

Mikroprocesory

6. Čítače / časovače a měření času

Stanislav Vitek

Katedra radioelektroniky

České vysoké učení technické v Praze

V předchozích přednáškách jsme viděli

- jak vypadá obecná struktura mikrokontroléru, periferie jsou připojeny ke sběrnícím
- práce s obecným vstupně výstupním obvodem
- princip a obsluha přerušení
- práce s registry
- spuštění a běh programu

Obsah přednášky

1. Zdroje času v mikroprocesorovém systému
2. Čítače / časovače
 - Komparační systém
 - PWM jako speciální případ komparace
 - Záchytný systém
3. SW metody měření času
4. Watchdog časovače
5. Speciální časovací obvody

Motivace a význam měření času

Čas jako klíčová veličina v řídicích systémech: určování polohy, synchronizace, řízení motorů, měření signálů.

Příklady:

- PWM → řízení výkonu motoru / LED jasu
- Input capture → měření otáček, doby letu signálu (ToF)
- Output compare → generování přesných impulzů (komunikace, synchronizace)
- Real-time → časové základny, watchdogy, plánování úloh v RTOS

Zároveň: téměř veškeré digitální systémy, které používáme, jsou synchronní, a zdroj hodinového signálu je nutný pro řízení všech dílčích částí systému.

1. Zdroje času v mikroprocesorovém systému

Interní a externí oscilátory

- HSI (High Speed Internal) – integrovaný RC, levný, nepřesný ($\pm 1\text{--}2\%$)
- HSE (High Speed External) – krystal, vyšší přesnost (typ. $\pm 30\text{ ppm}$)
- LSI / LSE – nízkofrekvenční pro RTC, watchdog

Frekvenční násobiče a děliče

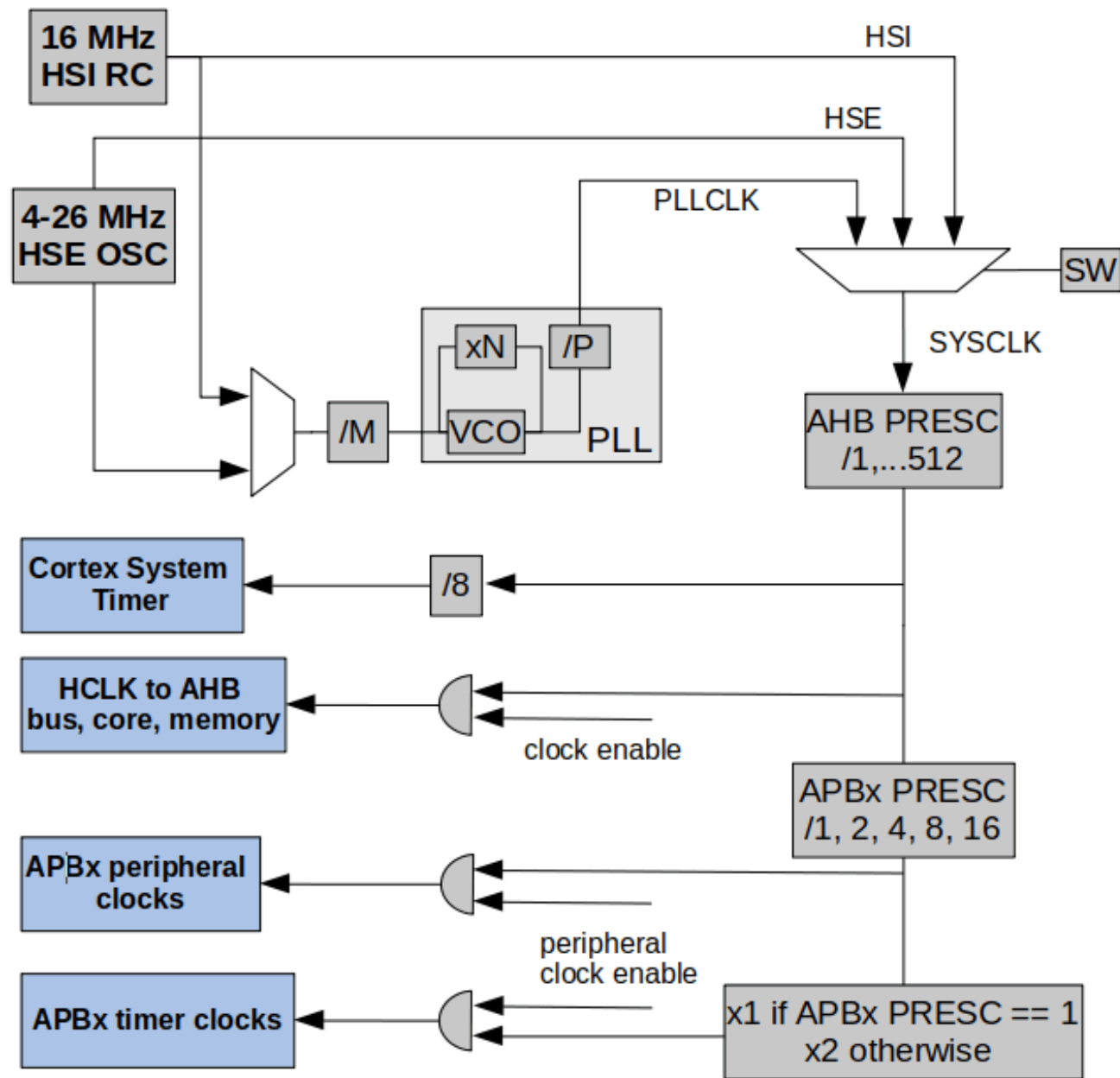
- PLL – zvyšuje frekvenci CPU i časovačů
- Přepínatelné prescalery pro AHB, APB1/2
- Stabilita, jitter, teplotní závislost

Pro metrologické aplikace → nutnost kalibrace, případně externí časová reference

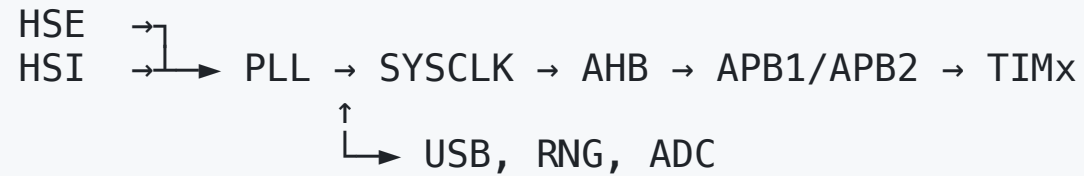
Zdroje systémových hodin

Každý mikrokontrolér STM32 má několik možných zdrojů hodin:

Zdroj	Frekvence	Přesnost	Typické využití
HSI – High Speed Internal	8–16 MHz	±1–2 %	Výchozí po resetu, bez krystalu
HSE – High Speed External	4–26 MHz	±30 ppm	Přesné aplikace, USB, měření času
LSI – Low Speed Internal	~32 kHz	±5 %	Watchdog, backup RTC
LSE – Low Speed External	32.768 kHz	±20 ppm	RTC, nízkopříkonové časování
PLL – Phase-Locked Loop	násobí frekvenci	—	Vytváří systémový takt (SYSCLK)



Typická hierarchie hodinového systému



- **SYSCLK** — hlavní hodinový signál pro CPU (Cortex-M4)
- **AHB** — hlavní sběrnice (paměť, **DMA**, **GPIO**)
- **APB1**, **APB2** — periferní sběrnice (**TIMy**, **UART**, **ADC**, **SPI** atd.)
- **TIMxCLK** — odvozený signál pro časovače (z **APB**)

Rodina	STM32 F0	STM32 L0	STM32 F3	STM32 F4	STM32 F7
SYSCLK	48 MHz	32 MHz	72 MHz	168 MHz	216 MHz

PLL – Phase Locked Loop

PLL násobí vstupní frekvenci (např. z HSE) na vysokou hodnotu pro CPU i periferie.

Základní rovnice:

$$f_{VCO} = \frac{f_{IN}}{PLL_M} \times PLL_N \quad f_{SYSCLK} = \frac{f_{VCO}}{PLL_P} \quad f_{USB} = \frac{f_{VCO}}{PLL_Q}$$

Příklad konfigurace (RCC_PLLCFGR , HSE = 8 MHz , SYSCLK = 168 MHz)

Parametr	Hodnota	Výpočet
PLL_M	8	8 MHz / 8 = 1 MHz
PLL_N	336	1 MHz × 336 = 336 MHz
PLL_P	2	336 / 2 = 168 MHz (SYSCLK)
PLL_Q	7	336 / 7 = 48 MHz (USB)

Příklad inicializace hodinového systému

```
// Zapnutí HSE a PLL, přepnutí na SYSCLK = 168 MHz
RCC->CR |= RCC_CR_HSEON;
while (!(RCC->CR & RCC_CR_HSERDY));

RCC->PLLCFGR = (8 << RCC_PLLCFGR_PLLM_Pos) |
               (336 << RCC_PLLCFGR_PLLN_Pos) |
               (2 << RCC_PLLCFGR_PLLP_Pos) |
               RCC_PLLCFGR_PLLSRC_HSE;

RCC->CR |= RCC_CR_PLLON;
while (!(RCC->CR & RCC_CR_PLLRDY));

FLASH->ACR = FLASH_ACR_ICEN | FLASH_ACR_DCEN |
             FLASH_ACR_LATENCY_5WS;

RCC->CFGR |= RCC_CFGR_SW_PLL;
```

Děličky a větve pro periférie

Každá větev má vlastní děličku:

AHB Prescaler: CPU, paměť, DMA

- `RCC_CFGR.HPRE` , dělička: 1–512

APB1 Prescaler: `TIM2–7` , `USART2–5` , `I2C`

- `RCC_CFGR.PPRE1` , dělička: 1, 2, 4, 8, 16

APB2 Prescaler: `TIM1` , `TIM8` , `ADC` , `USART1`

- `RCC_CFGR.PPRE2` , dělička: 1, 2, 4, 8, 16

Speciální případ — frekvence časovačů (TIMxCLK)

Časovače mají speciální násobič, pokud běží na sběrnici APB, která je dělená.

Pokud je dělička $APB \neq 1$, pak je frekvence časovače dvojnásobkem frekvence sběrnice.

Příklad:

1. APB1 - dělička 4

- $f_{APB1} = 42 \text{ MHz}$, $f_{TIMxCLK} = 84 \text{ MHz}$

2. APB2 - dělička 2

- $f_{APB2} = 84 \text{ MHz}$, $f_{TIMxCLK} = 168 \text{ MHz}$

To je důvod, proč i při dělení sběrnice časovače běží „rychleji“

- potřebují vysoké rozlišení pro PWM a měření.

HSI kalibrace

Po resetu je `HSI` vždy výchozí zdroj systémových hodin.

Tovární kalibrace

`HSI` má v registru `RCC->ICSCR` (Internal Clock Source Calibration Register)

- pole `HSICAL[7:0]`, které obsahuje tovární kalibrační hodnotu (typicky $\pm 1\%$).
- pole `HSITRIM[4:0]` umožňuje uživatelskou jemnou kalibraci ($\pm 1\%$ rozsah).

```
RCC->ICSCR = (RCC->ICSCR & ~RCC_ICSCR_HSITRIM_Msk) |  
             (16 << RCC_ICSCR_HSITRIM_Pos);
```

HSI stabilita

Parametr	Typická hodnota	Komentář
Přesnost po kalibraci	$\pm 1 \%$	továrně měřená při 25 °C, 3.3 V
Teplotní drift	$\pm 0.1 \%$ / 10 °C	při výrazném zahřátí až $\pm 2 \%$
Napěťová závislost	$\pm 0.5 \%$	mezi 1.8–3.6 V
Start-up time	$\sim 100 \mu\text{s}$	velmi rychlý náběh oproti krystalu

Kalibrace pomocí externího signálu:

S externím přesným zdrojem (např. z GPS nebo LSE) lze HSI dynamicky přeladit podle měření přes Timer input capture → tzv. software trimming.

Interní (HSI / LSI)

- Aktivují se nastavením `RCC->CR_HSION` nebo `RCC->CSR_LSION` .
- Startují velmi rychle (desítky μs), MCU může běžet téměř okamžitě po resetu.

Externí (HSE / LSE)

- Aktivují se bitovým polem `RCC->CR_HSEON` / `RCC->BDCR_LSEON` .
- Krystal potřebuje dobu náběhu typicky:
 - HSE: 0.5–5 ms (záleží na kapacitách, krystalu, teplotě)
 - LSE: až 500 ms (32 kHz krystal je pomalý)
- Po zapnutí je nutné počkat na příznak `RDY` (`HSERDY` , `LSERDY`).

Clock Security System (F4/F7) - pokud `HSE` ztratí synchronizaci, přepne na HSI (+ vyvolá NMI), aktivace:

`RCC_CR_CSSON = 1` .

Kdy se jádro restartuje (Reset conditions)

1. **Power-on reset (POR)** - napájení se zvedne nad prah, `RCC_CSR_PORRSTF`
2. **Pin reset (NRST)** - reset tlačítko / pin, `RCC_CSR_PINRSTF`
3. **Software reset** - např. `NVIC_SystemReset()`, `RCC_CSR_SFTRSTF`
4. **Independent watchdog reset** - timeout IWDG, `RCC_CSR_IWDGRSTF`
5. **Window watchdog reset** - timeout WWDG, `RCC_CSR_WWDGRSTF`
6. **Brown-out reset** - pokles napětí pod mez `RCC_CSR_BORRSTF`
7. **Clock failure reset** - výpadek HSE oscilátoru, `RCC_CSR_CSSRSTF`

Po resetu se `HSI` znovu zapne a stane se `SYSCLK`.

`PLL`, `HSE`, `LSE` se vypínají → je třeba je reinicializovat.

Hodnoty registrů (kromě zálohové oblasti `RTC`) se vrací do výchozího stavu.

2. Čítače / časovače

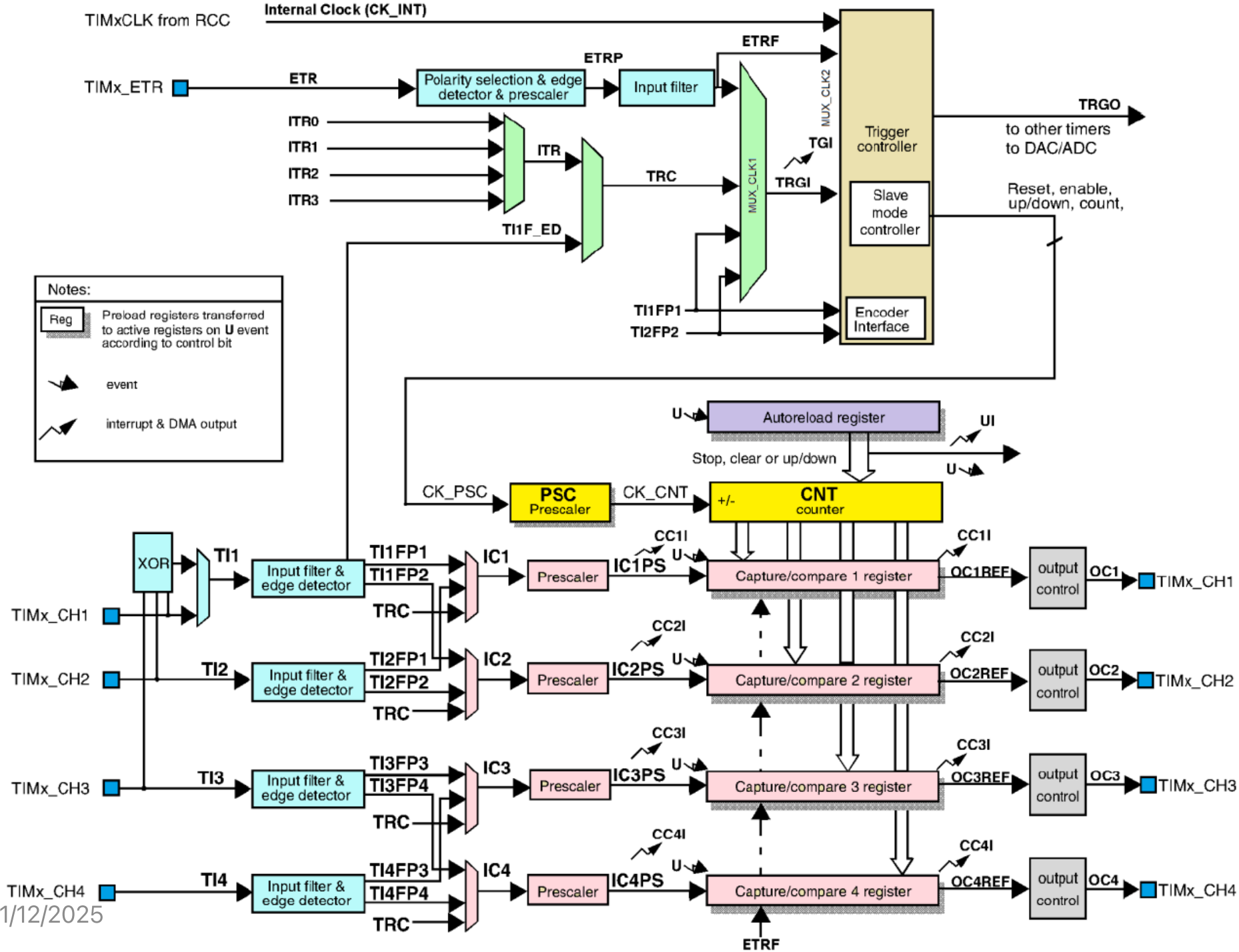
Funkce a režimy čítače

Nejjednodušší režim čítače je prosté čítání nahoru, dolů nebo obousměrné interního nebo externího signálu s pevnou nebo proměnnou horní mezí.

Umožňuje čítat události, generovat časové intervaly (řízení, časová základna, RTOS).

Změna intervalu:

- Programovým přednastavením – problémy (viz dále)
- S obvodovým přednastavením
 - Po přetečení čítače
 - Po externí události.



Základní nastavení čítače - prescaler a interval

Aktivace časové základny `TIM2` připojeného k `APB1` proběhne zápisem do `RCC`

```
RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
```

Pro frekvenci f_{CNT} , s jako čítač běží, platí $f_{CNT} = \frac{f_{TIMxCLK}}{PSC+1}$

Za předpokladu, že frekvence hodin připojených k čítači je 84 MHz, následujícím nastavením prescaleru `PSC` získáme rozlišení čítače 1 μ s

```
TIM2->PSC = 84 - 1;
```

Nastavení intervalu čítače je pak provedeno registrem `ARR`, zde na 1 ms

```
TIM2->ARR = 1000;
```

Dosažení intervalu čítače

Defaultní hodnota `ARR` je `0xFFFF` pro 16b a `0xFFFF FFFF` pro 32b časovače.

- při vzestupném čítání dosáhne `CNT==ARR` → přetečení → `CNT=0`
- při sestupném čítání dosáhne `CNT==0` → podtečení → `CNT=ARR`

Při `CNT==ARR` dojde k signalizaci události **update event**

- v příslušném `SR` se nastaví bit `UIF` (Update Interrupt Flag)
- pokud je nastaven bit `UIE` v `TIMx_DIER`, vygeneruje se přerušení `TIMx_IRQn`

Většina časovačů má tzv. shadow registry:

- Preload registry pro `ARR`, `CCR`, někdy i `PSC`.
- Tyto hodnoty se nepřeklopí okamžitě, ale až při update eventu.

CR1 – Control Register 1

Uveden pouze výčet atributů základní funkcionality, ovládání pokročilých funkcí bude zmíněno u popisu těchto funkcí.

CEN (bit 0) – Start/stop čítače

```
TIM2->CR1 |= TIM_CR1_CEN; // start  
TIM2->CR1 &= ~TIM_CR1_CEN; // stop
```

UDIS (bit 1) – Zakázání update

Pokud je nastavený `UDIS = 1`, pak se:

- negenerují update eventy (`UEV`) a nenastavuje `UIF` (Update Interrupt Event) v `SR`
- neaktualizují prelobované registry (`ARR`, `PSC`)

Typické použití

ARM časovače mají vnitřní pipeline, která zapisuje nové hodnoty pro `ARR`, `PSC` nebo `CCR` registry do jejich preload registrů, a to skutečných registrů se hodnota zkopíruje při update události. (Takže ke změně parametrů může dojít jindy, než je očekáváno.)

Bezpečná změna parametrů časovače:

- `UDIS = 1` → přepis registrů → `UDIS = 0` → update přes `UG`.

```
TIM2->CR1 |= TIM_CR1_UDIS;    // Disable update events
TIM2->PSC = 83;                // 84 - 1
TIM2->ARR = 999;
TIM2->CR1 &= ~TIM_CR1_UDIS;    // Enable updates
TIM2->EGR |= TIM_EGR_UG;      // Force update (manual reload)
```

Např. při řízení více motorů PWM je třeba, aby všechny kanály aktualizovaly hodnoty `CCR` v přesně daném okamžiku.

URS (bit 2) – Zdroj update

Říká, kdy se generuje update event:

0 → z přetečení nebo SW zápisu do EGR (viz předchozí příklad)

1 → pouze z přetečení čítače

OPM (bit 3) – One Pulse Mode

Po jednom cyklu (dosažení ARR nebo compare match) se čítač automaticky vypne.

- typické pro generování jednoho pulzu, např. v kombinaci s PWM.

DIR (bit 4) – Směr čítání

Pouze u „up/down“ timerů (TIM2–5, TIM1–8). Základní timery (TIM6/7) jsou jen count-up.

0 → směrem nahoru (CNT++)

1 → směrem dolů (CNT--)

ARPE (bit 7) – Auto-reload preload

Když je zapnutý, změna registru **ARR** se neaplikuje ihned po zápisu, ale až po dalším update eventu **UEV** .

Kombinace s bitem **UDIS**

ARPE	UDIS	Výsledek
1	0	ARR se aktualizuje při každém UEV (běžné použití)
1	1	ARR se <i>neaktualizuje vůbec</i> , dokud není znovu nepovolen update nebo není generován UG ručně
0	x	ARR se vždy přenesse ihned, bez ohledu na UDIS

SR – Status Register

Indikuje, co se stalo – přerušení, zachycení, přetečení atd.

UIF - Update interrupt flag, přetečení (`CNT == ARR`)

CC1IF..CC4IF - Capture/Compare flags, zachyceno / dosaženo

CC10F..CC40F - Overcapture flags, nové zachycení dřív, než jsi přečetl to staré

TIF - Trigger flag, u synchronizovaných timerů

Zatímco **CCxIF** se smažou přečtením, ostatní je nutné smazat zápisem `0` :

```
if (TIM2->SR & TIM_SR_UIF) { // test flagu
    TIM2->SR &= ~TIM_SR_UIF; // clear flag
    // obsluha přerušení
}
```

CCMRx – Capture/Compare Mode Register

Každý registr CCMR obsluhuje 2 kanály: CCMR1 → CH1 a CH2, CCMR2 → CH3 a CH4

Struktura (pro input capture mód)

CC1S – výběr vstupu pro kanál 1

- 00 – output compare (PWM) 01 – vstup na pinu CH1
- 10 – vstup na pinu CH2 (cross-mapping) 11 – vstup pro synchronizaci

IC1PSC – prescaler vstupního signálu

IC1F – filtr vstupu (debounce)

CC2S – výběr vstupu pro kanál 2

IC2PSC – prescaler, zachytne 1., 2., 4. nebo 8. událost

IC2F – vstupní filtr

CCER – Capture/Compare Enable Register

Určuje, které kanály jsou aktivní a na jakou hranu reagují.

CCxE Capture enable, **1** = kanál povolen

CCxP Polarity, **0** = vzestupná, **1** = sestupná hrana

CCxNE (jen advanced) – negovaný výstup

CCxNP druhá polarita pro komplementární výstup (jen advanced)

```
TIM2->CCER |= TIM_CCER_CC1E;      // povolit capture
TIM2->CCER &= ~TIM_CCER_CC1P;     // vzestupná hrana
TIM2->CCER |= TIM_CCER_CC2E;
TIM2->CCER |= TIM_CCER_CC2P;      // sestupná hrana
```

CCRx – Capture/Compare Register

Aktuální naměřený čas (v Input Capture módu) nebo cílový čas (v Output Compare módu).

Každý kanál má svůj vlastní: CCR1 , CCR2 , CCR3 , CCR4

Když při Input Capture timer detekuje hranu:

- obsah CNT (aktuální čítač) se zkopíruje do CCRx
- a nastaví se flag CCxIF v SR

Poté už si software může zaregistrovaný čas přečíst:

```
if (TIM2->SR & TIM_SR_CC1IF) {  
    uint32_t t1 = TIM2->CCR1;  
}
```

Komparační systém

Komparační systém se skládá z jednoho nebo několika komparačních registrů, časovače a komparátorů, které indikují shodu obsahu registrů s časovačem. Systém může pracovat v konfiguraci PWM nebo v konfiguraci generování řídicích signálů.

Základní idea:

- Čítač `CNT` běží podle hodin f_{TIM} a porovnává se s hodnotou v registru `CCRn`.
- Když `CNT == CCRn`, generuje se Compare Match Event → může:
 - změnit výstupní pin (toggle, set, reset), podle režimu komparace
 - vyvolat přerušení,
 - spustit DMA přenos,
 - vytvořit interní synchronizační událost (např `TRG0` pro `ADC`).

Režimy komparace

Režim	OCxM kód	Chování při shodě $CNT = CCRx$
Frozen	000	Žádná akce
Active on match	001	Nastaví pin do 1
Inactive on match	010	Nastaví pin do 0
Toggle on match	011	Invertuje stav pinu
Force active/inactive	100/101	Trvale 1/0
PWM mode 1	110	Pin aktivní, dokud $CNT < CCRx$
PWM mode 2	111	Pin aktivní, dokud $CNT > CCRx$

Příklad - Generování pulzu pomocí One Pulse Mode (OPM)

One-Pulse Mode (OPM) generuje **jednorázový** puls definované délky. Po skončení se časovač automaticky zastaví.

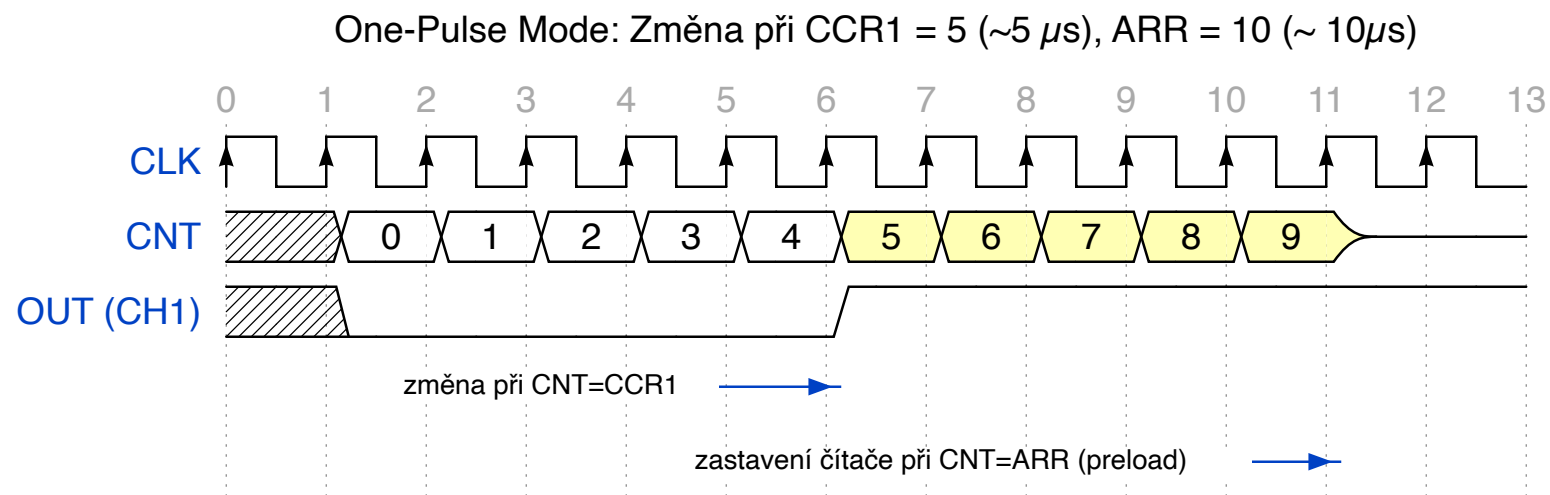
Základní koncept:

- Časovač se spustí, běží do přetečení (`CNT` dosáhne `ARR`)
- Během běhu řídí výstup podle zvoleného compare režimu
- Po přetečení se **automaticky zastaví** (`CEN = 0`)

Varianta A - režim toggle on match

```
TIM2->PSC = 83;           // 1 MHz (1 µs tick)
TIM2->ARR = 10;           // délka periody
TIM2->CCR1 = 5;           // match po 5 hodinových cyklech

TIM2->CCMR1 = (3 << TIM_CCMR1_OC1M_Pos); // toggle on match
TIM2->CCER |= TIM_CCER_CC1E; // Povolení výstupu
TIM2->CR1 |= TIM_CR1_OPM;    // One-Pulse Mode
TIM2->CR1 |= TIM_CR1_CEN;    // Start
```



Po dosažení **ARR** v One Pulse Mode se čítač zastaví, výstup zůstane v posledním stavu (zde HIGH) a po restartu se invertuje.

Varianta B - PWM mode 1

```
// Inicializace (jen jednou)
TIM2->PSC = 83; // 1 MHz
TIM2->ARR = 10; // délka periody (11 taktů: 0-10)
TIM2->CCR1 = 5; // 50% duty cycle (HIGH po dobu 5 taktů)
```

```

TIM2->CCMR1 = (0b110 << TIM_CCMR1_OC1M_Pos); // PWM mode 1
TIM2->CCMR1 |= TIM_CCMR1_OC1PE;           // Preload enable
TIM2->CCER |= TIM_CCER_CC1E;             // Output enable
TIM2->CR1 |= TIM_CR1_OPM;                // One-Pulse Mode

// Pro každý nový puls:
TIM2->EGR |= TIM_EGR_UG;                 // Update event
TIM2->CR1 |= TIM_CR1_CEN;                // Start
// Časovač automaticky zastaví po ARR, výstup je HIGH po dobu CCR1

```

Jak to funguje:

- PWM mode 1: Výstup je HIGH když $CNT < CCR1$
- Update event nastaví $CNT = 0 \rightarrow$ výstup skočí na HIGH
- OPM zastaví čítač po přetečení \rightarrow výstup skočí na LOW

Příklad - Generování pulzu pomocí dvou kanálů

Alternativní přístup:

- Použití dvou compare kanálů s rozdílnými časy pro vytvoření pulsu.
- Tato metoda generuje **dva samostatné výstupy** (PA0=CH1, PA1=CH2), které lze kombinovat externí logikou.

Princip

- **CH1 (PA0):** "Active on match" → nastaví výstup na HIGH při `CNT == CCR1`
- **CH2 (PA1):** "Inactive on match" → nastaví výstup na LOW při `CNT == CCR2`
- **Výsledek:** Puls o délce `CCR2 - CCR1`

Použití

- Řízení H-můstku, diferenciální signály, synchronizace s externí logikou.

Konfigurace

```
// GPIO konfigurace pro dva výstupy
GPIOA->MODER |= GPIO_MODER_MODE0_1 | GPIO_MODER_MODE1_1; // AF mode
GPIOA->AFR[0] |= (1 << 0) | (1 << 4); // AF1: TIM2_CH1, TIM2_CH2

// Timer konfigurace
TIM2->PSC = 83; // 1 MHz
TIM2->ARR = 100; // 100 µs perioda
TIM2->CCR1 = 10; // SET po 10 µs
TIM2->CCR2 = 30; // RESET po 30 µs → puls 20 µs

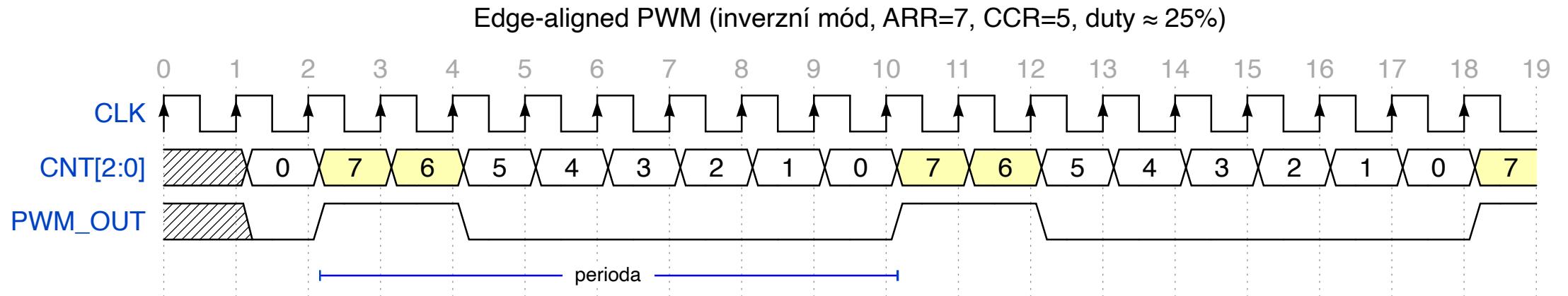
// Output Compare režimy
TIM2->CCMR1 = (0b001 << TIM_CCMR1_OC1M_Pos) // Active on match (CH1)
             | (0b010 << TIM_CCMR1_OC2M_Pos); // Inactive on match (CH2)

TIM2->CCER = TIM_CCER_CC1E | TIM_CCER_CC2E; // Povolit oba výstupy
TIM2->CR1 |= TIM_CR1_OPM; // One-Pulse Mode
TIM2->CR1 |= TIM_CR1_CEN; // Start
```

PWM jako speciální případ komparace

PWM režimu porovnává $CNT < CCRx$, délku pulzu na výstupu určuje poměr $CCRx / ARR$.

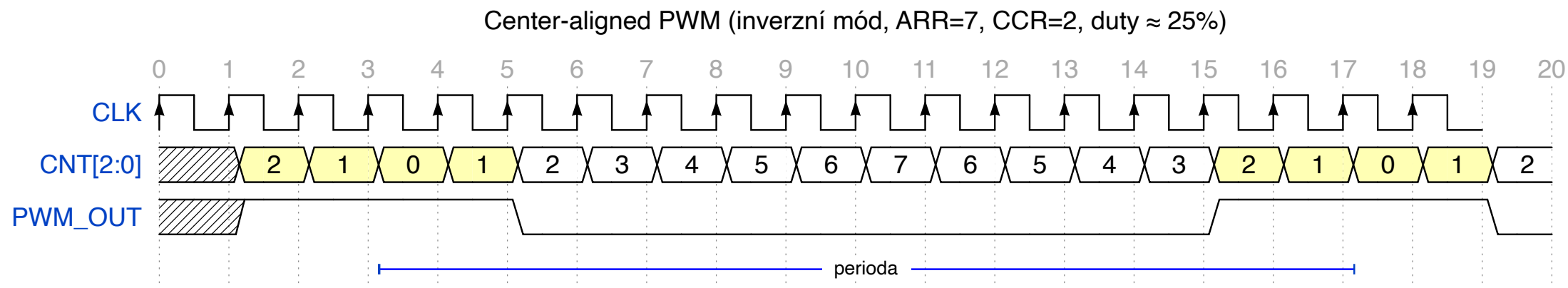
- ARR = perioda
- CCR = šířka pulzu
- PWM Mode 1: aktivní pro $CNT < CCR$, $OCxM = 110$
- PWM Mode 2: inverzní, aktivní pro $CNT > CCR$, $OCxM = 111$



Způsoby generování PWM

Způsob generování PWM ovlivňuje nastavní pole `CMS` (Center-aligned mode selection), umožňující tzv. center-aligned PWM, kde čítač běží nahoru i dolů a PWM je plynulejší.

- `00` - edge-aligned, klasické PWM
- `01` - center-aligned 1, match při směru nahoru
- `10` - center-aligned 2, match při směru dolů
- `11` - center-aligned 3, match při obou směrech



Realizace edge-aligned PWM - TIM1, GPIOA8

Klíčová nastavení, rezervy v ošetření registrů, nastavení autoreload, atd.

Nastavení GPIO

```
// GPIOA8 jako alternativní funkci AF1 (TIM1_CH1)
GPIOA->MODER |= (GPIO_MODER_MODE8_1); // 10: Alternate function mode
GPIOA->AFR[1] |= (0x1U << GPIO_AFRH_AFSEL8_Pos); // AF1 = TIM1_CH1
```

Základní parametry časovače

```
TIM1->PSC = 0; // Prescaler = 0 (taktujeme z APB2)
TIM1->ARR = 7; // Auto-reload
TIM1->CCR1 = 5; // Compare → cca 25 %

TIM1->CR1 |= TIM_CR1_DIR; // DIR = 1 → čítá dolů
```

PWM mode 1 na CH1 a povolení výstupu

```
TIM1->CCMR1 |= (6U << TIM_CCMR1_OC1M_Pos); // 110: PWM mode 1
TIM1->CCMR1 |= TIM_CCMR1_OC1PE;           // preload enable

TIM1->CCER |= TIM_CCER_CC1E;
```

TIM1 musí mít povolen výstup v registru BDTR

```
TIM1->BDTR |= TIM_BDTR_MOE;
```

Spuštění čítače

```
TIM1->EGR = TIM_EGR_UG; // vynutit update event
TIM1->CR1 |= TIM_CR1_CEN; // start
```

Center-aligned PWM

V center-aligned režimu čítač běží nahoru i dolů mezi `0` a `ARR`.

Komparátor `CCRx` překlápí výstup dvakrát za periodu

- jednou při vzestupné a jednou při sestupné hraně
- vytváří symetrické PWM kolem středu periody.

Efektivní frekvence PWM je poloviční oproti edge-aligned módu (protože perioda je dvojnásobná).

- minimalizuje spektrální rušení
- je to preferovaný režim pro řízení motorů.

Generování harmonického signálu PWM

Pomocí PWM můžeme vytvořit unipolární nebo bipolární harmonický nebo jiný signál.

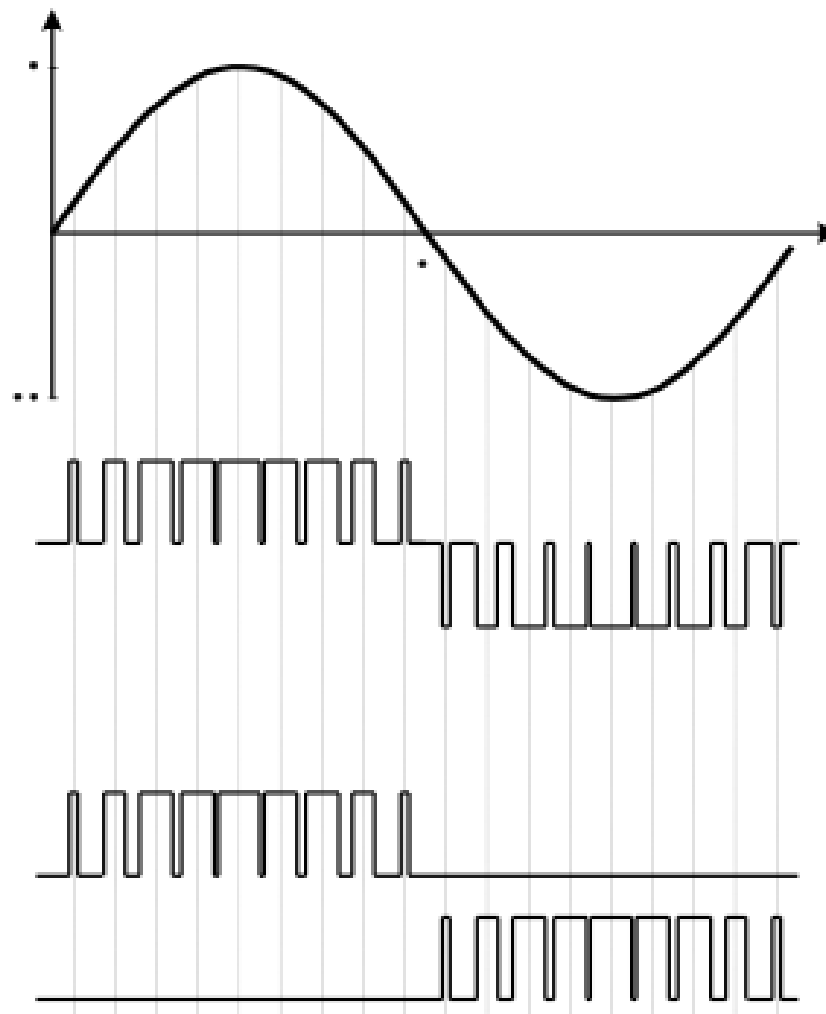
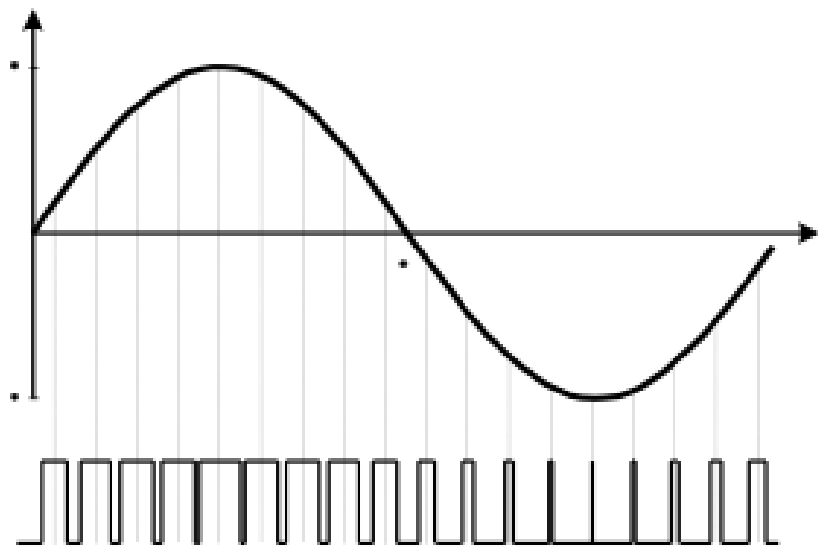
Pro získání průběhu s únosným zkreslením je potřeba minimálně 40 period PWM do periody generovaného průběhu, realizace předpočítanou tabulkou hodnot `CCR`

```
#define PWM_STEPS 256
#define ARR_VALUE 1000

uint16_t sin_table[PWM_STEPS];

for (int i = 0; i < PWM_STEPS; i++) {
    float angle = (2.0f * M_PI * i) / PWM_STEPS;
    float s = (sinf(angle) + 1.0f) / 2.0f; // posun do 0-1
    sin_table[i] = (uint16_t)(s * ARR_VALUE);
}
```

Generování harmonického signálu PWM



Specifika PWM pro řízení motorů

Pokročilé časovače (`TIM1` , `TIM8`) mají pro každý kanál i komplementární výstup (např. `CH1` a `CH1N`).

- Každý pár řídí horní a dolní tranzistor v polovičním můstku.
- Aktivní logika je opačná – když `CH1 == 1` → `CH1N = 0` .

Když jsou párové spínací tranzistory v jedné větvi H-můstku, nikdy nesmějí být sepnuté současně — jinak by se napájecí napětí zkratovalo přímo na zem .

- při vypínání má tranzistor určitý čas zpoždění (t_{off}),
- druhý tranzistor se mezitím už může začít zapínat (t_{on}),
- a v této krátké době mohou být oba částečně vodivé → proudový zkrat, přehřátí, destrukce.

Řešení: vložit „mrtvý čas“

Advanced časovače dokáží automaticky mezi sepnutými horního a dolního tranzistoru vložit krátké zpoždění – dead-time, obvykle v rozmezí:

- 100 ns až 2 μ s, podle výkonové elektroniky,
- řízené v `BDTR`→`DTG` (Dead-Time Generator).

Tedy:



Během „mrtvého času“ jsou oba tranzistory vypnuté, takže proud teče diodami nebo přes motorovou indukčnost.

Prakticky na STM32

Nastavení přes registr `BDTR` :

- `DTG` určuje dead-time v násobcích časové základny timeru.
- Např. při `TIM1_CLK = 84 MHz` a `DTG = 84` → dead-time $\approx 1 \mu\text{s}$.

Zjednodušeně:

```
Dead-time = DTG × t_timer
```

```
TIM1->BDTR |= (84U << TIM_BDTR_DTG_Pos); // 84 taktů deadtime
```

(když je `DTG < 128` , pro vyšší hodnoty se používají jiné prescalery)

Příklad - Komplementární PWM

```
// Nastavení časovače TIM1: PSC=0, ARR=999, CCR1=500
// PA8 → TIM1_CH1 (AF1)
GPIOA->MODER |= GPIO_MODER_MODE8_1;
GPIOA->AFR[1] |= (0x1U << GPIO_AFRH_AFSEL8_Pos);

// PB13 → TIM1_CH1N (AF1)
GPIOB->MODER |= GPIO_MODER_MODE13_1;
GPIOB->AFR[1] |= (0x1U << GPIO_AFRH_AFSEL13_Pos);

// PWM mode 1
TIM1->CCMR1 |= (6U << TIM_CCMR1_OC1M_Pos); // 110 = PWM mode 1

// Povolit výstupy CH1 a CH1N
TIM1->CCER |= TIM_CCER_CC1E | TIM_CCER_CC1NE;

// Nastavit dead-time a povolit hlavní výstup
TIM1->BDTR = (40U << TIM_BDTR_DTG_Pos) | TIM_BDTR_MOE;
```

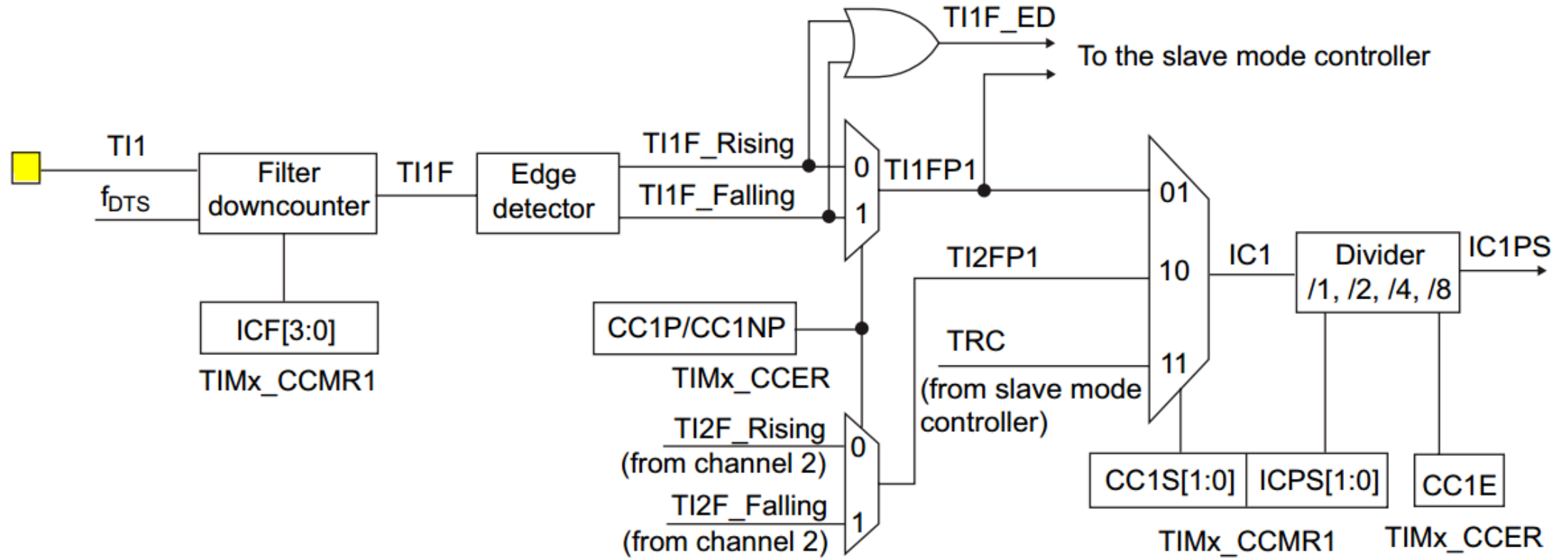
Záchytný systém

Každý kanál časovače (`TIMx_CHy`) lze nastavit do režimu Input Capture.

Tento režim slouží k měření času mezi událostmi (např. mezi hranami vstupního signálu).

Princip:

1. Čítač běží z referenčního taktu (např. 84 MHz / prescaler).
2. Na vstupu (např. `PA0 = TIM2_CH1`) přichází signál.
3. Když se detekuje hrana (rising/falling nebo obojí),
→ hodnota `CNT` se uloží do registru `CCRx` .
4. Rozdíl mezi dvěma zachycenými hodnotami dává periodu signálu.

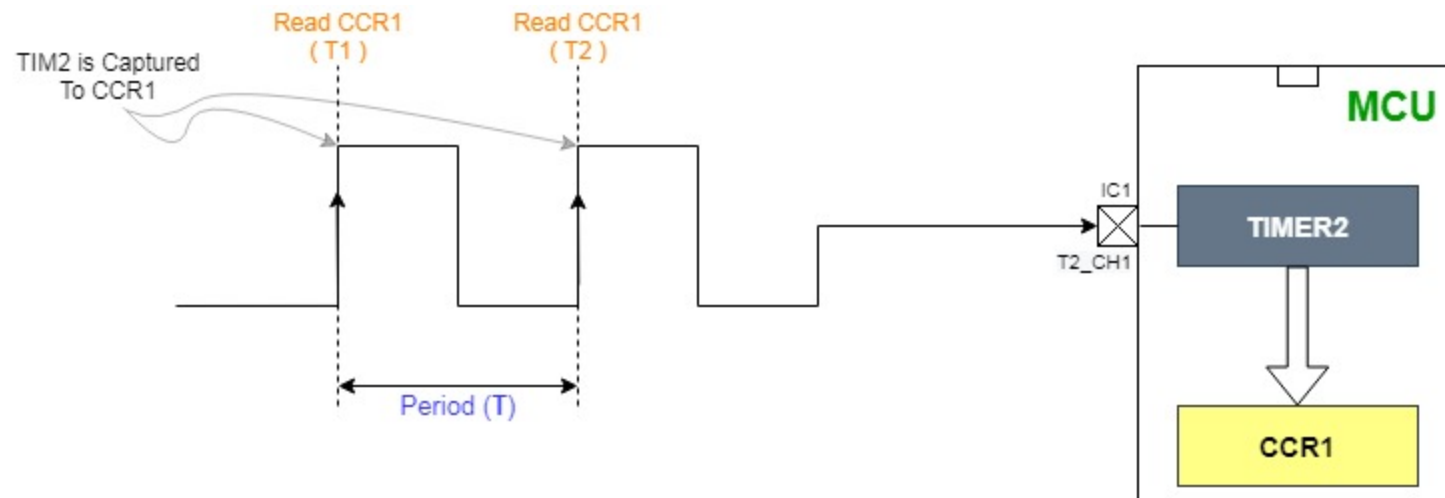


Výpočet periody a frekvence

Pokud známe akt časovače f_{TIM} a rozdíl mezi dvěma záchyty $\Delta = CCR_2 - CCR_1$

Pak

$$T_{signal} = \frac{\Delta}{f_{TIM}}, f_{signal} = \frac{f_{TIM}}{\Delta}$$



Registry pro záchytný systém na STM32F4

Registr	Význam
TIMx_CCMR1	Nastavení módu (Input Capture, filtrace, dělič)
TIMx_CCER	Aktivace kanálu a volba hrany
TIMx_CCRx	Zde se uloží zachycená hodnota
TIMx_SR	Signalizace události CCxIF
TIMx_CNT	Čítač, jehož hodnota se zachytává

Příklad - měření délky periody

Čítač `CNT` běží kontinuálně s frekvencí f_{TIM} . Při detekci hrany na vstupu se aktuální hodnota `CNT` automaticky zkopíruje do registru `CCR1` a nastaví se flag `CC1IF`.

Měření periody:

1. Zachytit první hranu → `CCR1 = t1`
2. Zachytit druhou hranu (další perioda) → `CCR1 = t2`
3. Perioda: $T = \frac{t_2 - t_1}{f_{TIM}}$

Aktivace a konfigurace brány a časovače

```
// Aktivace GPIOA, TIM2
RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;

// PA0 jako AF1 (TIM2_CH1)
GPIOA->MODER |= (2 << (0 * 2)); // AF mode
GPIOA->AFR[0] |= (1 << (0 * 4)); // AF1 = TIM2
```

Nastavení záchytného systému časovače TIM2

```
TIM2->PSC    = 83;           // 84 MHz / (83+1) = 1 MHz = 1 µs rozlišení
TIM2->CCMR1  = TIM_CCMR1_CC1S_0; // CC1S = 01 → CC1 mapován na TI1 (vstup)
TIM2->CCER   = TIM_CCER_CC1E;  // Povolit capture na kanálu 1
                                     // CC1P = 0 (default) → vzestupná hrana
TIM2->CR1    = TIM_CR1_CEN;     // Spustit čítač

// výpočet
while (!(TIM2->SR & TIM_SR_CC1IF)); // čekej na zachycení (CC1IF)
uint32_t first = TIM2->CCR1;
TIM2->SR &= ~TIM_SR_CC1IF;

while (!(TIM2->SR & TIM_SR_CC1IF));
uint32_t second = TIM2->CCR1;
TIM2->SR &= ~TIM_SR_CC1IF;

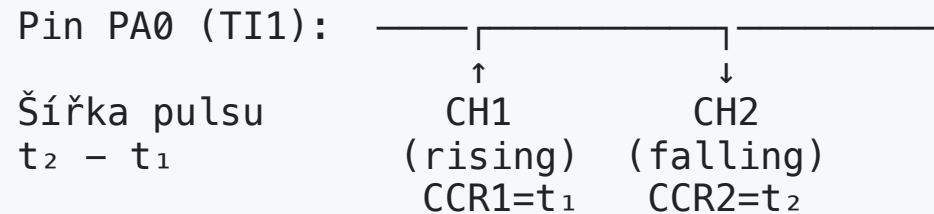
uint32_t delta = second - first; // perioda signálu v µs (při PSC=83)
float freq = 1e6 / delta;        // frekvence v Hz
```

Poznámky:

- Čekání na zachycení může uvíznout, pokud signál nepřichází → v produkčním kódu přidat timeout
- Kontrolovat flag `CC10F` (overcapture) pro detekci ztracených dat při rychlém signálu

Příklad - měření délky pulsu

Pro měření jsou použity dva capture kanály na stejný vstupní pin - jeden pro vzestupnou hranu, druhý pro sestupnou. Rozdíl časů = šířka pulsu.



Konfigurace, GPIO a časovač stejně jako v předchozím příkladu

```
// CH1 → vzestupná hrana, CH2 → sestupná hrana (oba na TI1)
TIM2->CCMR1 = TIM_CCMR1_CC1S_0 // CH1 mapován na TI1
              | TIM_CCMR1_CC2S_1; // CH2 mapován na TI1

TIM2->CCER = TIM_CCER_CC1E // CH1 enable, rising edge (CC1P=0)
             | TIM_CCER_CC2E // CH2 enable
             | TIM_CCER_CC2P; // CH2 falling edge (CC2P=1)

TIM2->CR1 = TIM_CR1_CEN; // Start
```

Měření pulsu

```
uint32_t measure_pulse_width_us(void) {
    uint32_t t1, t2;

    // Čekaj na vzestupnou hranu
    while (!(TIM2->SR & TIM_SR_CC1IF));
    t1 = TIM2->CCR1;
    TIM2->SR &= ~TIM_SR_CC1IF;

    // Čekaj na sestupnou hranu
    while (!(TIM2->SR & TIM_SR_CC2IF));
    t2 = TIM2->CCR2;
    TIM2->SR &= ~TIM_SR_CC2IF;

    // Výpočet šířky (TIM2 je 32-bit, default ARR = 0xFFFFFFFF)
    if (t2 >= t1) {
        return (t2 - t1); // Běžný případ
    } else {
        return (0xFFFFFFFF - t1 + t2 + 1); // Přetečení čítače
    }
}
```

Poznámka: Přetečení nastane jen při pulsech delších než ~71 minut (při 1 MHz).

Vlastnosti a limity zachytávacího systému

1. Rozlišení a minimální měřitelná délka pulzu

Minimální rozlišitelný časový rozdíl je dán frekvencí časovače:

- 84 MHz časovač → rozlišení 11,9 ns
- 1 MHz časovač → rozlišení 1 μ s

Pokud přijde puls široký 500 ns a timer běží jen na 1 MHz (1 μ s tick):

- Časovač vidí 0 nebo 1 tick → velká kvantizační chyba nebo úplně ztracený puls.
- Řešení: zvýšit takt časovače, případně použít rychlejší čítač (např. TIM1 , TIM8) připojený na APB2 .

2. Rychlá sekvence pulsů (příliš krátká perioda)

Pokud přichází impulsy rychleji, než se stihne:

- zpracovat přerušení,
- nebo přečíst `CCR` registr,

→ nový záchyt přepíše starý, než ho SW stihne načíst.

STM32 sice má flag `CCx0F` (Capture Overflow Flag), ale pokud přerušení nestíhá, je měření ztraceno

Řešení:

- Použít DMA, které přenáší hodnoty `CCR` přímo do RAM.
- Nebo použít přerušení pouze každých `N` pulsů (počítat mezery statisticky).
- Případně využít Trigger controller (`ETR`) pro synchronizaci s jiným časovačem.

3. Puls delší než perioda čítače (přetečení)

Když časovač přeteče mezi dvěma záchyty, pak $CCR2 < CCR1$

Rozdíl je třeba korekčně přepočítat přes přetečení.

$$\Delta = (CCR2 - CCR1 + ARR + 1) \bmod (ARR + 1)$$

Příklad

Timer 16bit → max hodnota 65 535

Takt 1 MHz → přeteče každých 65 ms

Pokud perioda signálu > 65 ms (tj. < 15 Hz) → přetečení je nevyhnutelné.

Řešení:

- Použít větší prescaler nebo 32bitový timer (TIM2, TIM5).
- Nebo zachytávat i přetečení (UIF flag) a zahrnout ho do výpočtu.

4. Zpoždění mezi hraniční detekcí a záchytem

Zachycení probíhá hardwarově synchronně s vnitřním taktem,
→ ale ne okamžitě s hranou – je tam 1–2 cykly latence.

Tzn. pokud se měří velmi krátké intervaly (např. v řádu desítek ns),
→ každý kanál může mít malou, ale konzistentní offsetovou chybu.

Řešení:

- Kalibrace: změření referenčního puls a odečtení offsetu.
- Při měření poměru střídy (rising–falling) se chyba často vyruší.

5. Puls s rušením nebo zákmit

Input Capture umí použít digitální filtr (ICF bity v CCMR1).

Rušený signál (Hallova sonda) může způsobit falešnou hranu → úplně chybný výsledek.

Řešení: Nastavit filtr nebo zpracovávat více zachycení a průměrovat. O filtraci více později.

6. Sdílení pinů a synchronizace

U STM32 mohou být piny pro zachytávací režim multiplexované (např. PA0 – TIM2_CH1 nebo WKUP). Nevhodná konfigurace GPIO (pull-up/pull-down) může způsobit trvalou log. 1 → žádné zachycení.

Řešení: Testovat pomocí TIMx->CNT čítače v běhu, jestli se skutečně mění CCR .

Kaskádování časovačů

Každý timer v STM32 má spouštěcí a synchronizační vstup `TRGI` a výstup `TRGO` .

Pomocí těchto signálů lze časovače propojit tak, aby jeden řídil druhý.

Typicky:

```
TIMx -> TRGO  →  TIMy -> TRGI
```

Tedy:

- Master `TIMx` generuje spouštěcí událost (update, compare match, apod.)
- Slave `TIMy` reaguje na ni podle svého nastavení v registru `SMCR` (Slave Mode Control Register)

Důležité registry

Každý `TIMx` má Slave Mode Control Register (`SMCR`), kde se dá nakonfigurovat, co se má stát při detekci triggeru (interního nebo externího).

`SMCR.SMS`

- Určuje režim slave časovače (např. Reset, Gated, Trigger, External clock)

`SMCR.TS`

- Zdroj triggerovacího signálu (např. `ITR0 = TIM1_TRG0` , `ITR1 = TIM2_TRG0` , ...)

`CR2.MMS`

- Určuje, co master vysílá na výstupu `TRG0` (např. update event, compare event, output reference, ...)

Možné režimy kaskádování

1. Reset Mode

`SMS = 100` - Slave timer se vynuluje při každém triggeru z masteru.

→ vhodné pro generování „vnořených“ časových oken (např. měření period signálu).

`TIM2` (master) → update event → reset `TIM3` (slave)

2. Gated Mode

`SMS = 101` - Slave běží jen když je aktivní signál TRGI.

→ využívá se např. pro měření délky pulzu.

`TRGI = HIGH` → čítač běží

`TRGI = LOW` → čítač stojí

3. Trigger Mode

`SMS = 110` - Slave se spustí jedním impulsem z masteru (poté běží samostatně).
→ např. spuštění několika PWM časovačů současně.

4. External clock mode

TRGI se použije jako taktovací signál čítače.

→ umožňuje vytvořit delší periodu (např. master přeteče → slave se inkrementuje).

Typicky:

`TIM2` overflow → `TIM3++`

Příklad - nulování čítače s prvním pulsem

Chci, aby první příchod pulsu (např. z externího čidla) vynuloval čítač — tedy, aby čítač začal měřit relativně od něj.

1. V registru `SMCR` nastavím:

- `SMS = 100` (Reset mode)
- `TS = 101` (např. `TI1FP1` jako trigger source)

2. V `CCMR1` povolím capture na kanálu 1, a případně nastavím filtr `IC1F` .

3. Výsledek:

- Každý příchod pulsu na CH1 (pokud splní filtr) → `CNT = 0`
- PWM, měření doby nebo input capture pak běží synchronně s tímto resetem.

Příklad - prodloužení periody

Chci mít velmi pomalé PWM (~1 Hz), ale nechci použít velký prescaler.

1. Nastavím `TIM2` (master):

- běží rychleji, např. 10 kHz
- každým overflowem generuje `TRG0`

2. Nastavím `TIM3` (slave):

- `SMS = 111` (External Clock Mode 1)
- `TS = ITR1` (`TIM2_TRG0`)
- čítač `TIM3` inkrementuje každým přetečením `TIM2`

→ výsledný efekt: `TIM3` běží jako 16bit čítač s krokem 10 kHz, tj. efektivní perioda až stovky sekund, bez ztráty rozlišení.

Digitální filtr (ICxF)

V STM32F4 (a obecně u většiny ARM timerů) má každý input capture kanál (CHx) možnost aktivovat digitální filtr pro vstupní signál.

Účelem je odfiltrovat krátké zákmity nebo rušení na vstupním pinu (např. z enkodéru nebo při galvanickém oddělení optočlenem).

Bez filtru by se totiž na vstupu mohla: **(a)** zachytit falešná hrana, **(b)** resetovat čítač dřív než má, **(c)** nebo vygenerovat několik capture událostí místo jedné.

Každý filtr má dva parametry:

- vzorkovací frekvenci (jak rychle sleduje vstup),
- počet po sobě jdoucích shodných vzorků, které musí mít stejnou úroveň, aby to uznal jako platnou hranu.

Reprezentace v registru:

V registru **CCMRx** (Capture/Compare Mode Register) jsou 4 bity **ICxF [3:0]** .

Každá kombinace (0–15) odpovídá určitému filtračnímu času, tedy délce pulzu, kterou musí vstup udržet stabilní, aby byl uznán.

ICxF	Popis	Efektivní filtr
0b0000	filtr vypnut	reaguje okamžitě
0b0001–0b0011	2/4 vzorky, f_{DTS}	mírné zpoždění
0b0100–0b0111	8 vzorků při $f_{DTS}/2$ až $/8$	filtruje rušení v řádu ns– μ s
0b1000–0b1111	16 vzorků při $f_{DTS}/32$ až $/128$	velmi silný filtr (pomalejší reakce)

Praktický příklad

Řekněme, že timer běží na 84 MHz → perioda = 11,9 ns.

Pokud nastavíš:

```
IC1F = 0b1110 // 8 vzorků, f_DTS = fCK_INT / 32
```

→ vzorkovací frekvence je $84 \text{ MHz} / 32 = 2,625 \text{ MHz}$

→ perioda vzorku $\approx 381 \text{ ns}$

→ 8 vzorků musí být stejných → vstup musí být stabilní $\sim 3 \mu\text{s}$, aby byl považován za platný.

To znamená, že:

- krátké zákmity (např. 100 ns) se ignorují,
- ale reálný signál s periodou $10 \mu\text{s}$ se už zachytí normálně.

3. Softwarové měření času v embedded systémech

Motivace

V mnoha aplikacích:

- je potřeba změřit dobu trvání události, interval nebo zpoždění,
- ale není k dispozici volný HW časovač (TIM),
- nebo je nutné měřit s rozlišením menším než $1\ \mu\text{s}$, tedy s přístupem přímo k jádru CPU.

Příklady:

- odhad spotřeby funkcí v běhu (profiling),
- softwarový timeout,
- přesné řízení časování při bitbangu,
- validace ISR latence.

Základní softwarové metody

1. Zpoždění pomocí smyčky

Nejjednodušší, ale nejméně přesné.

```
void delay_cycles(uint32_t n) {  
    while(n-->0) {  
        __NOP(); // nebo prázdná smyčka  
    }  
}
```

- ✓ Jednoduché, nezabírá HW.
- ✗ Silně závislé na optimalizaci překladače a frekvenci CPU.
- ✗ Blokující — CPU nic jiného nedělá.

Vhodné pouze pro hrubé ladění nebo časování při fixním taktu.

2. Odměřování podle systémového ticku

Pokud běží `SysTick` (typicky 1 ms), lze měřit čas v ms.

```
volatile uint32_t ms_ticks = 0;

void SysTick_Handler(void) {
    ms_ticks++;
}

uint32_t millis(void) {
    return ms_ticks;
}

void delay_ms(uint32_t t) {
    uint32_t start = millis();
    while ((millis() - start) < t);
}
```

Inicializace SysTicku

```
#include "stm32f4xx.h"

void SysTick_Init(void)
{
    // SysTick běží z jádrového taktu (např. 84 MHz)
    SysTick->LOAD = 0xFFFFFF;           // max hodnota (24 bitů)
    SysTick->VAL = 0;                   // reset čítače
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk // clock = CPU clock
                  | SysTick_CTRL_ENABLE_Msk; // povolit čítač
}
```

- ✓ Stabilní, pokud systém běží s fixním taktem.
- ✓ Integrovatelné do běžného firmware.
- ✗ Rozlišení omezeno periody SysTicku .
- ✗ Měření krátkých intervalů (<1 ms) není přesné.

3. Použití DWT (Data Watchpoint and Trace Unit)

DWT je součástí Cortex-M3/M4/M7 jader, a obsahuje cyklový čítač (CYCCNT), který měří přímo počet taktů CPU od povolení.

Inicializace DWT

```
#include "core_cm4.h"

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

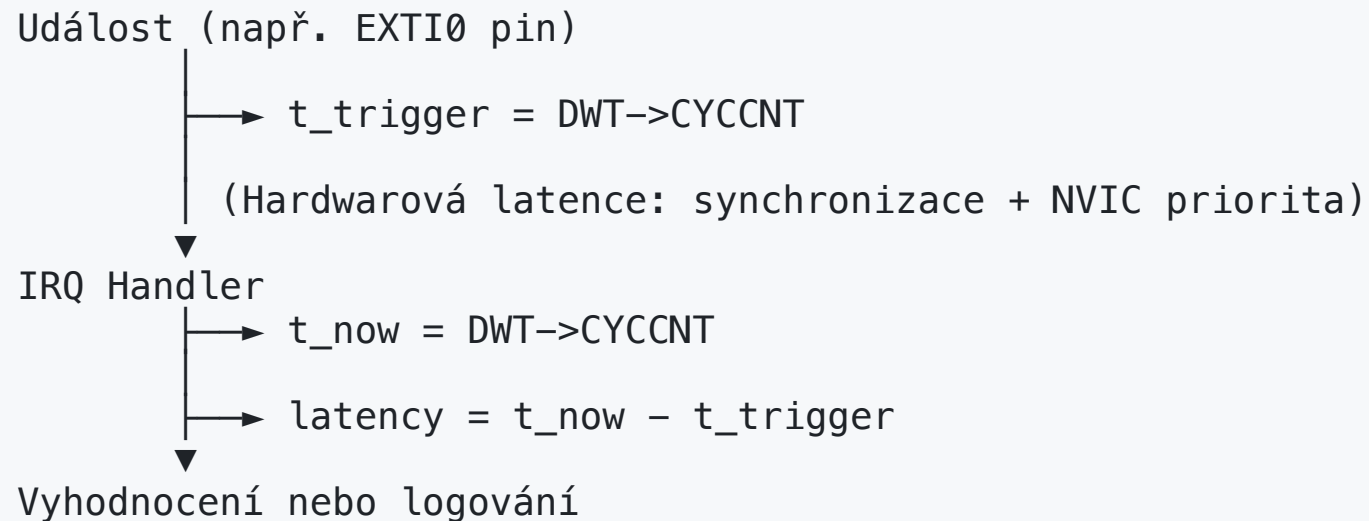
Měření intervalu

```
uint32_t DWT_GetCycles(void) {  
    return DWT->CYCCNT;  
}  
  
float measure_function(void (*fn)(void)) {  
    uint32_t start = DWT_GetCycles();  
    fn();  
    uint32_t end = DWT_GetCycles();  
    return (float)(end - start) / SystemCoreClock; // v sekundách  
}
```

- ✓ Rozlišení - 1 cyklus CPU (např. 11.9 ns při 84 MHz)
- ✓ Přesnost - Prakticky ideální, běží v taktu jádra
- ✓ Energetická náročnost - Nízká, jednotka `DWT` běží i v debug módu
- ✗ Omezení - `DWT` může být zablokováno v některých low-end MCU (např. F0)

Příklad - měření latence ISR

1. V okamžiku události se zaznamená čas vzniku – `t_trigger` .
2. V ISR se změří aktuální `t_now` hodnota čítače `DWT->CYCCNT` .
3. Rozdíl `t_now - t_trigger` udává latenci v taktech jádra.



Nastavení DWT čítače

```
// Povolit DWT a cyklový čítač
CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;
DWT->CYCCNT = 0;
DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;
```

Simulace události

```
volatile uint32_t t_trigger = 0;
volatile uint32_t latency = 0;

void trigger_event(void) {
    // Simulace události (např. externí hrana)
    t_trigger = DWT->CYCCNT;
    EXTI->SWIER |= EXTI_SWIER_SWIER0; // Softwarově vyvolané přerušení
}
```

Obsluha přerušení s měřením

```
void EXTI0_IRQHandler(void) {
    uint32_t t_now = DWT->CYCCNT;
    latency = t_now - t_trigger; // počet taktů mezi triggerem a IRQ

    // Vyčištění příznaku přerušení
    EXTI->PR = EXTI_PR_PR0;
}
```

Při čistě externí události (např. pin `EXTI`) se dá `t_trigger` zaznamenat např. pomocí:

- přerušení z jiného kanálu stejného signálu,
- nebo v simulaci — generováním SW přerušení.

Při reálném hardwarovém měření lze signál přivést na osciloskop a porovnat s momentem, kdy MCU vyvolá GPIO v handleru.

4. Watchdog časovače

Co je watchdog timer?

Watchdog timer (WDT) je hardwarový časovač, který slouží jako **bezpečnostní mechanismus** pro detekci a obnovu ze softwarových chyb.

Princip činnosti

1. Watchdog běží nezávisle na hlavním programu
2. Software musí periodicky "krmit" (resetovat) watchdog
3. Pokud software watchdog nenakrmí včas → **reset MCU**

Typické příčiny selhání softwaru

- Nekonečná smyčka
- Deadlock (uváznutí)
- Přetečení zásobníku
- Poškození paměti
- Chyba v přerušení

Proč používat watchdog?

Spolehlivost embedded systémů

- Embedded zařízení často běží bez dozoru (remote IoT, průmyslové aplikace)
- Restart je lepší než zamrzlý systém
- Watchdog poskytuje **poslední linii obrany**

Typické scénáře použití

- IoT senzory v terénu
- Průmyslové řídicí systémy
- Automobilová elektronika
- Medicínské přístroje
- Spotřební elektronika

Typy watchdog timerů v STM32

STM32 mikrokontroléry obsahují dva typy watchdog timerů:

Vlastnost	IWDG	WWDG
Název	Independent Watchdog	Window Watchdog
Zdroj hodin	LSI (~32 kHz)	APB1 (PCLK1)
Nezávislost	Plně nezávislý	Závisí na APB1
Aktivita v sleep	Běží i ve Stop/Standby	Zastaví se
Časové okno	Ne (jen timeout)	Ano (min + max)
Early warning	Ne	Ano (EWI)

IWDG – Nezávislý Watchdog

- Používá svůj vlastní, **nezávislý interní nízkorychlostní oscilátor** LSI (typicky 32 kHz)
- Zůstává aktivní i v režimech spánku (Stop, Standby) – pokud není zakázán
- Jedná se o **12bitový čítač s odpočítáváním** (down-counter)
- Reset nastane, pokud čítač dosáhne hodnoty `0x000`

Klíčová výhoda

Nezávislost na systémových hodinách → watchdog funguje i při selhání hlavního oscilátoru

Základní omezení

IWDG kontroluje pouze to, zda aplikace stále běží a aktualizuje čítač. **Nedokáže detekovat**, že se aplikace zacyklila v nekonečné, ale rychlé smyčce, která čítač aktualizuje příliš často.

IWDG – registry

Registr	Název	Funkce
IWDG_KR	Key Register	Ovládání watchdogu (start, reload, unlock)
IWDG_PR	Prescaler Register	Nastavení předděličky (0-7)
IWDG_RLR	Reload Register	Hodnota pro reload (12-bit)
IWDG_SR	Status Register	Příznaky busy (PVU, RVU)

Klíčové hodnoty pro IWDG_KR

- `0x5555` – Unlock (povolení zápisu do PR a RLR)
- `0xAAAA` – Reload (nakrmení watchdogu)
- `0xCCCC` – Start (spuštění watchdogu)

IWDG – výpočet timeoutu

Timeout watchdogu se vypočítá podle vzorce:

$$T_{out} = \frac{1}{f_{LSI}} \times 2^{PR+2} \times (RL + 1)$$

Kde:

- $f_{LSI} \approx 32000$ Hz (interní LSI oscilátor)
- PR = hodnota prescaleru (0-7)
- RL = hodnota reload registru (0-4095)

Příklad

Pro $PR = 4$ (dělička 64) a $RL = 500$:

$$T_{out} = \frac{1}{32000} \times 64 \times 501 \approx 1.003 \text{ s}$$

IWDG – rozsah timeout hodnot

Prescaler (PR)	Dělička	Min timeout	Max timeout
0	/4	0.125 ms	512 ms
1	/8	0.25 ms	1.024 s
2	/16	0.5 ms	2.048 s
3	/32	1 ms	4.096 s
4	/64	2 ms	8.192 s
5	/128	4 ms	16.384 s
6	/256	8 ms	32.768 s
7	/256	8 ms	32.768 s

Poznámka: Hodnoty jsou přibližné, LSI má toleranci $\pm 5\%$

IWDG – inicializace

Cílem je nastavit IWDG na timeout přibližně 500 ms.

```
void IWDG_Init(void) {  
    // 1. Povolení zápisu do registrů (Write Access)  
    IWDG->KR = 0x5555;  
  
    // 2. Nastavení pre-scaleru (děličky) na /32  
    // Dělička /32 je PR[2:0] = 0x03  
    IWDG->PR = 0x03;  
  
    // 3. Nastavení hodnoty pro reload – 500  
    // IWDG_RLR (Reload Register) – 12 bitů  
    IWDG->RLR = 500;  
  
    // 4. Počkat na aktualizaci registrů (busy flag)  
    while (IWDG->SR & (IWDG_SR_PVU | IWDG_SR_RVU));  
  
    // 5. Start Watchdogu  
    IWDG->KR = 0xCCCC;  
}
```

IWDG – použití

```
void IWDG_Feed(void) {
    // Reload čítače – klíčová hodnota 0xAAAA
    IWDG->KR = 0xAAAA;
}





// Příklad použití v hlavní smyčce
int main(void) {
    SystemInit();
    IWDG_Init();

    while(1) {
        // Hlavní kód aplikace
        ProcessSensors();
        HandleCommunication();
        UpdateDisplay();

        // Na konci každé iterace reload watchdogu
        IWDG_Feed();
    }
}
```

IWDG – best practices

Kde krmit watchdog

-  V hlavní smyčce po dokončení všech kritických operací
-  V RTOS: dedikovaný watchdog task s nejnižší prioritou
-  V přerušení (ISR) – skryje problémy v hlavním kódu
-  V timer callback – nedetekuje zamrznutí hlavní smyčky

Doporučení

```
// Správně: krmení až po úspěšném průchodu všemi moduly
if (SensorOK && CommOK && SystemOK) {
    IWDG_Feed();
}

// Špatně: bezpodmínečné krmení
IWDG_Feed(); // Nedetekuje částečné selhání
```

WWDG - Okénkový Watchdog

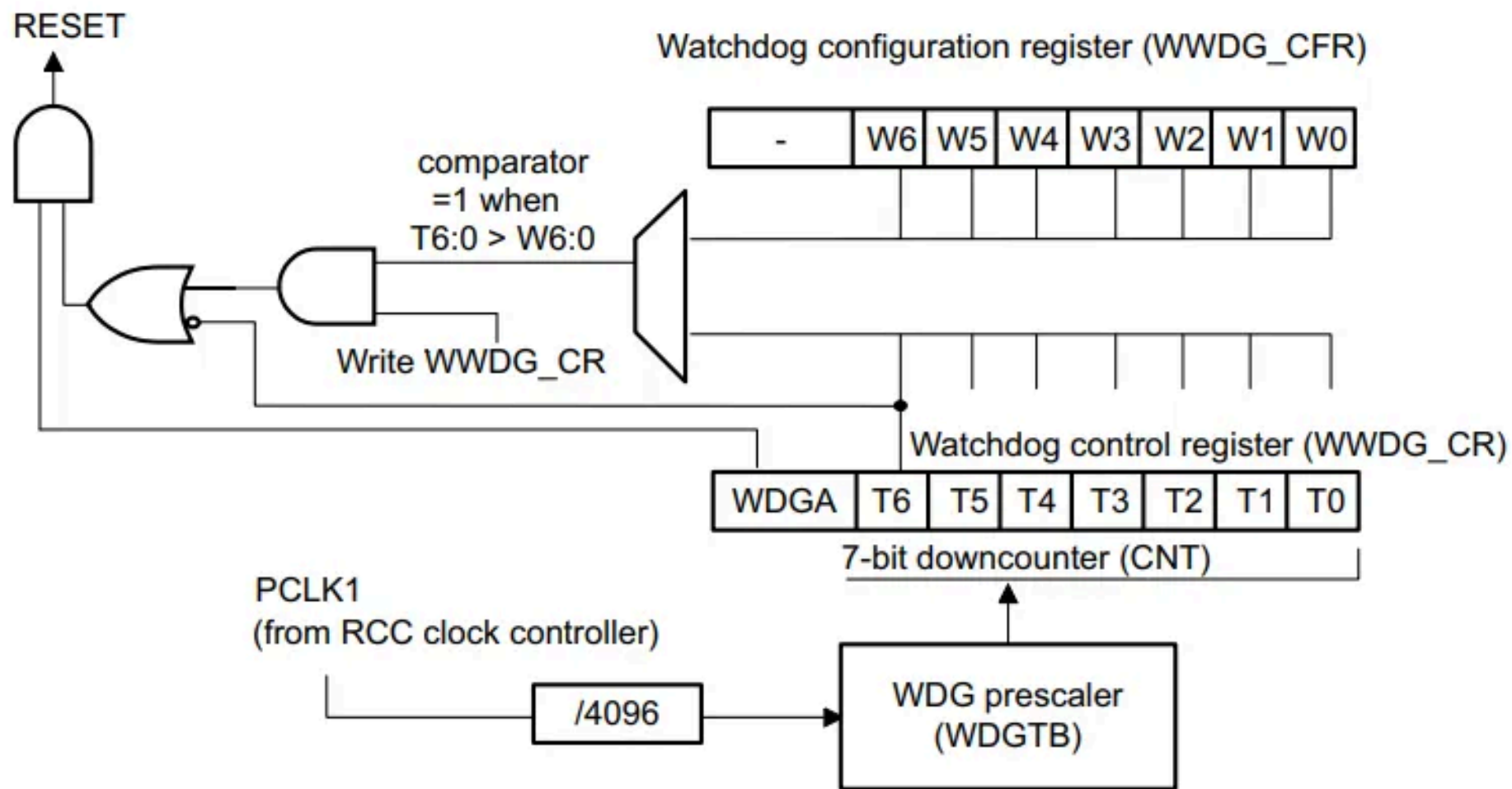
WWDG vyžaduje reload čítače v rámci přesně definovaného časového okna.

- Používá hodiny periférií (APB1 / PCLK1)
- **Zastaví se** v režimech spánku (Stop, Standby)
- 7bitový čítač (T [6:0]), hodnoty 0x40 až 0x7F
- Reset nastane při:
 - Reload **příliš brzy** (hodnota čítače > okno W)
 - Reload **příliš pozdě** (čítač dosáhl 0x3F)

Klíčová výhoda

Early Wakeup Interrupt (EWI) – přerušení před resetem umožňuje uložit kritická data

WWDG – princip časového okna



WWDG – registry

Registr	Název	Funkce
WWDG_CR	Control Register	Čítač T[6:0] + aktivace WDGA
WWDG_CFR	Configuration Register	Okno W[6:0] + dělička + EWIF
WWDG_SR	Status Register	Příznak EWIF

WWDG_CR (Control Register)

- Bit 7 (WDGA): Aktivace watchdogu (1 = enabled)
- Bits 6:0 (T): Hodnota čítače (0x40-0x7F platné)

WWDG_CFR (Configuration Register)

- Bit 9 (EWI): Early Wakeup Interrupt enable
- Bits 8:7 (WDGTB): Prescaler (00=/1, 01=/2, 10=/4, 11=/8)
- Bits 6:0 (W): Hodnota okna

WWDG – výpočet timeoutu

$$T_{timeout} = \frac{1}{f_{PCLK1}} \times 4096 \times 2^{WDGTB} \times (T[5:0] + 1)$$

Příklad

Pro $f_{PCLK1} = 42 \text{ MHz}$, $WDGTB = 3$ (dělička 8), $T = 0x7F$ (127):

$$T_{timeout} = \frac{1}{42000000} \times 4096 \times 8 \times 64 \approx 49.9 \text{ ms}$$

Časové okno

- **Minimální čas** (příliš brzy): určen hodnotou W
- **Maximální čas** (příliš pozdě): určen hodnotou T při startu

WWDG – inicializace

```
#define WWDG_PRESCALER 0x03 // Dělička /8
#define WWDG_WINDOW 80 // Okno W = 80
#define WWDG_COUNTER 0x7F // Počáteční hodnota = 127

void WWDG_Init(void) {
    // 1. Povolení hodin pro WWDG
    RCC->APB1ENR |= RCC_APB1ENR_WWDGEN;

    // 2. Konfigurace: Dělička a Hodnota Okna
    WWDG->CFR = (WWDG_PRESCALER << 7) | WWDG_WINDOW;

    // 3. Volitelné: Povolení EWI přerušení
    // WWDG->CFR |= WWDG_CFR_EWI;
    // NVIC_EnableIRQ(WWDG_IRQn);

    // 4. Start WWDG (nastavení čítače + aktivace)
    WWDG->CR = WWDG_COUNTER | WWDG_CR_WDGA;
}
```

WWDG – použití

```
void WWDG_Feed(void) {
    // Reload čítače – musí být v okně!
    // Hodnota musí být > 0x3F a <= aktuální čítač
    WWDG->CR = WWDG_COUNTER | WWDG_CR_WDGA;
}

// Příklad: kontrola, zda jsme v okně
void WWDG_SafeFeed(void) {
    uint8_t counter = WWDG->CR & 0x7F;

    // Krmit pouze pokud jsme v okně (counter <= W)
    if (counter <= WWDG_WINDOW && counter > 0x3F) {
        WWDG->CR = WWDG_COUNTER | WWDG_CR_WDGA;
    }
}
```

WWDG – Early Wakeup Interrupt (EWI)

EWI umožňuje provést kritické operace těsně před resetem.

```
void WWDG_IRQHandler(void) {
    // Vymazat příznak přerušení
    WWDG->SR &= ~WWDG_SR_EWIF;

    // Poslední šance – uložit kritická data!
    SaveCriticalDataToBackupRAM();
    LogErrorState();

    // Můžeme zkusit nakrmit WWDG (pokud to má smysl)
    // WWDG_Feed();

    // Nebo nechat proběhnout reset...
}

void WWDG_Init_WithEWI(void) {
    RCC->APB1ENR |= RCC_APB1ENR_WWDGEN;
    WWDG->CFR = (WWDG_PRESCALER << 7) | WWDG_WINDOW | WWDG_CFR_EWI;
    NVIC_EnableIRQ(WWDG_IRQn);
    WWDG->CR = WWDG_COUNTER | WWDG_CR_WDGA;
}
```

Kdy použít který watchdog?

Scénář	Doporučení
Low-power aplikace (Stop/Standby)	IWDG – běží nezávisle
Detekce příliš rychlého kódu	WWDG – okénkový režim
Potřeba varování před resetem	WWDG – EWI přerušení
Nezávislost na systémových hodinách	IWDG – vlastní LSI
Přesné časování	WWDG – APB1 hodiny
Maximální spolehlivost	Oba – kombinace

Kombinace obou watchdogů

```
// IWDG: dlouhý timeout, detekce zamrznutí
// WWDG: krátký timeout, detekce timing anomálií
IWDG_Init(2000); // 2 sekundy
WWDG_Init(50); // 50 ms okno
```

Typické chyby při použití watchdogů

✗ Krmení v přerušení

```
void TIM2_IRQHandler(void) {  
    IWDG_Feed(); // ŠPATNĚ! Skryje problémy v main()  
}
```

✗ Příliš dlouhý timeout

```
IWDG->RLR = 4095; // 32+ sekund – příliš dlouho na detekci
```

✗ Krmení bez kontroly stavu systému

```
while(1) {  
    DoSomething();  
    IWDG_Feed(); // Krmí i když DoSomething() selhalo  
}
```

5. Speciální časovací obvody

TPU – Time Processing Unit (NXP/Freescale)

TPU je samostatný, mikroprogramovatelný periferní procesor určený k přesnému řízení časově kritických úloh. Vznikl v době, kdy klasické MCU měly slabé časovače — TPU odlehčuje CPU tím, že zvládá všechny časové operace autonomně.

➡ Používá se hlavně u MCU z řad MPC56xx, MPC57xx (PowerPC architektura) nebo S32K.

TPU nepočítá čas přímo, ale provádí mikroprogramové rutiny reagující na časové události:

- Každý kanál má časový komparátor.
- Při dosažení určitého času se vyvolá mikroinstrukční sekvence (uložená v TPU ROM).
- Ta může změnit výstupní pin, zapsat data do RAM, přepočítat novou periodu atd.
- CPU nemusí vůbec zasahovat – vše běží deterministicky.

Mikroinstrukce TPU / eTPU – přehled

TPU (a ještě více eTPU) má vlastní mikroinstrukční procesor – 24bitové (TPU) nebo 32bitové (eTPU) instrukce. Tyto instrukce se nekompilují klasickým překladačem C, ale pomocí speciálního assembleru nebo překladače NXP (např. eTPU2+ Development Toolchain od NXP nebo 3rd-party jako AshWare).

Typ	Popis	Příklad
LD / ST	Načtení / uložení dat z Channel RAM nebo parametrů	LD A, P
CMP / BRA	Podmíněné větvení (branch)	CMP A, B; BRA GT, label
ADD / SUB / AND / OR	Aritmetika a logika	ADD A, #1
MOV / CLR / SET	Manipulace s registry	SET FLAG0
OC / IC	Nastavení compare/capture události	OC ON, MATCH_A
WAIT / LINK	Čekání na událost nebo volání další rutiny	WAIT MATCH_A
UPDATE	Aktualizace výstupu (např. PWM)	UPDATE MATCH_B
END	Ukončení běhu kanálu do další události	END

Příklad eTPU mikroprogramu

```
; PERIOD      - PWM period
; DUTY        - PWM high time
; STATE       - 0=low, 1=high

ENTRY:
LD   A, TCR1           ; načti aktuální čas
ADD  A, PERIOD         ; spočítej čas konce periody
ST   A, MATCH_B       ; nastav compare B (konec periody)

LD   A, TCR1
ADD  A, DUTY           ; spočítej čas přechodu high→low
ST   A, MATCH_A       ; nastav compare A (přepnutí)

SET  OUTPUT_HIGH      ; začni PWM v HIGH
WAIT MATCH_A          ; čekej na konec HIGH fáze
SET  OUTPUT_LOW       ; přepni výstup LOW
WAIT MATCH_B          ; čekej na konec periody
LINK ENTRY            ; začni znovu
```