5.4 Deep Dive: ESP32 Hardware Interrupts

Configuring Hardware Timers on the ESP32

The ESP32 microcontroller comes with four general-purpose 64-bit hardware timers. These timers are highly precise and can be used to generate interrupts at specific intervals, independent of the RTOS scheduler.

The configuration involves four main steps:

1. **Initialize the Timer:** `timerBegin(uint8_t num, uint16_t prescaler, bool countUp)`
   - `num`: The timer you want to use (0 to 3).
   - `prescaler`: A value used to divide the base clock (usually 80 MHz). A prescaler of 80 will make the timer count up every 1 microsecond (80,000,000 Hz / 80 = 1,000,000 Hz).
   - `countUp`: true for counting up, false for counting down.
2. **Attach the ISR:** `timerAttachInterrupt(hw_timer_t *timer, void (*fn)(void), bool edge)`
   - This function links your ISR function to the hardware timer.
3. **Set the Alarm Value:** `timerAlarmWrite(hw_timer_t *timer, uint64_t alarm_value, bool autoreload)`
   - This sets the counter value at which the interrupt will be generated. For a 1 MHz timer clock, an alarm_value of 1,000,000 will trigger an interrupt every second.
   - If autoreload is true, the timer will automatically restart after the interrupt, making it periodic.
4. **Enable the Alarm:** `timerAlarmEnable(hw_timer_t *timer)`
   - This starts the timer and enables the interrupt generation.

Writing an Interrupt Service Routine (ISR)

An ISR is a special function that the CPU executes in response to a hardware interrupt.

On the ESP32, it is critical to use the `IRAM_ATTR` attribute in the function definition:

```
void IRAM_ATTR onTimer() {
  // Your interrupt code here...
}
```

`IRAM_ATTR` tells the compiler to place the ISR code into the ESP32's Internal RAM (IRAM). This is essential for performance and reliability. If an ISR is in flash memory, the CPU may have to wait for the flash to be read, which can introduce unacceptable latency and jitter into the interrupt response time.

Example of a complete hardware timer setup:

```
// Timer handle
hw_timer_t *timer = NULL;

// The ISR function to be called
void IRAM_ATTR onTimer() {
  // Toggle an LED or perform a quick action
  digitalWrite(LED_PIN, !digitalRead(LED_PIN));
}

void setup() {
  pinMode(LED_PIN, OUTPUT);

  // 1. Initialize timer 0 with a prescaler of 80
  timer = timerBegin(0, 80, true);

  // 2. Attach the ISR to our timer
  timerAttachInterrupt(timer, &onTimer, true);

  // 3. Set the alarm to trigger every 1,000,000 counts (1 second)
  timerAlarmWrite(timer, 1000000, true);

  // 4. Enable the alarm
  timerAlarmEnable(timer);
}
```

The Golden Rules of ISRs

Interrupt Service Routines are powerful but dangerous if used incorrectly. Because they can interrupt any part of your code at any time, they must follow strict rules to avoid crashing the system.

1. **Keep it Fast:** An ISR must execute as quickly as possible. The longer an ISR runs, the more it delays the main program and other interrupts. The best ISRs do the absolute minimum work required, such as setting a flag or sending data to a queue, and then exit.

Defer all complex data processing to a regular FreeRTOS task.

2. **Never Block:** An ISR **must never, ever block**. This means you cannot call functions like `vTaskDelay()`, `delay()`, or wait for a semaphore, mutex, or queue. The system is in a special interrupt context, and attempting to block will lead to a system crash.
3. **Use ISR-Safe API Functions:** When you need to interact with FreeRTOS from an ISR (for example, to signal a task), you **must** use the special ISR-safe version of the API function. These functions are specially designed to be non-blocking and safe to call from an interrupt context. They are easily recognizable as they all end with the suffix `...FromISR()`.
   - **Correct:** `xSemaphoreGiveFromISR()`
   - **Incorrect:** `xSemaphoreGive()`
   - **Correct:** `xQueueSendFromISR()`
   - **Incorrect:** `xQueueSend()`
4. **Be Careful with Global Variables:** If an ISR modifies a global variable that is also accessed by a task, you must protect that variable to prevent data corruption (a "race condition"). The primary method for this is using a **critical section**, which will be discussed in the next part.

Revision #1
Created 2025-08-28 12:54:24 UTC by GI
Updated 2025-08-28 12:57:26 UTC by GI