to allocated memory.

---

2. Constant / macro prefixes

- **pd** → **portable definition**
    - Example: pdTRUE, pdFALSE, pdPASS, pdFAIL.
    - These are just standardized macros (integers) so code works on any platform.
- **port** → **port-specific** (depends on the MCU/architecture)
    - Example: portTICK_PERIOD_MS depends on configTICK_RATE_HZ.
    - portMAX_DELAY is the max delay representable in ticks.

---

3. Type prefixes

- **BaseType_t** → signed integer, base type (portable).
- **UBaseType_t** → unsigned base type.
- **TickType_t** → tick counter type.
- **TaskHandle_t** → handle type for tasks.

✅ **So in your code:**

- vTaskDelay → "void function that delays a task."
- portTICK_PERIOD_MS → "port-dependent definition for 1 tick in milliseconds."

- pdTRUE/pdFALSE → portable definitions of boolean return values.

---

1. What xTaskCreate does

When you call:

```
xTaskCreate(
   toggleLED,       // Task function
   "Blink",         // Name
   1024,            // Stack size
   &pin,            // <- parameter you pass
   1,               // Priority
   NULL             // Handle
);
```

- FreeRTOS doesn't "unpack" the parameter.
- It simply **stores that pointer** (&pin) and gives it to your task function when it starts running.

---

2. Where you handle it

Inside your task function (toggleLED), you are responsible for turning that void* back into something useful.

So yes — you would put this line **at the start of toggleLED**:

```
void toggleLED(void *parameter) {
   int ledPin = *(int*)parameter;   // get the pin number
   pinMode(ledPin, OUTPUT);         // set as output

   while(1) {
      digitalWrite(ledPin, HIGH);
      vTaskDelay(500 / portTICK_PERIOD_MS);
```

```
        digitalWrite(ledPin, LOW);
        vTaskDelay(500 / portTICK_PERIOD_MS);
    }
}
```

---

3. Important warning

Remember from earlier: the parameter must point to something that stays valid:

- ✅ Global/static variable
- ✅ Dynamically allocated memory (malloc)
- ❌ Local variable in setup() (goes out of scope once setup() ends)

---

configTICK_RATE_HZ

- **Definition:** how many **ticks per second** the RTOS runs at.
- Example:
  - configTICK_RATE_HZ = 1000 → 1000 ticks every second.
  - That means **1 tick = 1 ms**.

---

portTICK_PERIOD_MS

- **Definition:** how many **milliseconds per tick**.
- It's just the inverse of TICK_RATE_HZ.
- Formula:
- portTICK_PERIOD_MS = 1000 / configTICK_RATE_HZ
- Example:
  - If configTICK_RATE_HZ = 1000 → portTICK_PERIOD_MS = 1 ms/tick.
  - If configTICK_RATE_HZ = 100 → portTICK_PERIOD_MS = 10 ms/tick.

✅ **So your statement is correct:**

- TICK_RATE_HZ = ticks per second.
- TICK_PERIOD_MS = milliseconds per tick.

Measuring Task Stack Usage in FreeRTOS

- **Stack size** in xTaskCreate() is given in **words**, not bytes.
  - On ESP32: 1 word = 4 bytes.
  - Example: 1024 words = 4096 bytes.
- **Monitoring stack usage:**
  - uxTaskGetStackHighWaterMark(handle) → returns the **minimum free stack (words)** ever available for a task.
  - vTaskGetInfo(handle, &status, pdTRUE, eInvalid) → detailed info, including status.usStackHighWaterMark.
- **Interpreting results:**
  - Large high-water mark = stack oversized (can shrink).
  - Very small (<50 words on ESP32) = risk of overflow (increase stack).
- **Overflow detection:**
  Enable configCHECK_FOR_STACK_OVERFLOW = 1 or 2 in FreeRTOSConfig.h and implement
- void vApplicationStackOverflowHook(TaskHandle_t xTask, char *pcTaskName);

  → Called automatically if a stack overflow is detected.

📌 **Notes: Variables, Addresses, and Pointers in FreeRTOS (ESP32 / Arduino)**

1. Variables

- Declaring a variable reserves space in RAM and gives it a name.
- TickType_t delay_time = 400;
  - delay_time → variable name
  - value → 400
  - type → TickType_t (on ESP32: uint32_t)

---

2. Addresses

- Every variable lives at a specific RAM location (e.g., 0x3FFB1234).
- &delay_time → "the address of delay_time."
- Type of &delay_time is TickType_t* ("pointer to TickType_t").

---

3. Pointers

- A pointer is a variable whose **value is an address**.
- TickType_t* ptr = &delay_time;

- ptr stores 0x3FFB1234
- type = “pointer to TickType_t”
- *ptr → follows the address and reads/writes the integer stored there.

---

4. Why cast to void*?

- FreeRTOS API is generic:
- xTaskCreatePinnedToCore(TaskFunction_t, ..., void *pvParameters, ...);
- It always expects a void* for parameters.
- Casting (void*)&delay_time:
    - keeps the **same address bits** (0x3FFB1234)
    - just changes the type → void* (generic pointer).

---

5. Using inside the task

- FreeRTOS gives you back the same pointer (pvParameters).
- Cast it back to the right type, then dereference:
- void vBlinkTask(void *pv) {
- TickType_t d_ms = *(TickType_t*)pv; // read integer (400) from address
- TickType_t d = pdMS_TO_TICKS(d_ms);
- for (;;) { ... }
- }

---

6. Common mistake

(void*)delay_time // ❌ passes the VALUE 400 as if it were an address

- This creates a pointer to memory address 0x190 (decimal 400).
- Dereferencing that leads to undefined behavior.

✅ Correct:

(void*)&delay_time // pass the ADDRESS of the variable

---

7. Alternative trick (passing value directly)

- On 32-bit systems like ESP32:
- xTaskCreatePinnedToCore(..., (void*)400, ...);
- Then in task:
- TickType_t d_ms = (TickType_t)pv; // cast value back, no deref
- Works, but less clear and not portable.

---

8. Lifetime matters

- If you pass &delay_time for a **global/static** variable → ✅ safe (lives forever).
- If you pass the address of a **local variable in setup()** → ❌ unsafe (address becomes invalid after setup() exits).

👉 **Mental model:**

- Variable = box in memory with a name + type.
- &variable = address of that box.
- Pointer = variable that stores an address.

- Cast to void* = “forget what’s inside the box for now.”

Core ideas

- **Tick**: hardware timer interrupt (ESP32: typically 1 ms). On each tick, the **scheduler** decides which task runs.
- **Priority first**: Highest priority Ready task runs. Same priority → **round-robin** per tick.
- **Preemption**: When a higher-priority task becomes **Ready**, it can preempt a lower-priority one at the next scheduling point (often the tick).
- **Idle task**: Runs when nothing else is Ready.

Task states

- **Ready** → can run when chosen.
- **Running** → currently executing on the CPU.
- **Blocked** → waiting on time (vTaskDelay) or an event (queue/semaphore). Auto-returns to Ready when unblocked.
- **Suspended** → manually stopped via vTaskSuspend(). Only vTaskResume() moves it back to Ready.

Context switching

- Switching tasks saves CPU registers/PC and uses each task’s **stack** to store context. Ensure **enough stack** at xTaskCreate* (ESP32 stack size is in **words**, so 1024 ⇒ ~4096 bytes).

ISRs vs tasks

- **ISRs** outrank tasks. Keep them **short** and signal tasks using …FromISR API (e.g., give semaphore/queue). Avoid heavy work in ISRs.

Single-core vs multi-core

- ESP32 has 2 cores. To keep behavior predictable in demos, **pin tasks** to one core (xTaskCreatePinnedToCore) or build unicore. Otherwise, two tasks may truly run at once on different cores.

Making preemption visible (serial demo pattern)

- Print **one char at a time** in a lower-priority task and vTaskDelay(1) or taskYIELD() between chars.
- A higher-priority task that wakes periodically (e.g., every 10–100 ms) will **sprinkle** output between characters.
- Slow serial baud (e.g., 300 bps) makes interleaving obvious. Serial.flush() after each char forces blocking (demo-only).