Mastering Time: A Deep Dive into STM32 Timers

In the world of embedded systems, time is everything. From blinking an LED to controlling a high-speed motor, precise timing is the cornerstone of almost every application. In this lesson, we will explore the fundamental concepts of time management and dive deep into the powerful and versatile timer peripherals offered by STM32 microcontrollers.

The Two Faces of Time in Embedded Systems

In any system, we care about time in two fundamental ways:

- **Absolute Time (or "Wall Time"):** This is time as we humans perceive it—the current date and clock time (e.g., "September 29, 2025, 2:15 PM"). It's essential for applications that need to log events with timestamps, schedule tasks at specific times of the day, or display a calendar to the user.
- **Relative Time:** This refers to the duration or interval between events. It is the workhorse of embedded control systems. We use relative time to:
  - Measure the duration between two events (e.g., the time between two rising edges of a sensor signal).
  - Schedule a task to run after a specific delay.
  - Generate precise, blocking delays in code.
  - Create periodic interrupts to run tasks at fixed intervals (e.g., reading an ADC every 1ms).

The STM32 Time Management Toolkit

STM32 microcontrollers provide a rich set of peripherals dedicated to time management, each with a specific purpose:

Peripheral	Primary Use Case
RTC (Real-Time Clock)	Keeping track of **absolute** calendar time, even when the main CPU is in a low-power state.
SysTick Timer	A simple, 24-bit timer built into the ARM Cortex-M core, primarily used by an RTOS for its system "tick".
General, Basic, & Advanced Timers	The highly versatile workhorses for all **relative** time tasks (delays, PWM, input capture, etc.).
Watchdog Timers (WDT)	A safety mechanism that resets the MCU if the main software "hangs" or crashes.
Debug Watch Timer (DWT)	A high-resolution counter in the core's debug unit, often used for precise performance profiling.

For this lesson, we will focus on the most versatile and commonly used peripherals: the **general-purpose timers**.

Core Concepts: From Counters to Timers

At its heart, a timer is a simple concept built upon a digital counter.

- **Counter:** A digital circuit that counts incoming electrical pulses or "edges". It can be configured to count up from 0 to a maximum value (`MAX`) or count down.
- **Timer:** A timer is simply a counter that is fed a clock signal with a **known, fixed frequency**. Because we know the time between each clock pulse, we can use the counter's value to measure time.

The Fundamental Timing Equation

Let's establish the relationship between clock frequency, counts, and time.

- A clock with a frequency **F** produces **F** pulses per second.
- The period **T** of one clock pulse is **T = 1 / F** seconds.
- If we count **N** clock pulses, the total elapsed time **t** is: `t = N * T = N / F`

Conversely, if we want to measure a specific time duration **t**, the number of clock pulses **N** we need to count is: `N = t * F`

Example:

- If our timer's clock is **1 MHz (1,000,000 pulses/sec)** and we count **5,000,000 pulses**, the elapsed time is: `t = 5,000,000 / 1,000,000 = 5 seconds`.
- To generate a delay of **2 seconds** with the same 1 MHz clock, we need to count: `N = 2 * 1,000,000 = 2,000,000` pulses.

Introducing the Prescaler

Microcontroller clocks often run at very high frequencies (e.g., 72 MHz). Counting every single pulse would cause our timer to overflow very quickly and would limit the maximum delay we can generate.

To solve this, timers include a **Prescaler (PSC)**. The prescaler is a divider that slows down the clock before it reaches the main counter.

- The effective timer clock frequency becomes: `Timer_Clock = Peripheral_Clock / (PSC + 1)`
- Our timing equation is now: `t = N / (F / (PSC + 1))`

The prescaler allows us to "zoom out" our time measurement, enabling us to generate much longer delays with the same size counter register.

An Overview of STM32 Timers

STM32 MCUs offer a variety of timers, categorized by their features and complexity.

Timer Type	Bit Width	Key Features
Basic Timers (TIM6, TIM7)	16-bit	Time-base generation only; no input/output channels. Perfect for simple periodic interrupts.
General Purpose Timers (e.g., TIM2-TIM5)	16-bit or 32-bit	The versatile workhorses. Support time-base, input capture, output compare, PWM generation, and more.
Advanced Control Timers (TIM1, TIM8)	16-bit or 32-bit	All the features of GP timers, plus advanced motor control features like complementary outputs with dead-time insertion and a break input for safety.

All of these timers can be used to perform the most fundamental task: **Time-Base Generation**.

Time-Base Generation

A time-base generator is used to create a precise delay or a periodic interrupt. It relies on three key registers:

- `PSC` **(Prescaler Register):** Divides the main peripheral clock.
- `ARR` **(Auto-Reload Register):** Defines the "top" value the counter will count to.
- `CNT` **(Counter Register):** The actual up/down counter.

When the timer is enabled, the `CNT` register increments at the prescaled clock frequency. When `CNT` reaches the value in `ARR`, it triggers an **Update Event (UEV)** and automatically resets back to 0 to start counting again. This update event is what we use to signal that our desired time has elapsed.

We can use this update event in two ways: polling or interrupts.

Method 1: Polling for Delays (Blocking)

In this method, we start the timer and then get stuck in a loop, constantly checking a status flag until the time has elapsed. This is simple but **inefficient**, as it blocks the CPU from doing any other work.

Polling Implementation Steps:

1. **Initialize Timer:**
   - Enable the timer's peripheral clock via the RCC registers.
   - Set the **Prescaler (`PSC`)** register to get the desired counter clock tick rate. For example, to get a 1ms tick from a 16 MHz clock, `PSC` would be `16000 - 1`.
2. **Start Delay Function:**
   - Set the **Auto-Reload (`ARR`)** register to the number of ticks you want to wait (e.g., `1000` for a 1000ms delay).
   - Reset the **Counter (`CNT`)** to 0.
   - Start the timer by setting the `CEN` bit in the **Control Register 1 (`CR1`)**.
3. **Wait (Poll):**
   - Enter a `while` loop that continuously checks the **Update Interrupt Flag (`UIF`)** in the **Status Register (`SR`)**.
4. **Cleanup:**
   - Once the `UIF` flag is set (meaning the timer has overflowed), clear the flag by writing 0 to it.
   - (Optional) Stop the timer by clearing the `CEN` bit.

Bare-Metal Code Example (Polling):

```
// Assumes TIM_PSC is defined to be 16000 for a 1ms tick from a 16MHz clock
void TimerInit(void) {
    // 1. Enable timer clock (e.g., for TIM6 on APB1)
    RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
    // 2. Set timer prescaler
    TIM6->PSC = TIM_PSC - 1; // Formula is PSC+1, so we subtract 1
}

// Generates a blocking delay for a given number of milliseconds
void TimerDelayMs(uint32_t ms) {
    // 1. Set the auto-reload value (the number of ticks to count)
    TIM6->ARR = ms;
    // 2. Reset the counter to start from 0
    TIM6->CNT = 0;
    // 3. Start the timer
```

```
    TIM6->CR1 |= TIM_CR1_CEN;

    // 4. Poll the update flag until it is set
    while(!(TIM6->SR & TIM_SR_UIF)) {
        // CPU is stuck here doing nothing useful
    }

    // 5. Clear the update flag for the next use
    TIM6->SR &= ~TIM_SR_UIF;

    // (Optional) Stop the timer to save power
    // TIM6->CR1 &= ~TIM_CR1_CEN;
}
```

Method 2: Using Interrupts (Non-Blocking)

This is the far more powerful and efficient method. We configure the timer and then let the CPU go do other work. When the time has elapsed, the timer will automatically trigger an interrupt, forcing the CPU to jump to our handler function.

Interrupt Implementation Steps:

1. **Initialize Timer:**
    - Perform the same clock and prescaler setup as the polling method.
    - Set the **Auto-Reload (`ARR`)** value for the desired periodic interval.
    - Enable the **Update Interrupt Enable (`UIE`)** bit in the **Interrupt Enable Register (`DIER`)**.
    - Enable the corresponding interrupt line in the **NVIC (Nested Vectored Interrupt Controller)**.
    - Start the timer by setting the `CEN` bit in `CR1`.
2. **Implement the Interrupt Handler (ISR):**
    - Create a function with the exact name specified in the device's vector table (e.g., `TIM6_DAC_IRQHandler`).
    - Inside the handler, first check that the `UIF` flag is indeed set.
    - **Crucially, clear the `UIF` flag.** If you don't, the CPU will exit the ISR and immediately re-enter it, getting stuck in an infinite loop.
    - Perform the short, time-critical task (e.g., toggle an LED, increment a counter).

Bare-Metal Code Example (Interrupt):

```
// Global variable to be updated by the ISR
volatile uint32_t g_system_ticks = 0;

// Initialize the timer to generate an interrupt every 'ms' milliseconds
void TimerInit_Interrupt(uint32_t ms) {
    // 1. Enable timer clock
    RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
    // 2. Set prescaler for a 1ms tick
    TIM6->PSC = TIM_PSC - 1;

    // 3. Set the auto-reload value for the desired period
    TIM6->ARR = ms;
    // 4. Reset counter
    TIM6->CNT = 0;

    // 5. Enable the update interrupt in the timer peripheral
    TIM6->DIER |= TIM_DIER_UIE;

    // 6. Enable the timer's interrupt in the NVIC
    NVIC_EnableIRQ(TIM6_DAC_IRQn);

    // 7. Start the timer!
    TIM6->CR1 |= TIM_CR1_CEN;
}

// The Interrupt Service Routine for TIM6
void TIM6_DAC_IRQHandler(void) {
    // Check if the update interrupt flag is the source of the interrupt
    if (TIM6->SR & TIM_SR_UIF) {
        // IMPORTANT: Clear the flag to prevent re-entry
        TIM6->SR &= ~TIM_SR_UIF;

        // Do the periodic work
```

```
        g_system_ticks++;
        // Example: a non-blocking LED toggle
        // LED_Toggle(LED_PIN);
    }
}

// Main application
int main(void) {
    // ... other initializations ...

    // Start a 100ms periodic interrupt
    TimerInit_Interrupt(100);

    while(1) {
        // The CPU is now free to do other tasks here.
        // For example, check the g_system_ticks variable
        // to perform actions at larger intervals.
    }
}
```

STM32 Timer - Output Compare Mode

Output Compare mode allows you to continuously compare the value of the timer's main counter (`CNT`) against a pre-loaded value in a dedicated **Capture/Compare Register (`CCRx`)**. When the counter's value matches the compare register's value, a specific action can be triggered.

This action can be:

1. **Generating an interrupt or a DMA request.**
2. **Changing the state of a physical output pin.**

This second capability is incredibly powerful, allowing the timer to directly control external hardware with microsecond precision, without any CPU intervention.

How Output Compare Works

The process is straightforward:

1. The timer's counter (`CNT`) is configured to count up to the value stored in the **Auto-Reload Register (`ARR`)**.
2. You load a value into one of the timer's **Capture/Compare Registers (`CCRx`)**. Each general-purpose timer typically has four of these channels (CCR1, CCR2, CCR3, CCR4).
3. As the `CNT` register increments, the hardware continuously compares its value to `CCRx`.
4. When `CNT == CCRx`, a "compare match" event occurs, and the timer performs a pre-configured action on the associated output pin (e.g., set it high, set it low, or toggle it).

Configuration Steps for Output Compare

Let's follow the steps from the user manual to configure a timer channel to toggle an output pin on a compare match.

1. **Select the Clock Source:** As with the time-base, configure the timer's clock source and prescaler (`PSC`) to define the tick rate of the counter.
2. **Set the Period (ARR) and Compare (CCR) Values:**
   - Write the maximum count value to the `TIMx_ARR` register.
   - Write the desired compare match value to the `TIMx_CCRx` register. This value must be less than or equal to the `ARR` value.
3. **Configure the Channel as an Output:** In the `TIMx_CCMRx` register, ensure the `CCxS` bits for the channel are set to `00`, which configures it as an output. (This is the default).
4. **Select the Output Mode:** This is the most important step. In the `TIMx_CCMRx` register, you set the `OCxM` bits to define the action on compare match. For example, to **toggle the output pin**, you set `OCxM = 011`.
5. **Enable the Output:** In the `TIMx_CCER` register, you must set the `CCxE` bit to enable the timer channel's output and connect it to the physical GPIO pin.
6. **Enable the Counter:** Finally, set the `CEN` bit in the `TIMx_CR1` register to start the timer.

Bare-Metal Code Example: Multi-Channel Toggle

This code configures `TIM4` with four channels, each set to toggle an output pin at different points in the timer's period. This would generate four different square waves with different duty cycles.

```
// Assumes TIM_PSC is defined to give a 1ms tick (e.g., 16000 for a 16MHz clock)
void Timer_OutputCompare_Init(void) {
    // Note: GPIO pin configuration for alternate function mode is required here!
    // (This step is shown in the PWM example below)

    // 1. Enable Timer4 clock
```

```
    RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;

    // 2. Set prescaler to achieve a 1ms counter tick
    TIM4->PSC = TIM_PSC - 1;
    // 3. Set the period to 500ms (the counter will count from 0 to 499)
    TIM4->ARR = 500 - 1;

    // 4. Set the compare values for each of the 4 channels
    TIM4->CCR1 = 100 - 1; // Match at 100ms
    TIM4->CCR2 = 200 - 1; // Match at 200ms
    TIM4->CCR3 = 300 - 1; // Match at 300ms
    TIM4->CCR4 = 400 - 1; // Match at 400ms

    // 5. Configure all 4 channels to "Toggle on Compare Match" mode (OCxM = 011)
    // For Channels 1 and 2 in CCMR1
    TIM4->CCMR1 |= (TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1M_0);
    TIM4->CCMR1 |= (TIM_CCMR1_OC2M_1 | TIM_CCMR1_OC2M_0);
    // For Channels 3 and 4 in CCMR2
    TIM4->CCMR2 |= (TIM_CCMR2_OC3M_1 | TIM_CCMR2_OC3M_0);
    TIM4->CCMR2 |= (TIM_CCMR2_OC4M_1 | TIM_CCMR2_OC4M_0);

    // 6. Enable the output for all 4 channels in the Capture/Compare Enable Register
    TIM4->CCER |= TIM_CCER_CC1E | TIM_CCER_CC2E | TIM_CCER_CC3E | TIM_CCER_CC4E;

    // 7. Enable the master timer counter to start everything
    TIM4->CR1 |= TIM_CR1_CEN;
}
```

STM32 Timer - Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is a powerful technique that leverages Output Compare mode to control the average power delivered to a device. It works by generating a square wave with a fixed frequency (**Period**) and varying the amount of time the signal is HIGH versus LOW (**Duty Cycle**).

This is the fundamental technology used for:

- Dimming LEDs
- Controlling the speed of DC motors
- Positioning servo motors
- Generating analog voltage levels via a filter

How PWM Works

In PWM mode, the Output Compare hardware is used to create the pulse automatically.

1. The timer counts up from 0 to the `ARR` value. This defines the **Period** of the PWM signal.
2. When the timer starts, the output pin is forced HIGH.
3. When the counter value `CNT` matches the compare value `CCR`, the output pin is forced LOW.
4. When the counter overflows (reaches `ARR`), it resets to 0, and the output is forced HIGH again, starting a new cycle.

By changing the `CCR` value, we change the point at which the signal goes low, thus modulating the width of the pulse and controlling the duty cycle.

- **PWM Period** is determined by the timer clock and the `ARR` value. `PWM Frequency = Timer_Clock / (ARR + 1)`
- **PWM Duty Cycle** is determined by the ratio of `CCR` to `ARR`. `Duty Cycle % = (CCR / (ARR + 1)) * 100`

Configuration Steps for PWM

Generating PWM requires configuring both the timer and the GPIO pin you want the signal to appear on.

1. **Configure GPIO Pin:** The timer needs to take control of a physical pin. You must enable the GPIO port's clock and configure the specific pin to its **Alternate Function (AF)** mode, selecting the correct AF number for the desired timer output.
2. **Setup PWM Frequency (Period):** Configure the timer's `PSC` and `ARR` registers to produce the desired PWM frequency. A common range is 10-100 kHz.
3. **Setup PWM Duty Cycle (Compare Value):** Set the initial duty cycle by writing a value to the `TIMx_CCRx` register.
4. **Select PWM Mode:** In the `TIMx_CCMRx` register, set the `OCxM` bits to `110` (PWM Mode 1) or `111` (PWM Mode 2). In Mode 1 (the most common), the output is HIGH as long as `CNT < CCR` and LOW otherwise.
5. **Enable Preload Registers:** This is a crucial step for glitch-free PWM. Set the `OCxPE` bit in `CCMRx` and the `ARPE` bit in `CR1`. This ensures that new values written to `CCR` and `ARR` are only applied at the end of a PWM cycle (at the update event), preventing malformed pulses.
6. **Enable Outputs:** Set the `CCxE` bit in `CCER` to enable the channel output. For Advanced Timers (TIM1/TIM8), you must also set the master **Main Output Enable (`MOE`)** bit in the `TIMx_BDTR` register.
7. **Enable Counter:** Set the `CEN` bit in `CR1` to start PWM generation.

Bare-Metal Code Example: Fading LED with PWM

This code configures `TIM8` to generate a 10 kHz PWM signal on pin `PC6`. The `main` loop then continuously changes the `CCR1` register to create a smooth LED fading effect.

```
#define TIM_PSC 16 // For a 1MHz Timer Clock from a 16MHz PCLK

void PWM_Init(void) {
    // --- 1. GPIO Configuration (PC6) ---
    // Enable clock for GPIOC
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOCEN;
    // Configure PC6 for Alternate Function mode
    GPIOC->MODER |= (2 << 6*2); // Set bits for pin 6 to 10 (AF mode)
    // Set PC6's alternate function to AF3 (TIM8_CH1)
    GPIOC->AFR[0] |= (3 << GPIO_AFRL_AFSEL6_Pos);

    // --- 2. Timer Configuration (TIM8) ---
    // Enable Timer8 clock (Advanced Timer on APB2)
    RCC->APB2ENR |= RCC_APB2ENR_TIM8EN;

    // Set prescaler to get 1MHz timer clock
    TIM8->PSC = TIM_PSC - 1;
    // Set period to 100 ticks for a 10kHz PWM frequency (1MHz / 100 = 10kHz)
    TIM8->ARR = 100 - 1;
    // Set initial duty cycle to nearly 0%
    TIM8->CCR1 = 1 - 1;

    // --- 3. PWM Mode Configuration ---
    // Select PWM Mode 1 (output is HIGH when CNT < CCR1)
    TIM8->CCMR1 |= TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1;

    // Enable preload for CCR1 and ARR for glitch-free updates
    TIM8->CCMR1 |= TIM_CCMR1_OC1PE;
    TIM8->CR1 |= TIM_CR1_ARPE;
```

```
    // --- 4. Enable Outputs ---
    // Enable Capture/Compare Channel 1 output
    TIM8->CCER |= TIM_CCER_CC1E;
    // Enable the main output for the advanced timer
    TIM8->BDTR |= TIM_BDTR_MOE;

    // --- 5. Start Timer ---
    TIM8->CR1 |= TIM_CR1_CEN;
}

int main(void) {
    int duty;
    SystemInit();
    PWM_Init();

    while(1) {
        // Fade In: Gradually increase duty cycle from 1% to 100%
        for(duty = 1; duty <= 100; duty++) {
            // Update the compare register to change the duty cycle
            TIM8->CCR1 = duty;
            DelayMs(50); // Wait briefly to see the change
        }

        // Fade Out: Gradually decrease duty cycle from 100% to 1%
        for(duty = 100; duty >= 1; duty--) {
            TIM8->CCR1 = duty;
            DelayMs(50);
        }
    }
}
```