ARM Microcontroller Internship Programme

Trainer: *Kiran Jaybhave*

Sunbeam – Industrial Training Programme

Sunbeam Pune (Hinjawadi)

Timer In STM32

- **Time Management in Embedded Systems**
  - **Why Time Matters**
    - In the world of embedded systems, **time is everything**.
    - From blinking a simple LED to controlling a high-speed motor, **precise timing** is the foundation of almost every embedded application.
- **The Two Faces of Time in Embedded Systems**
  - Embedded systems deal with time in **two fundamentally different ways**:

  1. **Absolute Time (Wall Time)**
     - Represents real-world **date and clock time**
     - **Example**: *September 29, 2025 – 2:15 PM*
     - **Used in applications such as:**
       - Data logging with timestamps
       - Scheduling tasks at specific times
       - Clocks, calendars, and monitoring systems

2. Relative Time

- Represents the **time interval between events**
- Forms the backbone of **embedded control systems**
- **Used to:**
  - Measure duration between two events
    - e.g., time between two rising edges of a signal
  - Schedule a task after a specific delay
  - Generate precise delays in software
  - Create **periodic interrupt**
    - e.g., reading a sensor every 1 ms

- **Absolute time tells us what time it is.**
- **Relative time tells us when something should happen.**

STM32 Time Management Toolkit

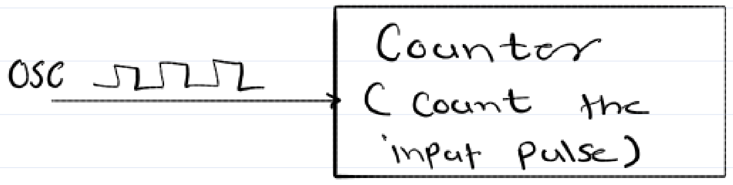
- STM32 microcontrollers provide a **rich set of time-management peripherals**, each designed for a **specific purpose** in embedded systems.
- Different timing problems require **different timer peripherals**.
- STM32 provides **multiple timing tools** to handle accuracy, power, safety, and performance needs.
- **Time-Related Peripherals in STM32**
    - **RTC (Real-Time Clock)**
        - Used to maintain **absolute calendar time** (date and time)
        - Continues running even when the MCU is in **low-power or standby mode**
        - Commonly used in:
            - Clocks and calendars
            - Data logging with timestamps
    - **SysTick Timer**
        - A **24-bit timer built into the ARM Cortex-M core**
        - Used to generate the **system tick**
        - Commonly used by:
            - HAL library
            - RTOS for task scheduling and delays

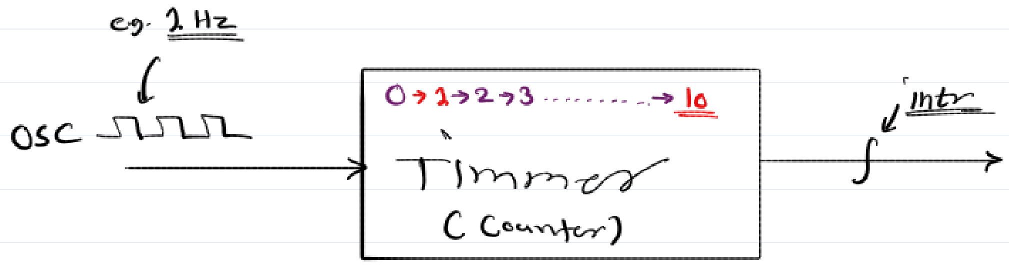
- **General, Basic & Advanced Timers**
  - These timers are the **main hardware timers** used for measuring and generating time intervals
  - Used for **relative time operations** in embedded systems
  - **Typical applications include:**
    - Software delays
    - Periodic interrupts
    - PWM generation
    - Signal measurement
      - **(Input Capture / Output Compare)**
- **Watchdog Timers (WDT)**
  - Used as a **safety mechanism**
  - Automatically **resets the MCU** if the software:
    - Hangs
    - Enters an infinite loop
    - Becomes unresponsive
- **Debug Watch Timer (DWT)**
  - A **high-resolution cycle counter** inside the ARM core's debug unit
  - Used for:
    - Precise timing measurements
    - Performance profiling
    - Accurate microsecond delays

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 (also called ***edges***)
    - Can be configured to:
      - Count **up** from 0 to a maximum value.
      - Count **down** from a maximum value to 0.
    - By itself, a counter **only counts events**, not time.
  - **Timer**
    - A **timer is a counter connected to a clock source**
    - The clock has a **known and fixed frequency**
    - Since the time between clock pulses is known:
      - The counter value can be used to **measure time**
      - Timers can generate **delays and periodic events**

door open ← events
motion sensor ←





84 MHz

Freq = Num of clock pulses / sec.

period of a clock $T = \frac{1}{F} = \frac{\text{sec}}{\text{pulses}}$

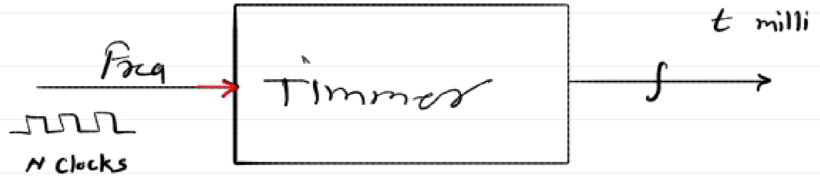
Time for N clocks = $N * T$

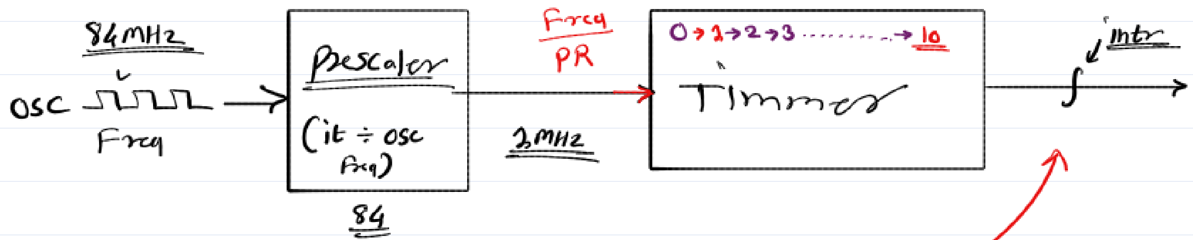
$= \frac{N}{F}$ sec

Num of clocks for t sec, $t = \frac{N}{F}$

$N = t * F$

Num of clocks for t millisec, $N = \frac{t}{1000} * f$





for t ms delay $N = \frac{t}{1000} * \frac{F}{PR}$

The Fundamental Timing Equation

- To understand how timers measure time, we need to relate **clock frequency**, **counts**, and **time**.
  - **Clock Frequency**
    - A clock with frequency **F** produces **F pulses per second.**
    - Unit: **Hertz (Hz)**
    - **E.g. : If freq. is 1 Hz i.e. 1cycle/sec**
  - **Clock Period**
    - Time for one clock pulse is called the **period.**
    - Represented by **T**
      - T = (1 / F) sec $T \Rightarrow \frac{1}{F_{clk}}$;
  - **Measuring Time Using a Counter**
    - If a timer counts **N clock pulses .**
    - Total elapsed time **time** is: $\rightarrow N \times T$
      - Time = N * T = N / F $\Rightarrow N \times \frac{1}{F} \Rightarrow \frac{N}{F}$
  - **Required Count for a Given Time**
    - If a delay of **t seconds** is required
    - **Number of pulses** to count is:
      - N = time * F

- **Example**
  - **Timer clock = 1 MHz (10,00,000 pulses/sec)**
    - **If we Counting 5,000,000 pulses**

      T = 5,000,000 pulses

      Time = 50,00,000 / 10,00,000

      = 5 seconds
    - **To generate a 2-second delay**

      N = 2 × 1,000,000

      = 2,000,000 pulses

      = 2 sec

Introducing the Prescaler

- **Problem Without a Prescaler**
  - Microcontrollers run at **very high clock speeds** (for example: **72 MHz**).
  - This means the timer receives **millions of clock pulses per second**
  - The counter reaches its maximum value **very fast**
  - As a result, we can generate only **very small delays**
- To solve this, timers include a Prescaler (PSC). The prescaler is a divider that slows down the clock before it reaches the main counter.
- **How the Prescaler Works**
  - Peripheral clock → **Prescaler** → Timer counter
  - The **prescaler divides** the **clock** by **(PSC + 1)**
    - **Timer Clock = Peripheral Clock / (PSC+1)**

- **Timing Equation with Prescaler**
  - **t = N / (F * (PSC+1)**
  - **Where:**
    - **F** = Peripheral clock frequency
    - **PSC** = Prescaler value
    - **N** = Number of counts
    - **t** = Time delay
  - **Example**
    - **Peripheral clock = 72 MHz**
    - **Prescaler = 7199**
      - **Timer Clock=72,000,000 / 7200 = 10,000 Hz**
      - **Timer now increments 10,000 times per second**
      - **Each count = 0.1 ms**
- **Prescaler slows down the clock so the timer can measure longer time intervals accurately.**

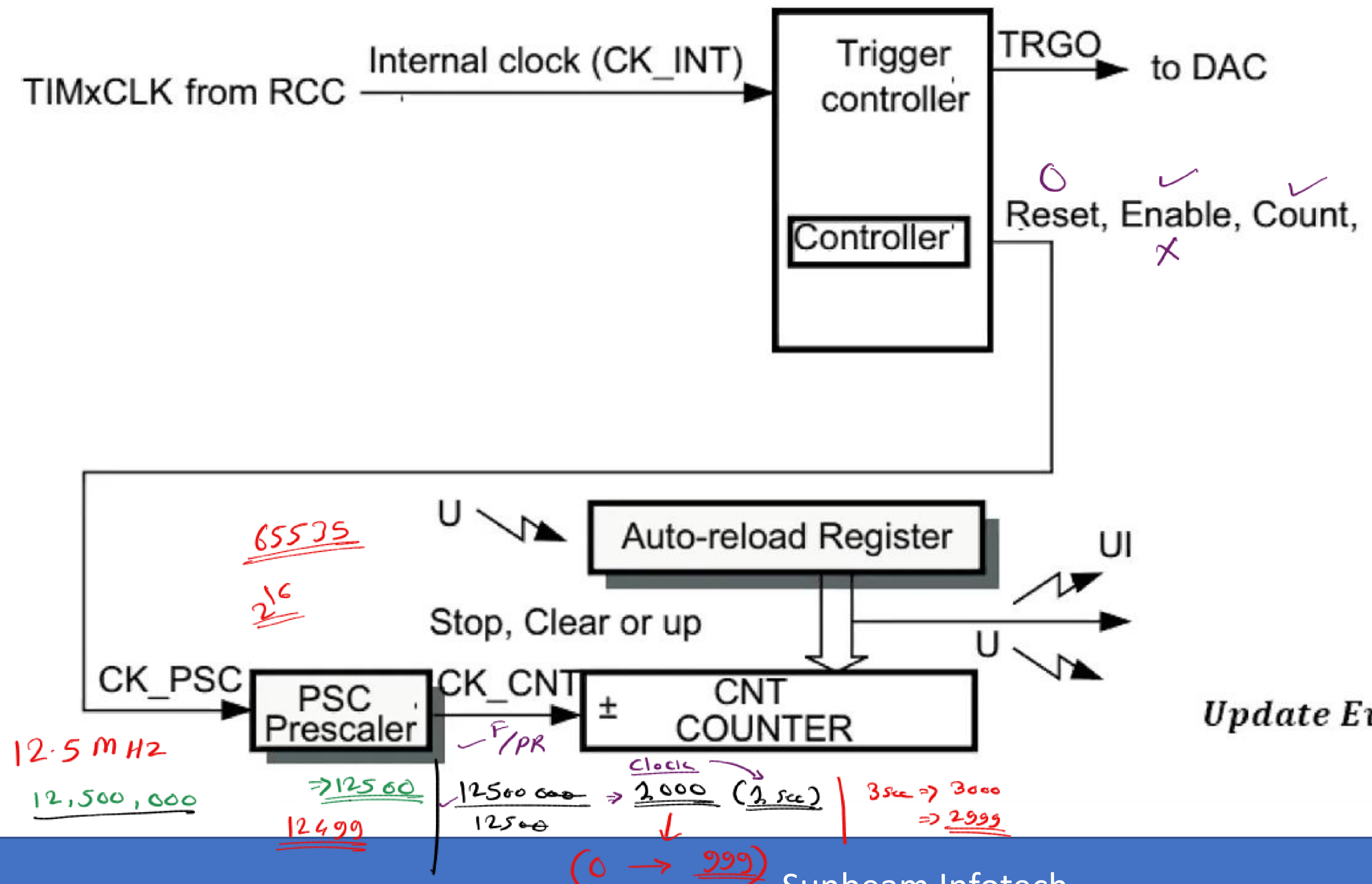
An Overview of STM32 Timers

- STM32 microcontrollers provide **different types of timers**, categorized based on their **features and complexity**.
- Each timer type is designed to solve **specific timing requirements** in embedded systems.
  - ❖ **Basic Timers (TIM6, TIM7)**
    - **Bit width:** 16-bit
    - Used only for **time-base generation**
    - Do **not** support input or output channels
    - Mainly used for:
      - Simple delays
      - Periodic interrupts
  - ❖ **General Purpose Timers (TIM2 – TIM5)**
    - **Bit width:** 16-bit or 32-bit
    - Support multiple timer functions, such as:
      - Time-base generation
      - Input Capture
      - Output Compare
      - PWM generation
    - Used in **most STM32 applications**

- ❖ **Advanced Control Timers (TIM1, TIM8)**
  - **Bit width:** 16-bit or 32-bit
  - Provide all features of General Purpose timers
  - Include additional features for control applications:
    - **Complementary PWM outputs**
      - Generate two opposite PWM signals, where one output is ON when the other is OFF. Used for controlling high-side and low-side switches in motor control.
    - **Dead-time insertion**
      - Adds a small delay between switching OFF one output and switching ON the other. Prevents short circuits and protects power devices.
    - **Break input for fault protection**
      - Automatically disables PWM outputs when a fault is detected. Protects the system from damage during abnormal conditions.

- **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.

Basic Timer (TIM6, TIM7)



$$H = \frac{F}{1000} * \frac{t}{PR}$$

$$H = \frac{12.5 * 1000 * 1000}{1000} * \frac{t}{12500}$$

$$H = 1 * t$$

100ms = 100 clock

1000ms = 10000 Clock (10 sec).

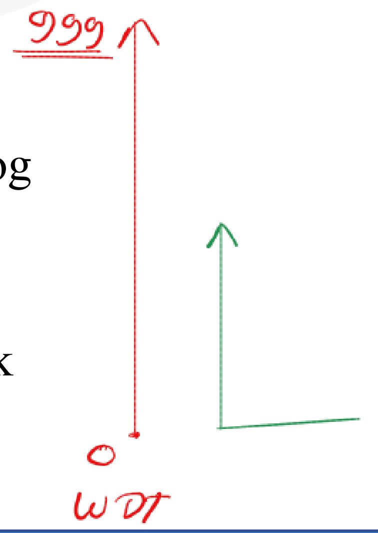
$$Update\ Event = \frac{TIMx\ CLK}{((PSC + 1)(ARR + 1))}$$

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.

Watchdog Timers (WDT)

- In any real-world product, software can and does fail.
- It can get stuck in an infinite loop, crash due to memory corruption, or be starved of CPU time by a buggy task.
- A Watchdog Timer (WDT) is an independent hardware timer designed to automatically recover the system from such software faults.
- **What is a Watchdog Timer?**
  - A Watchdog Timer is an **independent hardware timer**
  - It runs separately from the main application
  - Its job is to **monitor software health**
- **Basic Working Principle**
  - Watchdog starts a **countdown**
  - Software must periodically **refresh** (or “feed”) the watchdog
  - If the software:
    - Refreshes in time → system runs normally
    - Fails to refresh → watchdog assumes software is stuck
  - Watchdog then **resets the microcontroller**



- **Simple Analogy**
  - Think of the watchdog as a **heartbeat monitor**.
  - As long as the software sends a heartbeat, the system stays alive.
  - If the heartbeat stops, the system is restarted.

- **Purpose of Watchdog**
  - Automatically **recover from software faults**
  - Improve **system reliability**
  - Ensure system returns to a **known safe state**

- **Types of Watchdog Timers in STM32**
  - STM32 microcontrollers provide **two watchdog timers**:
    1. Independent Watchdog (IWDG)
    2. Window Watchdog (WWDG)

The SysTick Timer

- SysTick is a **24-bit down-counter** built directly into the **ARM Cortex-M core**
- It is **core-coupled**, not a peripheral timer
- Available on **all Cortex-M processors**
- **Why SysTick is Special**
    - **Standardized by ARM**
        - Same design and behavior on: STM32 , NXP Any Cortex-M MCU
        - This makes SysTick ideal for **portable software**
    - **Primary Purpose of SysTick**
        - To generate a **periodic interrupt**, called the **system tick.**
        - Acts as the **heartbeat of the system**
        - Used for:
            - Time tracking
            - Periodic tasks
            - Delay and timeout handling
            - RTOS scheduling (conceptual)

SysTick Programming (Using HAL)

- **SysTick in HAL**
  - In HAL-based projects, **SysTick is automatically configured**
  - Configuration happens when HAL_Init() is called
  - SysTick generates a **1 millisecond periodic interrupt**
  - This interrupt acts as the **global time base** for HAL
- **SysTick Configuration**
  - No manual configuration is required
  - HAL internally uses CMSIS to configure SysTick
    - HAL_Init();
- **SysTick Interrupt Handling**
  - SysTick interrupt occurs every **1 ms**
  - HAL provides the default handler:

```
void SysTick_Handler(void)
{
   HAL_IncTick();
}
```

  - HAL_IncTick() increments the **HAL system tick counter**

- **Using SysTick in Applications**
  - HAL provides APIs that rely on SysTick:
    - HAL_Delay(ms)
      - Blocking delay using SysTick
    - HAL_GetTick()
      - Returns elapsed time in milliseconds

```
uint32_t start = HAL_GetTick();
if (HAL_GetTick() - start >= 1000)
{
    // 1 second elapsed
}
```

- SysTick is a **core timer**, not a peripheral timer.
- It is used for **system timing**, not for application-specific delays.
- General-purpose timers are preferred for application timing.

Debug Watch Timer (DWT)

- **What is DWT?**
  - DWT (Debug Watch Timer) is a **core debug feature** available in **ARM Cortex-M processors**
  - It is part of the **ARM Core Debug unit**, not an STM32 peripheral
  - Provides a **cycle counter** that counts CPU clock cycles
- **Why DWT is Used**
  - General timers and SysTick work in **milliseconds**
  - Some peripherals and sensors require **microsecond-level accuracy**
  - DWT provides **cycle-accurate timing**
  - Ideal for **high-precision delays and profiling**
- **Key Feature Used in Projects**
  - **CYCCNT (Cycle Counter Register)**
    - Increments on every CPU clock cycle
    - Runs at **System Core Clock frequency**
    - Example:
      - CPU @ 168 MHz
      - 1 cycle = ~5.95 ns
      - Enables very accurate timing

- **Typical Applications of DWT**
  - Microsecond delays (e.g., DHT11 timing, sensor protocols)
  - Performance measurement
  - Code execution time profiling
  - Debugging and optimization

- **How DWT Works (Conceptual Flow)**
  - Enable DWT cycle counter
  - Reset cycle counter to zero
  - Wait until required number of cycles are counted
  - Continue execution

Thank you!

Kiran Jaybhave

email – kiran.jaybhave@sunbeaminfo.com