Mastering Execution Time in Hard Real-Time Systems: Why Every Microsecond Counts

In **safety-critical embedded systems**—from insulin pumps and patient monitors to flight-control computers and industrial robots—**missing a deadline isn't just a bug, it's a system failure** with potentially catastrophic consequences. To prevent this, you must guarantee **predictable, deterministic behavior**, meaning for any given inputs and initial state, both the output and its timing are always identical.

At the heart of that predictability lies **Worst-Case Execution Time (WCET) analysis**—determining the **absolute maximum time** any code path can take on your chosen hardware. **WCET is the cornerstone** for demonstrating that every task will finish **within its strict deadline**, even under the worst conditions. Without **reliable, high-resolution timing data**, you **cannot** meet the stringent safety standards (e.g., **DO-178C** in aerospace or **ISO 26262** in automotive) nor provide the deterministic scheduling demanded by real-time operating systems.

Yet achieving **microsecond** (or even **nanosecond**) precision across a wide variety of MCUs—STM32 F4/F7/G4/H7 families, NXP, TI, Renesas, etc.—poses a real challenge. Each device differs in clocks, bus architectures, peripheral sets and debugging features (like the **Data Watchpoint and Trace** unit). Add in your software environment—bare-metal versus complex RTOS with context switches and interrupt preemption—and timing variability balloons.

The solution? A **hardware-aware profiling framework** that:

1. **Abstracts** low-level timer details behind a **consistent API**, so application code never needs to touch registers directly.

2. **Detects** and leverages the **best available timing source** on each target (SysTick, DWT cycle counter, high-precision timers, etc.).

3. **Integrates cleanly** with both bare-metal loops and RTOS kernels, accounting for context-switch and interrupt overhead.

With this approach, your timing measurements remain **flexible**, **reusable across projects**, and precise to the **microsecond—or even nanosecond—level** wherever your safety-critical code runs.

Clocking Code: Measuring Execution Time on Microcontrollers

In real-time embedded systems, knowing exactly how long a block of code takes is **crucial**. Whether you're fine-tuning a motor control loop, optimizing interrupt latency, or verifying timing for safety-critical logic, **measuring execution time precisely is essential**.

However, **no single method works everywhere**. Different MCUs offer different capabilities, and you must choose the right one based on what’s available and what level of accuracy you need.

---

1. DWT Cycle Counter – The Most Precise (Cortex-M3/M4/M7/H7)

What it is: A built-in cycle counter that increments on every CPU clock. Found in the Debug Unit of many Cortex-M cores (except M0/M0+).

Why use it: Perfect for ultra-short code segments where you need **cycle-level precision** (e.g., profiling an ISR or DSP loop).

Example:

```
void DWT_Init(void) {
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
    DWT->CYCCNT = 0;
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
}

DWT_Init();
uint32_t start = DWT->CYCCNT;

critical_function();

uint32_t end = DWT->CYCCNT;
uint32_t time_us = (end - start) / (SystemCoreClock / 1000000);
```

Caution:

- Only available on higher-end Cortex-M cores.
- Can overflow during long measurements.
- Not always enabled after reset — must be initialized.

---

2. General-Purpose Timers (TIMx) – Flexible and Widely Available

What it is: Built-in timers (like TIM2) found on nearly every MCU. Can be configured for microsecond or even sub-microsecond resolution.

Why use it: Ideal when DWT is not available. Great for measuring longer intervals or peripheral behavior.

Example:

```
__HAL_TIM_SET_COUNTER(&htim2, 0);
HAL_TIM_Base_Start(&htim2);

uint32_t start = __HAL_TIM_GET_COUNTER(&htim2);

your_function();

uint32_t end = __HAL_TIM_GET_COUNTER(&htim2);
uint32_t elapsed_us = end - start;
```

Caution:

- You must ensure the timer is configured correctly (prescaler, clock).
- Needs overflow handling for long durations.
- Resolution depends on how you configure the timer clock.

---

3. SysTick / HAL_GetTick – Simple and Coarse

What it is: A basic system tick counter available on every Cortex-M core. Typically used for millisecond delays and RTOS timing.

Why use it: Convenient for coarse timing (e.g., delays, loops that run for >1ms). Useful in HAL-based projects without adding custom timers.

Example:

```
uint32_t t1 = HAL_GetTick();
slow_function();
uint32_t t2 = HAL_GetTick();
printf("Elapsed: %lu ms\n", t2 - t1);
```

Caution:

- Resolution limited to milliseconds (unless you use raw SysTick).
- Not accurate for short or fast code.
- Delayed if SysTick is paused (e.g., during low-power modes)

4. GPIO Toggle + Oscilloscope – External Validation

What it is: Toggling a GPIO pin at start and end, and measuring pulse width using an oscilloscope or logic analyzer.

Why use it: Best when you want to **verify timing externally** or visualize jitter in real time.

Example:

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
time_sensitive_function();
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
```

Caution:

- Requires access to a scope or analyzer.
- Introduces minimal delay due to GPIO write.
- Less useful for measuring very short internal code segments (like math loops).

Arduino Comparison (Optional Concept Bridge)

If you're used to Arduino:

- `millis()` is similar to HAL_GetTick (millisecond resolution).
- `micros()` is like using a hardware timer (microsecond resolution).

These are **convenient abstractions**, but they don't give you full control over precision and hardware like DWT or custom timers.

Cycle-Accurate Profiling on Cortex-M: SSD1306 Update Measured in Real Time

In hard-real-time systems, Here's how we measured an SSD1306 O-LED display repaint with **sub-µs precision** on an STM32H7 using the DWT cycle counter:

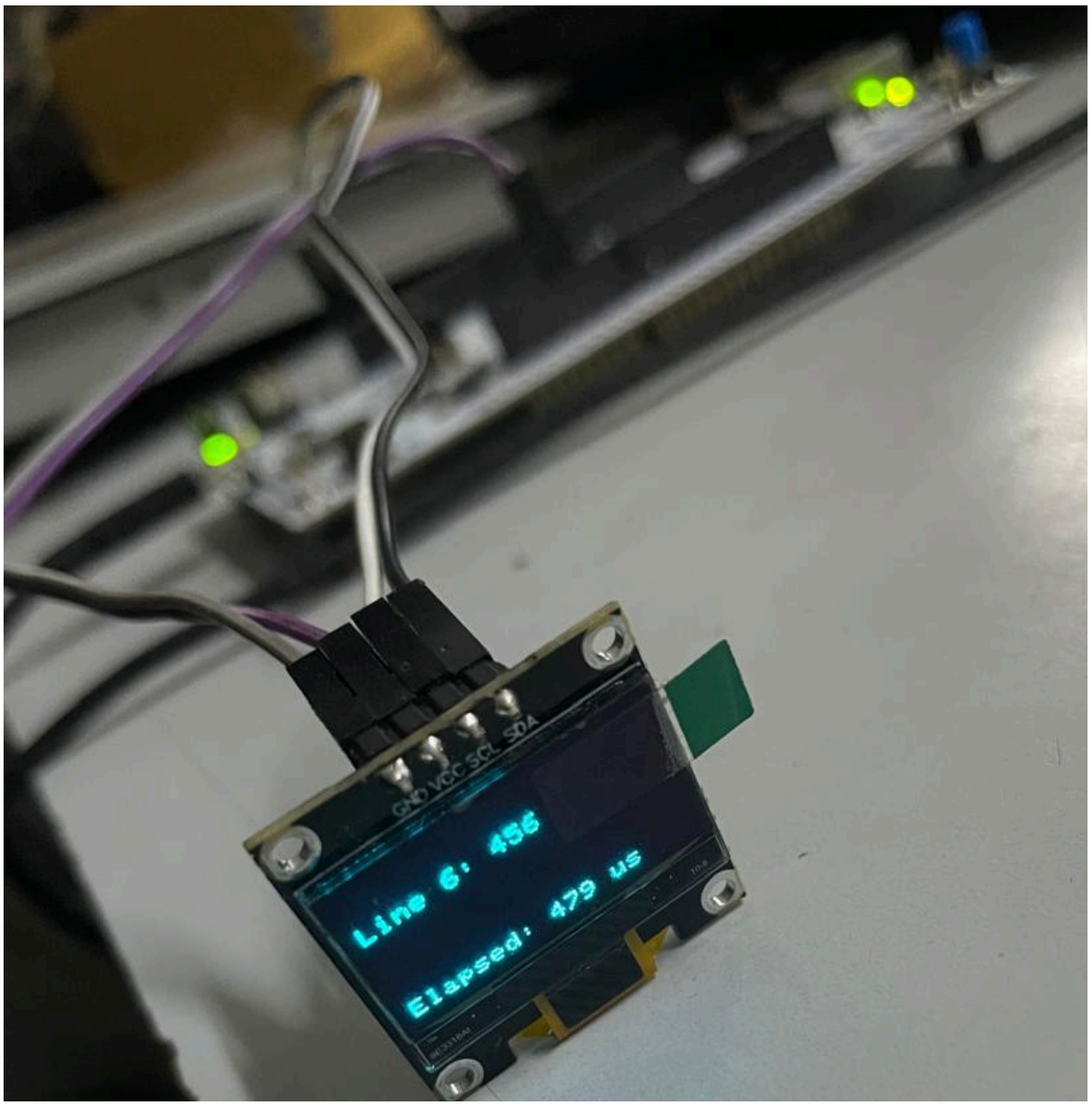
```
// Initialize DWT
DWT_Init();
```

```
/**
  * @brief Initialize DWT for cycle counting.
  */
void DWT_Init(void)
{
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;  // Enable DWT
    DWT->CYCCNT = 0;                                 // Reset cycle counter
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;             // Enable cycle counter
}
```

```
// Start measuring time
uint32_t start_cycles = DWT->CYCCNT;
// Code to measure
display_line_8("Line 8: 789");
display_line_7("Line 7: Example");
display_line_6("Line 6: 456");
display_line_5("Line 5: Display");
SSD1306_display_repaint();
// Stop measuring time
uint32_t end_cycles = DWT->CYCCNT;

// Calculate elapsed time in microseconds
uint32_t elapsed_us = (end_cycles - start_cycles) / (SystemCoreClock / 1000000);

// Display elapsed time on line 1
char elapsed_string[32];
snprintf(elapsed_string, sizeof(elapsed_string), "Elapsed: %lu us", elapsed_us);
display_line_1(elapsed_string);
SSD1306_display_repaint();  // Refresh display
```



This photo shows our STM32H7 driving a 128×64 SSD1306 OLED over I²C while simultaneously measuring the repaint time with the DWT cycle counter.

- **Top line ("Line 6: 456")** is one of the display buffers we wrote via `display_line_6()`.
- **Bottom line ("Elapsed: 479 µs")** is the measured update latency—under half a millisecond for the full screen refresh.
- **Wiring** at the top: GND, VCC, SCL, SDA hooked to the MCU's I²C pins.
- **Background**: you can see the STM32H7 eval board with its status LEDs lit, confirming the MCU is running at 550 MHz.

This validates that our repaint routine completes in under 0.5 ms, giving confidence in tight, deterministic UI updates for hard real-time applications.

Key Takeaways:

- **Cycle-Accurate**: ~1 CPU-cycle resolution on Cortex-M3/M4/M7/H7.
- **Minimal Overhead**: Bracket only the code under test—no hidden calls.
- **Validation**: Always cross-check with a GPIO toggle + oscilloscope for jitter analysis.

Why This Matters

- **Deterministic WCET**: Prove worst-case bounds for safety-critical tasks.
- **Optimization Insight**: Pinpoint latency hotspots in drivers, ISRs, and comms stacks.
- **Portable Framework**: Swap in TIMx or SysTick fallbacks on MCUs without DWT support.

Calculation Formula

cycles = CYCCNT_end - CYCCNT_start

elapsed_us = cycles / (SystemCoreClock / 1_000_000)

= cycles / CPU_MHz

Example (@550 MHz):

• 1 cycle ≈ 1.82 ns

• 550 cycles ≈ 1 µs

• cycles < 550 ⇒ displayed as 0 µs (sub-µs)

- Any `cycles < 550` → **0 µs** displayed ⇒ **sub-µs** execution!
