Mastering Time: A Deep Dive into STM32 Timers

STM32 Timer - Input Capture Mode

While Output Compare *generates* precise signals, **Input Capture** *measures* them. This mode is the timer's "stopwatch" function. It allows you to capture the exact value of the timer's counter at the moment an event occurs on an external input pin.

This capability is the foundation for a vast range of measurement tasks in embedded systems:

- **Frequency and Period Measurement:** By capturing the time between two consecutive rising edges of a signal.
- **Pulse Width Measurement:** By capturing the time on a rising edge and then on the corresponding falling edge.
- **Duty Cycle Measurement:** A combination of period and pulse width measurements.
- **Decoding Complex Signals:** Many sensors, like ultrasonic distance sensors or IR remote controls, encode information in the timing of their output pulses.

How Input Capture Works

The mechanism is an elegant reversal of Output Compare:

1. A timer is configured in "free-running" mode, with its counter (`CNT`) continuously incrementing at a known frequency.
2. A timer channel (e.g., CH1) is configured as an input, and its corresponding physical pin (e.g., PA0) is connected to the signal you want to measure.
3. You configure the channel to be sensitive to a specific signal edge (e.g., rising edge).
4. When that edge is detected on the pin, the hardware **instantly latches (copies) the current value of the `CNT` register into the `TIMx_CCR1` register**.
5. This capture event can also be configured to trigger an interrupt, allowing your software to read the captured timestamp from `CCR1` without needing to poll.

By capturing the timestamp of a start event and an end event, you can subtract the two values to get a precise count of the ticks that elapsed, which you can then convert into a time duration.

Configuration Steps for Input Capture

1. **Configure GPIO Pin:** Set the pin as an input, often a floating input or with a pull-resistor depending on the signal source.
2. **Enable Timer & GPIO Clocks:** Enable the peripheral clocks for both the timer and the GPIO port.

3. **Configure Time Base:** Set the timer's `PSC` to get a desired measurement resolution. Set the `ARR` to its maximum value (e.g., `0xFFFF` for a 16-bit timer) to let the counter run for as long as possible without overflowing.
4. **Configure the Channel for Input:**
    - In the `TIMx_CCMRx` register, set the `CCxS` bits to `01` to configure the channel as an input, mapping it to the corresponding pin.
    - In the `TIMx_CCER` register, set the `CCxE` bit to enable the capture unit and use the `CCxP`/`CCxNP` bits to select the polarity of the edge you want to capture (rising, falling, or both).
5. **Enable Interrupts (Recommended):** Set the `CCxIE` bit in the `TIMx_DIER` register and enable the timer's global interrupt in the NVIC.
6. **Enable Counter:** Set the `CEN` bit in `TIMx_CR1` to start the timer counting.

Your interrupt handler will then be triggered on each capture event, where you can read the timestamp from the `CCRx` register.

Application: HC SR-04 Interfacing

The HC-SR04 sensor measures distance based on the "time-of-flight" principle, where it measures the time it takes for a sound wave to travel to an object and return.

Using HC-SR04

1. Initiate measurement: Microcontroller sends a short, 10-microsecond high pulse to the sensor's TRIG pin to start a measurement.
2. Wait for echo: The sensor then sends out an 8-cycle ultrasonic burst and raises its ECHO pin high. When ECHO is raised, the ECHO pin will be pulled low by the sensor. Microcontroller should wait for it.
3. Capture pulse duration: The micro-controller captures and stores its counter value when the ECHO pin's signal goes from low to high (rising edge). It should also captures and stores the counter value again when the signal goes from high to low (falling edge).
4. Calculate distance: The difference between these two captured counter values gives you the exact time elapsed (the pulse duration), which you can then use to calculate the distance.

Programming Steps

1. Configure Timer Peripheral:
    - Select a timer (e.g., TIM2, TIM3, etc.).
    - Set the clock source for the timer (internal clock).
    - Choose one of its channels to operate in "Input Capture Direct Mode" and configure it to capture on both the rising and falling edges.
    - Enable the corresponding timer global interrupt in the NVIC settings.

2. Configure GPIO Pins:
    - Set the TRIG pin as a digital output pin.
    - Set the ECHO pin as a timer input capture pin, ensuring the correct channel is selected.
3. Write the Application Code:
    - In the main, send a 10us high pulse to the TRIG pin to start the measurement.
    - In the timer's interrupt service routine (ISR), handle the capture events.
    - On the rising edge, record the timer's counter value (e.g., TIMx->CCR1).
    - On the falling edge, record the new timer value and calculate the difference to find the pulse duration.
    - After calculating the distance, you can reset the timer for the next measurement.

Watchdog Timers

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.

The concept is beautifully simple: it's a **countdown timer** that is separate from your main application logic. Your main software loop has one critical job: it must periodically "pet" or "refresh" the watchdog to reset its countdown. If the software ever fails to do this within a specified time, the watchdog assumes the system has hung, and it triggers a **hardware reset** to bring the system back to a known, safe state.

STM32 microcontrollers offer two types of watchdogs.

1. Independent Watchdog (IWDG)

The IWDG is designed for maximum robustness. It is as independent as possible from the rest of the system.

- **Independent Clock:** It runs from its own dedicated, low-speed internal oscillator (`LSI`, typically ~32 kHz). This is a critical feature: it means the IWDG keeps running and can reset the MCU even if the main high-speed system clock fails.
- **Simple Timeout:** You configure a single timeout period. Your software must refresh the IWDG at any point *before* this period expires.
- **Forced Reset Only:** When it times out, its only possible action is to force a system-wide hardware reset.
- **Accuracy Trade-off:** Because the LSI clock is an RC oscillator, it is less precise than a crystal-based clock, so the timeout period is approximate.

The IWDG is a powerful tool for catching catastrophic system hangs. Its 12-bit counter and configurable prescaler give it a timeout range from about **125 microseconds to 32.8 seconds**.

Programming Sequence for IWDG:

1. **Enable LSI Clock:** The IWDG's clock source must be enabled and stable before configuration.
2. **Start IWDG:** Write the "start key" `0xCCCC` to the Key Register (`IWDG_KR`). Once started, it **cannot be stopped** except by a system reset.
3. **Enable Register Access:** Write the "unlock key" `0x5555` to `IWDG_KR`. This is a safety feature to prevent accidental changes to the configuration.
4. **Configure Timeout:** Set the prescaler (`IWDG_PR`) and reload (`IWDG_RLR`) registers to define the timeout period.
5. **Wait for Update:** Poll the Status Register (`IWDG_SR`) to ensure the new values have been loaded.
6. **Feed the Dog:** Periodically, your main application logic must write the "refresh key" `0xAAAA` to `IWDG_KR`. This reloads the down-counter with the value from `RLR` and starts the countdown again.
7. **Check Reset Reason (After Boot):** In your startup code, it's crucial to check the `IWDGRSTF` flag in the `RCC_CSR` register. If this flag is set, it means the current boot is the result of an IWDG timeout. You can then log this failure, blink an error LED, or enter a safe diagnostic mode.

Bare-Metal Code Example (IWDG):

```
// Initialize the Independent Watchdog with a given timeout in milliseconds.
void IWDG_Init(uint32_t ms) {
    // 1. Enable the Low-Speed Internal (LSI) oscillator and wait for it to be ready.
    RCC->CSR |= RCC_CSR_LSION;
    while(!(RCC->CSR & RCC_CSR_LSIRDY));

    // 2. Start the IWDG by writing the key value.
    IWDG->KR = 0xCCCC;

    // 3. Enable access to the IWDG configuration registers.
    IWDG->KR = 0x5555;

    // 4. Set the prescaler to its maximum value (256) for the longest possible timeout range.
    IWDG->PR = IWDG_PR_PR_0 | IWDG_PR_PR_1 | IWDG_PR_PR_2; // Divisor = 256

    // 5. Calculate the reload value based on the desired timeout.
    // LSI is ~32kHz. Timeout(s) = RLR * Prescaler / 32000
    // RLR = (Timeout_ms * 32) / Prescaler
    uint32_t reload_val = (ms * 32) / 256;
```

```
    IWDG->RLR = reload_val;

    // 6. Wait for the prescaler and reload registers to be updated.
    while(IWDG->SR);

    // 7. Perform the first refresh to start the countdown.
    IWDG_Refresh();
}

// "Pet the dog" to prevent a reset. This must be called periodically.
void IWDG_Refresh(void) {
    IWDG->KR = 0xAAAA;
}

// Check if the last reset was caused by the IWDG.
int Was_IWDG_Reset(void) {
    if (RCC->CSR & RCC_CSR_IWDGRSTF) {
        // Clear the reset flag so we don't detect it again on the next boot.
        RCC->CSR |= RCC_CSR_RMVF;
        return 1;
    }
    return 0;
}

// Example application demonstrating IWDG usage.
int main(void) {
    SystemInit();

    // Check if we are booting up because of a watchdog timeout.
    if (Was_IWDG_Reset()) {
        // If so, enter a "fault" state, e.g., blink a red LED rapidly.
        LedInit(LED_RED);
        while(1) {
            LedBlink(LED_RED, 100);
            DelayMs(100);
        }
```

```
    }

    // Normal boot sequence.
    SwitchInit(SWITCH_PIN);
    LedInit(LED_GREEN);

    // Initialize the IWDG with a 5-second timeout.
    IWDG_Init(5000);

    while(1) {
        // This loop simulates a task that must complete within the watchdog window.
        // It waits for a switch press.
        while(SwitchGetState(SWITCH_PIN) == 0) {
            // If the switch is never pressed, IWDG_Refresh() will not be called,
            // and the system will reset after 5 seconds. This simulates a hung task.
        }

        // The task "completed" successfully.
        LedBlink(LED_GREEN, 500);

        // We must refresh the watchdog to signal that the software is still alive.
        IWDG_Refresh();
    }
}
```

2. Window Watchdog (WWDG)

The WWDG is more sophisticated. It's designed to catch not only tasks that are running too slow (stuck), but also tasks that are running **too fast** (perhaps due to a timing bug or incorrect scheduler logic).

- **System Clock Dependent:** It is clocked from the main APB peripheral clock, making its timing very accurate.
- **Time "Window":** You configure a **maximum time** (the "window upper bound") and a **minimum time** (the "window lower bound"). You are only allowed to refresh the watchdog within this specific window.
    - Refreshing *after* the max time -> RESET.

    - Refreshing *before* the min time -> RESET.
- **Early Wakeup Interrupt (EWI):** The WWDG can be configured to trigger an interrupt just before the timeout window closes, giving the software a final chance to save critical state or perform a graceful shutdown.

The SysTick Timer

While the peripheral timers are powerful, the ARM Cortex-M core itself contains a simple, standardized 24-bit down-counter called the **SysTick Timer**.

- **Core-Coupled & Standardized:** Its design and register map are part of the ARM Cortex-M specification. This means it is identical on every Cortex-M MCU from every vendor. This standardization makes it the perfect tool for portable software, most notably **Real-Time Operating Systems (RTOS)**.
- **Primary Purpose:** To generate a periodic interrupt, known as the **system tick**. This tick is the fundamental heartbeat for an RTOS, which uses it to manage task scheduling (time-slicing), software timers, and other time-based kernel services.

SysTick Programming

The CMSIS (Cortex Microcontroller Software Interface Standard) provides a simple function, `SysTick_Config()`, that handles the setup with a single line of code.

1. **Configure and Start:** Call `SysTick_Config(reload_value)`. To get a 1ms tick, you would use: `SysTick_Config(SystemCoreClock / 1000);` This function calculates the reload value, sets the priority, enables the interrupt, and starts the timer.
2. **Implement the Handler:** You must provide the `SysTick_Handler()` function. This is a standard exception handler name defined in the startup file. Inside this function, you typically just increment a global tick counter variable.
3. **Use the Tick:** Your application can then use this global tick counter to implement non-blocking delays, check for timeouts, and manage other time-based logic.

Bare-Metal Code Example: SysTick for Delays

```
// This global variable acts as our system-wide tick counter.
// It MUST be declared as 'volatile' to prevent the compiler from optimizing away
// checks on its value, since it is modified by an interrupt.
volatile uint32_t g_jiffies = 0;

// The SysTick Interrupt Service Routine. This function is called by the hardware
// automatically every time the SysTick counter reaches zero.
```

```c
void SysTick_Handler(void) {
    g_jiffies++; // Increment our system tick counter.
}

// A simple blocking delay function that uses the SysTick counter.
void SysTick_DelayMs(uint32_t delay_ms) {
    // Calculate the tick value when the delay should end.
    uint32_t end_time = g_jiffies + delay_ms;

    // Wait in a loop until the global tick counter reaches the end time.
    while (g_jiffies < end_time);
}

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

    // Configure and start the SysTick timer to generate an interrupt every 1ms.
    // 'SystemCoreClock' is a global variable (updated by SystemInit()) that holds
    // the CPU's core clock frequency in Hz.
    SysTick_Config(SystemCoreClock / 1000);

    while(1) {
        // Now we can use our tick-based delay.
        LedToggle(LED_GREEN);
        SysTick_DelayMs(500); // Wait for 500ms
    }
}
```

Real-Time Clock (RTC)

While SysTick and peripheral timers are excellent for measuring relative time intervals, they lose their state when the MCU is reset or powered off. To keep track of **absolute calendar time**, we need the **Real-Time Clock (RTC)**.

The RTC is a special, ultra-low-power timer peripheral designed to run continuously, even when the rest of the MCU is in its deepest sleep mode or completely powered off, as long as it has a backup power source (like a small coin cell battery or supercapacitor).

Key Features of the STM32 RTC:

Feature	Description
Function	Provides absolute calendar (Year, Month, Day) and clock (Hour, Minute, Second) timekeeping.
Power Source	Designed to run from a separate backup power domain (`VBAT` pin), keeping time when main power is off.
Data Format	Internally uses **Binary Coded Decimal (BCD)** format for all time and date registers.
Core Registers	`RTC_TR` (Time Register) and `RTC_DR` (Date Register) hold the current time and date.
Alarms	Two independent alarms (A and B) that can be configured to trigger an interrupt at a specific time and date.
Periodic Wakeup	Can be configured to generate a periodic wakeup interrupt to wake the MCU from low-power modes.
Auto Management	Automatically handles leap years. Some variants also support daylight saving time adjustments.
Date Range	Typically supports years from 2000 to 2099.

The Importance of BCD (Binary Coded Decimal)

The RTC stores numbers in BCD format. In BCD, each decimal digit is encoded into a separate 4-bit nibble.

- **Example: 45 seconds**
  - **Binary:** `00101101` (2Dh)
  - **BCD:** `0100 0101` (45h) - The '4' and '5' are stored as separate nibbles.

This means your software must convert between normal binary numbers and BCD when setting or reading the time.

```
// Converts a standard 8-bit binary number to its 8-bit BCD representation.
uint8_t BinToBCD(uint8_t bin) {
    uint8_t tens = bin / 10;
    uint8_t units = bin % 10;
    return (tens << 4) | units;
}
```

```c
// Converts an 8-bit BCD number back to its binary representation.
uint8_t BCDToBin(uint8_t bcd) {
    uint8_t tens = (bcd >> 4);
    uint8_t units = bcd & 0x0F;
    return (tens * 10) + units;
}
```

RTC Clock Configuration

The accuracy of the RTC depends entirely on the accuracy of its clock source. The goal is to feed it a precise **1 Hz** clock. This is achieved using a cascade of prescalers.

- **Clock Source Options:**
  - **LSE (Low-Speed External):** A 32.768 kHz external crystal. This is the **most accurate and preferred** option for stable timekeeping.
  - **LSI (Low-Speed Internal):** The same ~32 kHz internal RC oscillator used by the IWDG. It's less accurate and will drift over time but requires no external components.
  - **HSE (High-Speed External):** The main system crystal (e.g., 8 MHz), divided down by a large prescaler. This is accurate but consumes more power and won't run when the main MCU is off.
- **Prescaler Chain:** The chosen RTC clock is fed through two dividers to get to 1 Hz. `1 Hz = RTCCLK / (Async_Prescaler + 1) * (Sync_Prescaler + 1)` _ **Example with LSE (32.768 kHz):** `1 Hz = 32768 / (127 + 1) * (255 + 1) = 32768 / (128 * 256)` _ **Example with LSI (~32 kHz):** `1 Hz = 32000 / (124 + 1) * (255 + 1) = 32000 / (125 * 256)`

Programming Sequence for RTC

The RTC lives in a special, write-protected "backup domain." Initializing it requires a specific, careful sequence.

1. **Enable Clocks:** Enable the clock for the **Power Interface (PWR)** and the RTC itself.
2. **Disable Write Protection:** Set the `DBP` (Disable Backup Domain Protection) bit in the `PWR_CR` register to allow access to RTC registers.
3. **Configure RTC Clock Source:** Select LSE, LSI, or HSE in the `RCC_BDCR` register and enable the RTC (`RTCEN`).
4. **Unlock RTC Registers:** The RTC registers themselves are write-protected. You must write a specific key sequence (`0xCA`, then `0x53`) to the `RTC_WPR` register to unlock them.
5. **Enter Initialization Mode:** Set the `INIT` bit in `RTC_ISR`. This freezes the calendar counters and allows you to safely update the configuration. Wait for the `INITF` flag to confirm the RTC is ready.

6. **Set Prescalers:** Configure the `RTC_PRER` register with the async and sync prescaler values to achieve a 1 Hz clock.
7. **Set Initial Time and Date:** Write the initial BCD-formatted values to the `RTC_TR` and `RTC_DR` registers.
8. **Exit Initialization Mode:** Clear the `INIT` bit in `RTC_ISR`. The calendar will now start running from the values you set.
9. **Re-enable Write Protection:** Clear the `DBP` bit in `PWR_CR` to lock the backup domain and prevent accidental changes.

Bare-Metal Code Example (RTC Initialization)

```
void RTC_Init(RTC_DateTypeDef* dt, RTC_TimeTypeDef* tm) {
    // 1. Enable PWR interface clock and allow access to backup domain
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    PWR->CR |= PWR_CR_DBP;

    // 2. Enable LSI clock and wait for it to be ready
    RCC->CSR |= RCC_CSR_LSION;
    while(!(RCC->CSR & RCC_CSR_LSIRDY));

    // 3. Select LSI as RTC clock source and enable the RTC
    RCC->BDCR |= RCC_BDCR_RTCEN | RCC_BDCR_RTCSEL_1;

    // 4. Unlock the RTC registers
    RTC->WPR = 0xCA;
    RTC->WPR = 0x53;

    // 5. Enter Initialization Mode
    RTC->ISR |= RTC_ISR_INIT;
    while(!(RTC->ISR & RTC_ISR_INITF));

    // 6. Configure prescalers for a 1 Hz clock from the ~32kHz LSI
    // 1Hz = 32000 / (124+1) * (255+1)
    RTC->PRER = ((125 - 1) << RTC_PRER_PREDIV_A_Pos) | ((256 - 1) << RTC_PRER_PREDIV_S_Pos);

    // 7. Set the initial date and time
    RTC_SetDate(dt);
    RTC_SetTime(tm);
```

```
    // Optional: Bypass the shadow registers for immediate reading
    RTC->CR |= RTC_CR_BYPSHAD;

    // 8. Exit Initialization Mode - the clock starts running now!
    RTC->ISR &= ~RTC_ISR_INIT;

    // 9. Re-enable write protection for the backup domain
    PWR->CR &= ~PWR_CR_DBP;
}

// Function to read the current time from RTC registers
void RTC_GetTime(RTC_TimeTypeDef* tm) {
    uint32_t time_reg = RTC->TR; // Read the 32-bit time register

    // Extract each BCD component, mask it, shift it, and convert to binary
    tm->Hours   = BCDToBin((time_reg & (RTC_TR_HT_Msk | RTC_TR_HU_Msk)) >> RTC_TR_HU_Pos);
    tm->Minutes = BCDToBin((time_reg & (RTC_TR_MNT_Msk | RTC_TR_MNU_Msk)) >> RTC_TR_MNU_Pos);
    tm->Seconds = BCDToBin((time_reg & (RTC_TR_ST_Msk | RTC_TR_SU_Msk)) >> RTC_TR_SU_Pos);
}

// Function to read the current date from RTC registers (similar logic)
void RTC_GetDate(RTC_DateTypeDef* dt) {
    uint32_t date_reg = RTC->DR;

    dt->Year    = BCDToBin((date_reg & (RTC_DR_YT_Msk | RTC_DR_YU_Msk)) >> RTC_DR_YU_Pos);
    dt->Month   = BCDToBin((date_reg & (RTC_DR_MT_Msk | RTC_DR_MU_Msk)) >> RTC_DR_MU_Pos);
    dt->Date    = BCDToBin((date_reg & (RTC_DR_DT_Msk | RTC_DR_DU_Msk)) >> RTC_DR_DU_Pos);
    dt->WeekDay = (date_reg & RTC_DR_WDU_Msk) >> RTC_DR_WDU_Pos; // Weekday is already binary
}
```

Assignments

1. Implement SysTick code in modular fashion i.e. systick.h and systick.c.
2. Glow LEDs in clockwise direction (similar to output compare mode) using SysTick_Delay.
3. Set the RTC to today's date and time. Configure Timer 9 for Periodic interrupt - 1 second. On each interrupt, read RTC and display on LCD.