COPYRIGHT

DISCLAIMER

This content is provided without warranty or representation of any kind. The use of the content is entirely at your own risk and Stellenbosch University (SU) will have no liability directly or indirectly as a result of this content.

The content must not be assumed to provide complete coverage of the particular study material. Content may be removed or changed without notice.

The video is of a recording with very limited post-recording editing. The video is intended for use only by SU students enrolled in the particular module.

UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

100
1918 · 2018

forward together · saam vorentoe · masiye phambili

Computer Systems / Rekenaarstelsels 245 - 2020

Lecture 14

Timers and Counter: Part 2

Tydhouers en Tellers: Part 2

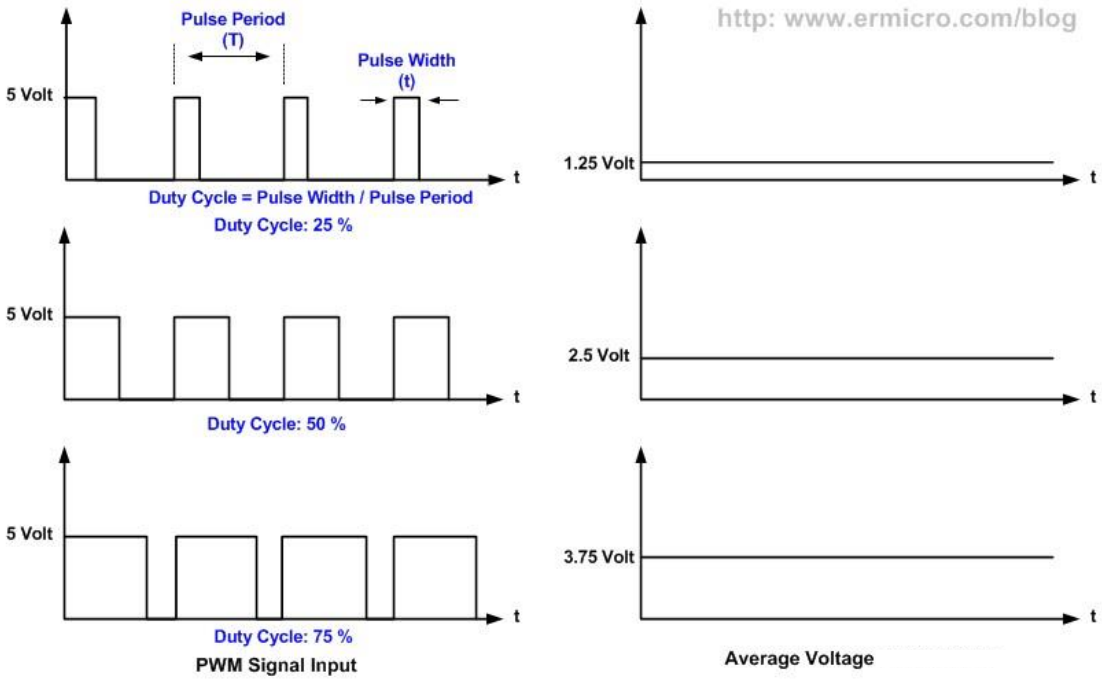
Dr Rensu Theart & Dr Lourens Visagie

Lecture Overview

- PWM mode
  - Edge-aligned
  - Center-aligned
- One-pulse mode
- SysTick Timer
- Watchdog Timers
- RTC

Pulse-width modulation (PWM) / Pulswydte modulasie (PWM)

- **Pulse-width modulation** (PWM) is a technique where the width of dital pulses is adjusted to generated different average DC voltages.
- It refers to a square wave that has a programmable pulse width or **duty cycle**.
- Most microcontrollers have a built-in timer that can be used to generate a PWM signal.



PWM Timing Diagram

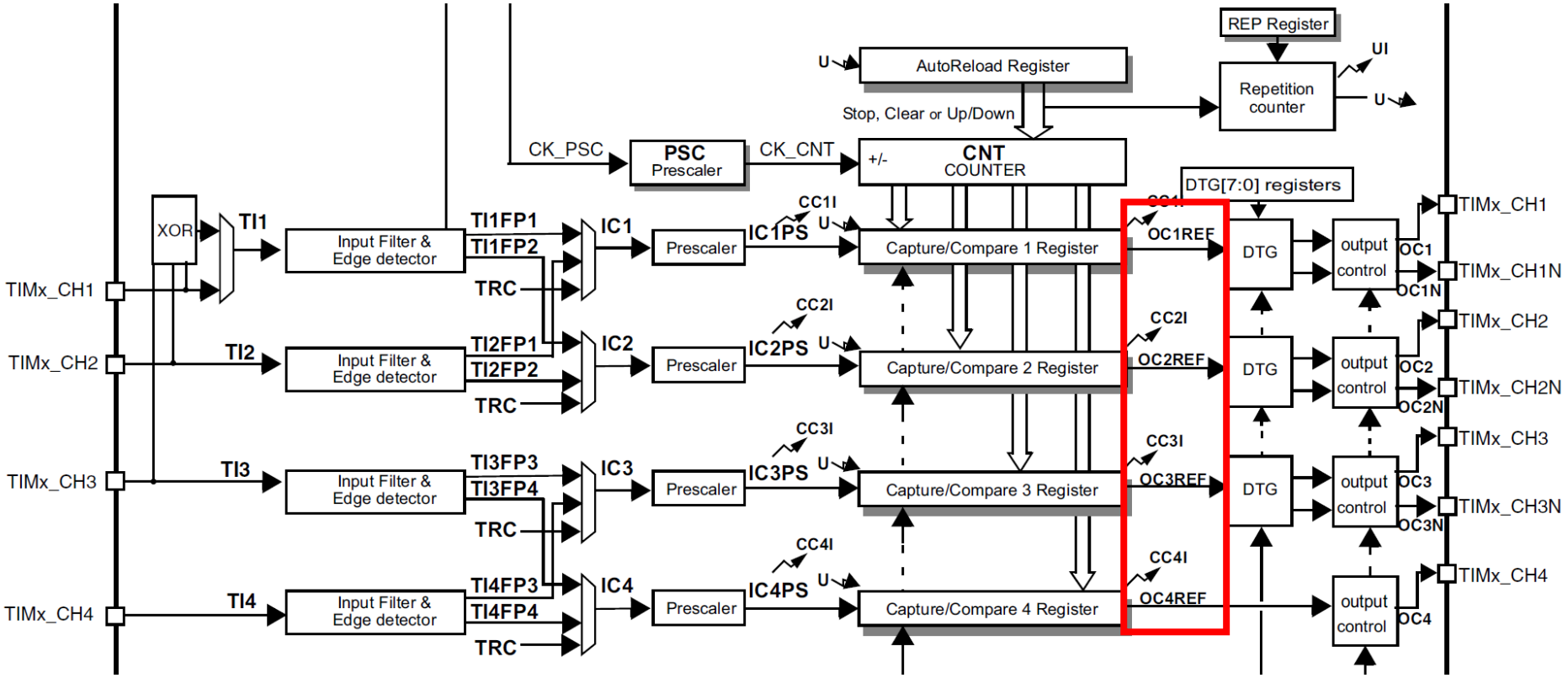
PWM on STM32F4

- The timers on the STM32F4 also support PWM mode, which allows you to generate PWM signals.
  - Frequency determined by the value of the TIMx_ARR register.
  - Duty cycle determined by the value of the TIMx_CCRx register.
- The output PWM signal is called Output Compare reference **OCxREF**.

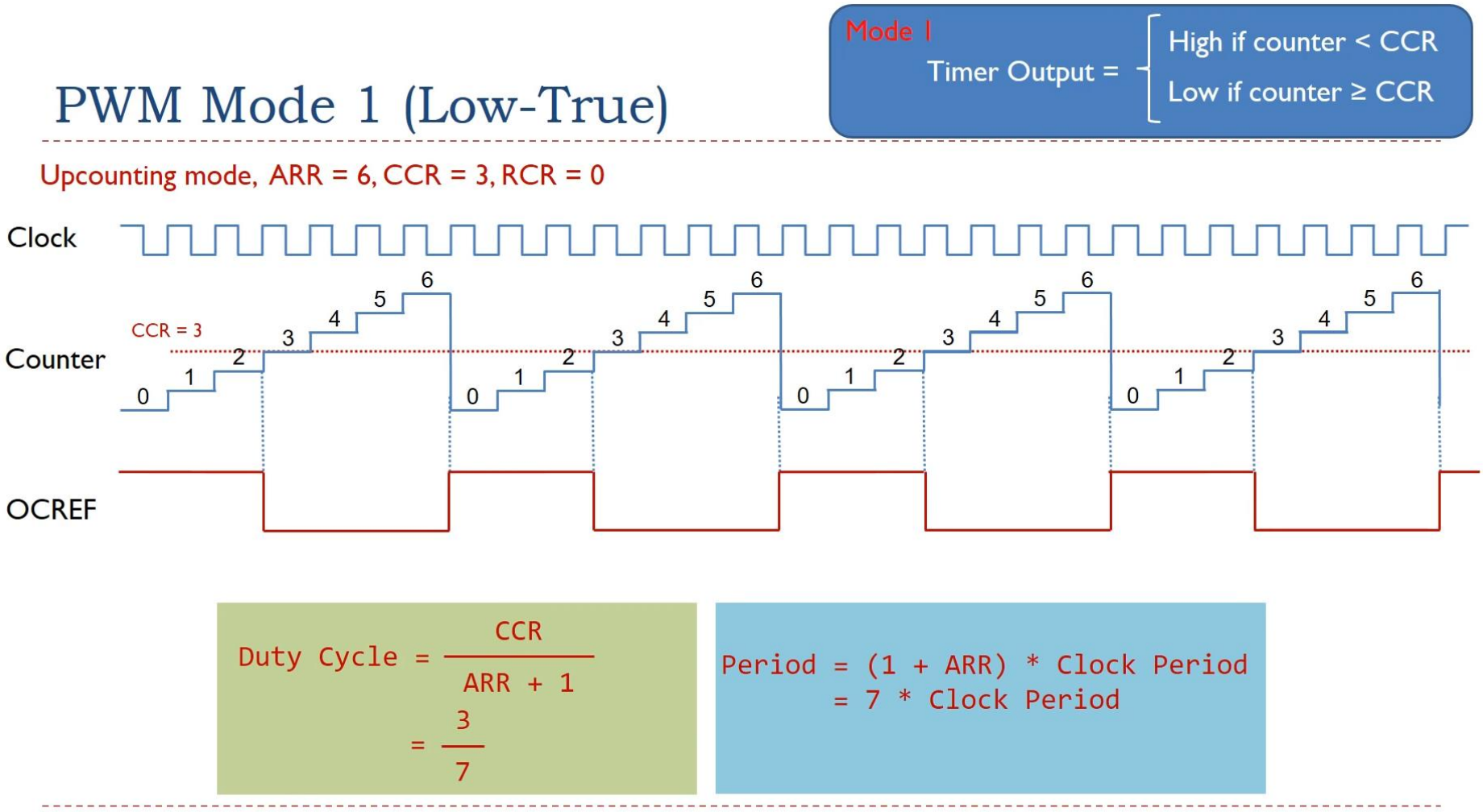
	Upcounting (TIMx_CNT<TIMx_CCRx)	Downcounting (TIMx_CNT>TIMx_CCRx)
PWM Mode 1	OCxREF = 1	OCxREF = 0
PWM Mode 2	OCxREF = 0	OCxREF = 1

- In PWM mode, TIMx_CNT and TIMx_CCRx are always compared to determine whether TIMx_CCRx ≤TIMx_CNT (for upcounting) or TIMx_CNT ≤TIMx_CCRx (for downcounting).
- The timer is able to generate PWM in edge-aligned mode or center-aligned mode.

PWM on STM32F4



PWM Mode 1 Example



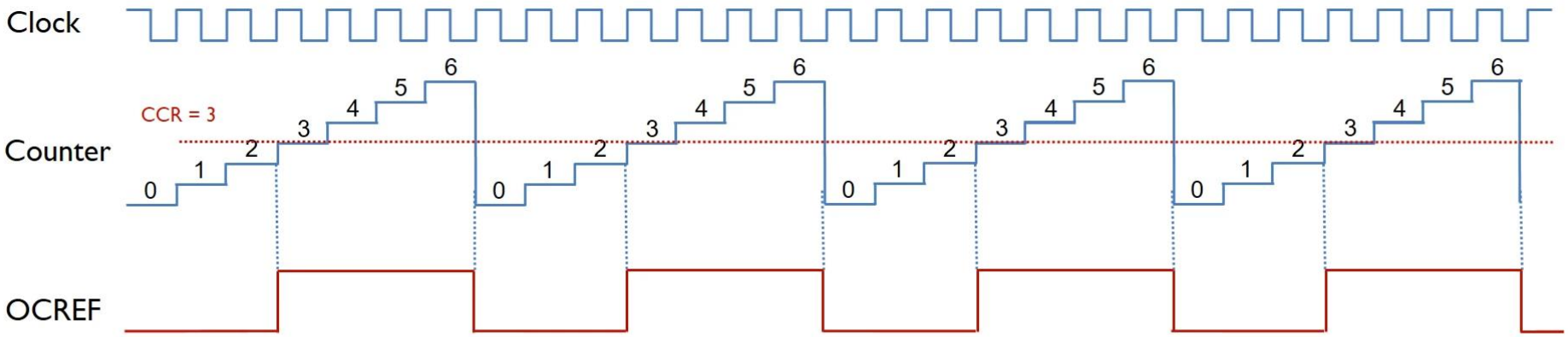
https://www.youtube.com/watch?v=zkrVHIcLGww

PWM Mode 2 Example

Mode 2
Timer Output = { Low if counter < CCR; High if counter ≥ CCR }

PWM Mode 2 (High-True)

Upcounting mode, ARR = 6, CCR = 3, RCR = 0



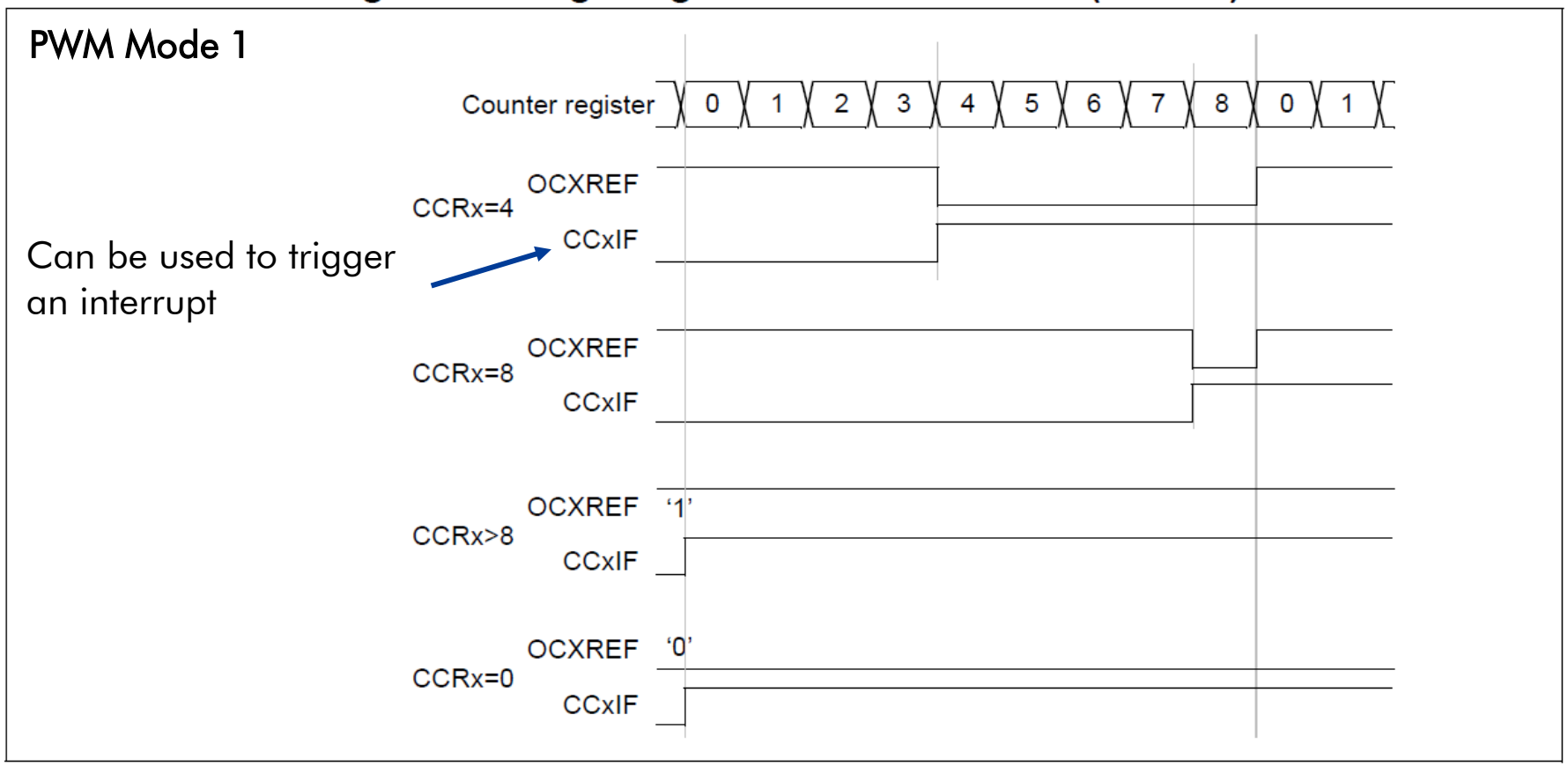
$$\text{Duty Cycle} = 1 - \frac{CCR}{ARR + 1} = \frac{4}{7}$$

$$\text{Period} = (1 + ARR) * \text{Clock Period} = 7 * \text{Clock Period}$$

PWM edge-aligned mode / PWM flank-belynde modus

- The reference PWM signal OCxREF is high as long as TIMx_CNT < TIMx_CCRx else it becomes low. If the compare value in TIMx_CCRx is greater than the auto-reload value (in TIMx_ARR) then OCxREF is held at '1'.

Figure 71. Edge-aligned PWM waveforms (ARR=8)

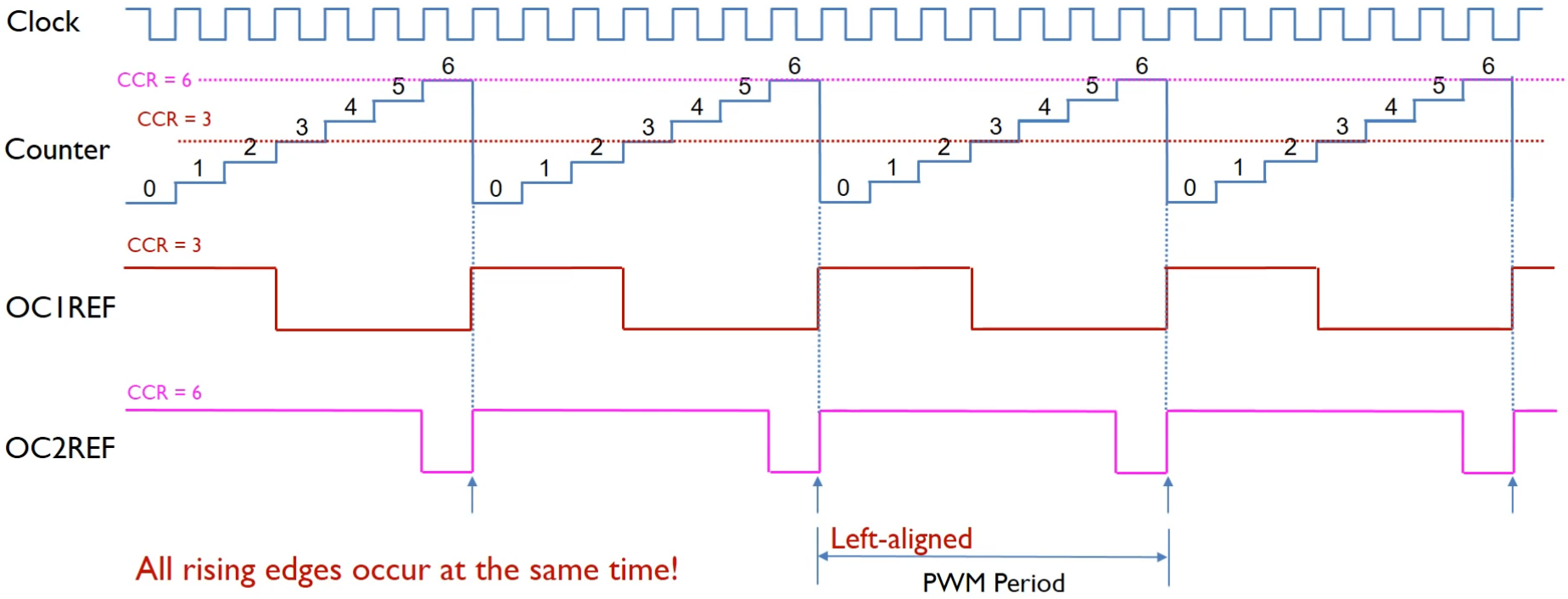


PWM edge-aligned mode / PWM flank-belynde modus

PWM Mode 1

Up-Counting: Edge-aligned

Upcounting mode, ARR = 6, CCR = 3, RCR = 0

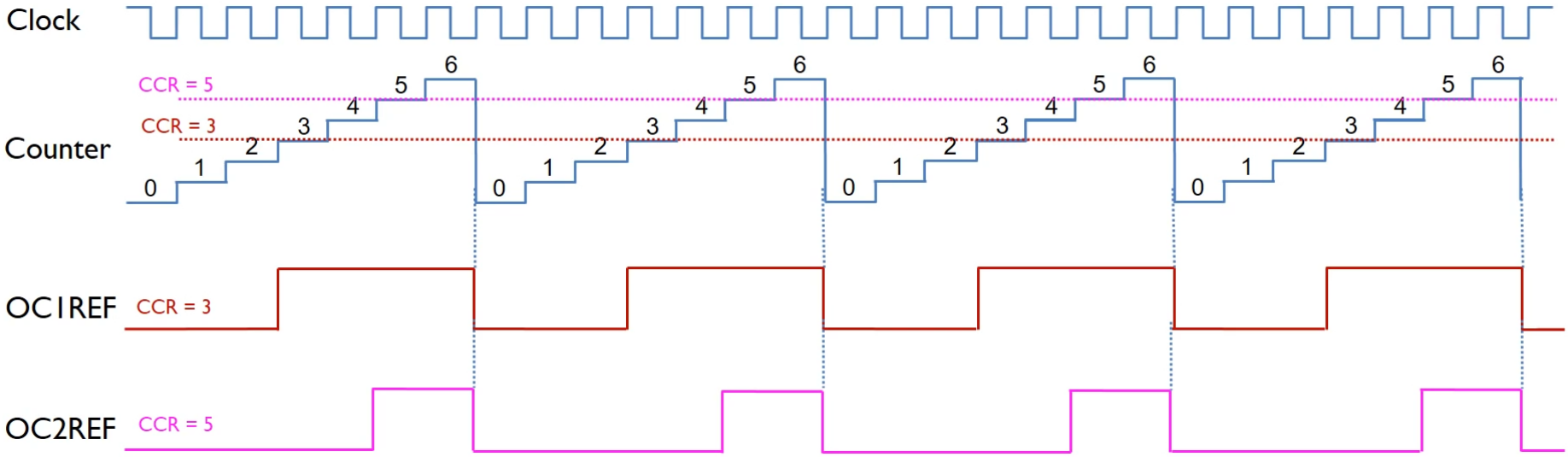


10

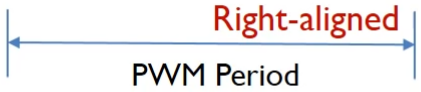
PWM edge-aligned mode / PWM flank-belynde modus

PWM Mode 2: Edge-aligned

Upcounting mode, ARR = 6, CCR = 3, RCR = 0



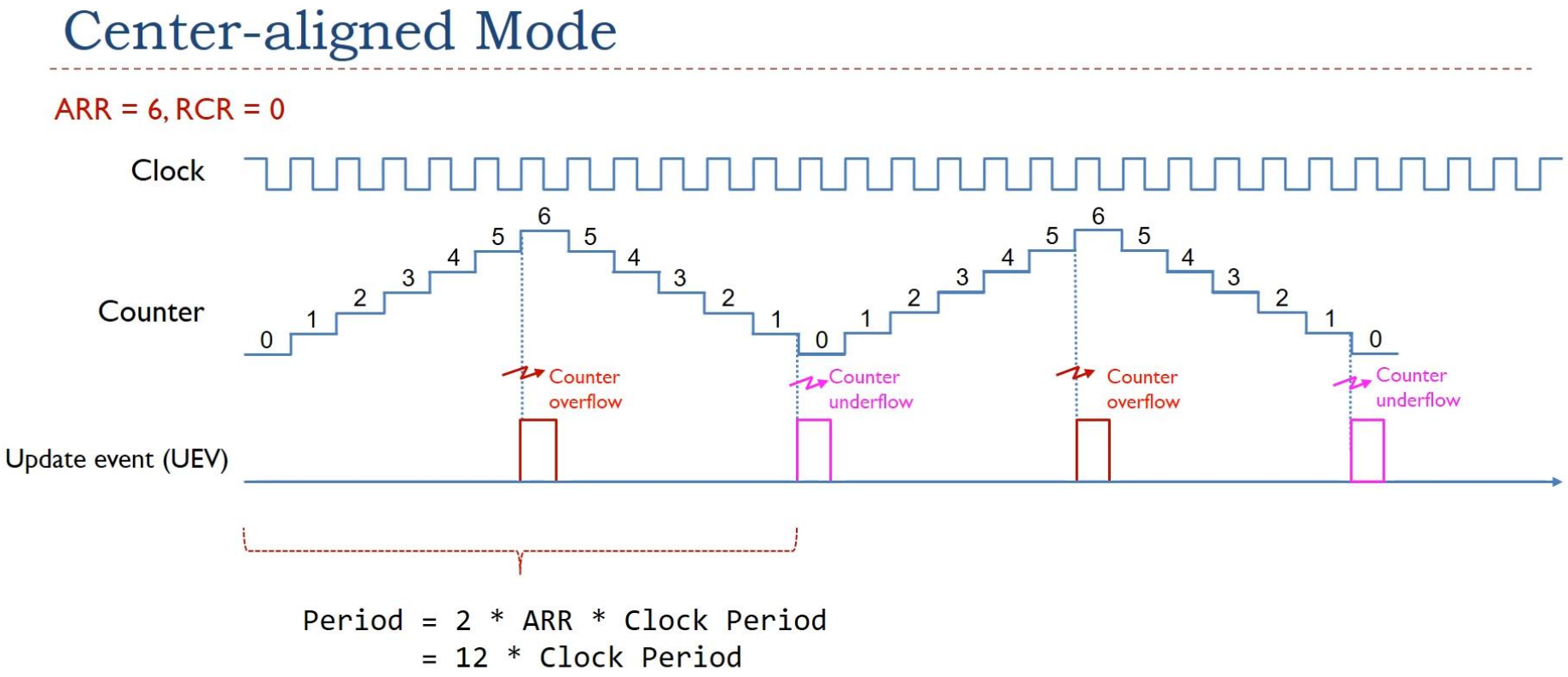
All falling edges occur at the same time!



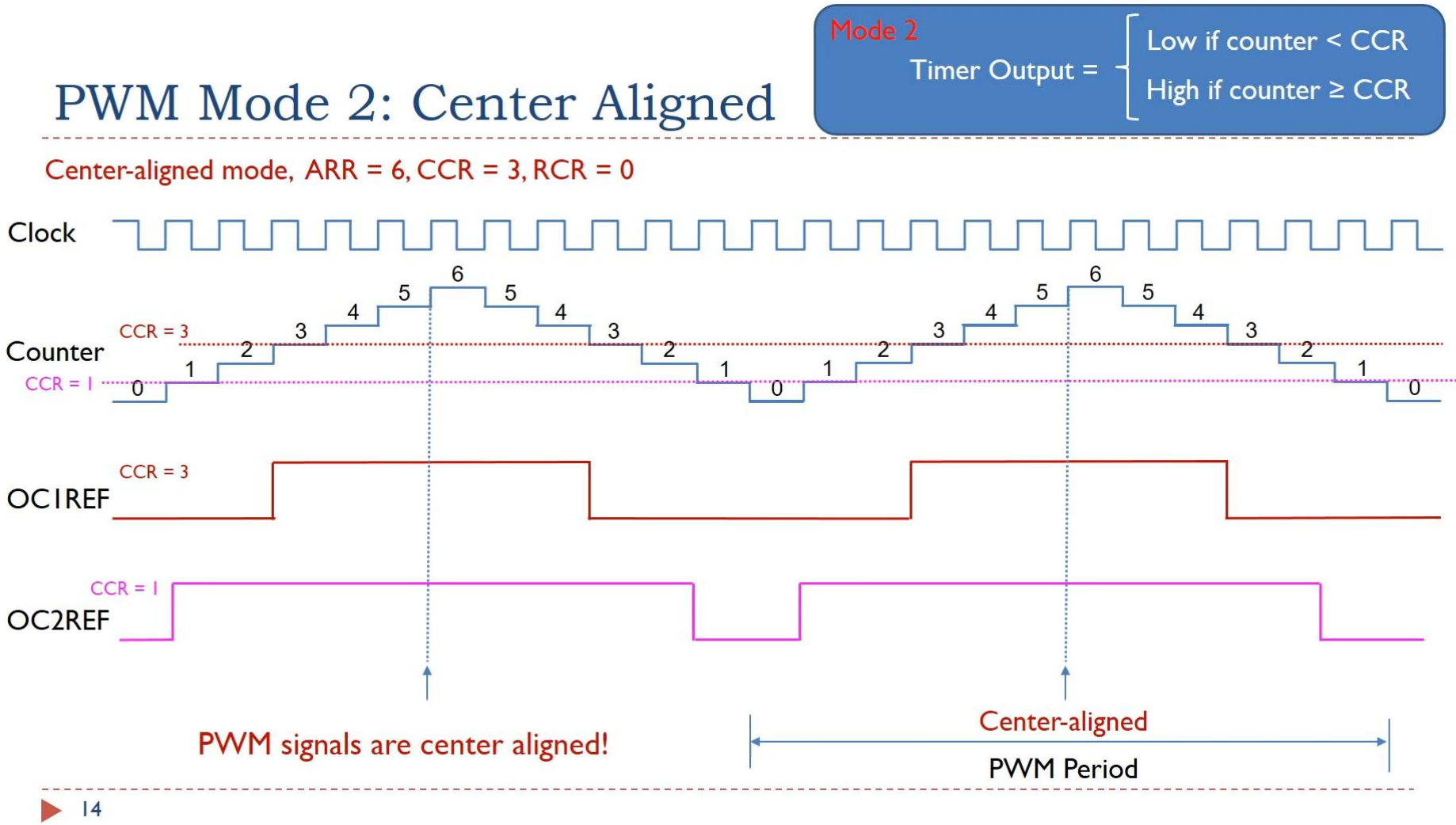
12

PWM center-aligned mode / PWM middel-belynde modus

- Center-aligned mode is active when the CMS bits in TIMx_CR1 register are different from '00' (all the remaining configurations having the same effect on the OCxRef signals).



PWM center-aligned mode / PWM middel-belynde modus



PWM mode

- To start the timer we must include the following line outside our while(1) loop:

```
HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1);
```

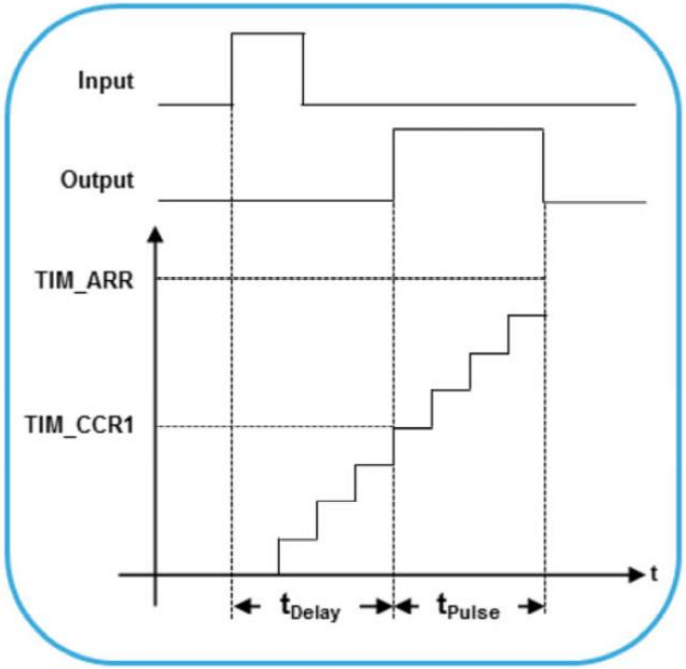
- `htim4` is auto generated by STM32CubeIDE when you set the timer up in the device configuration, matches the Timer set of that output.
- The channel should match the channel that the output pin was set up to use.

- You can also change the duty cycle programmatically by using:

```
htim4.Instance->CCR1 = newDutyCycle;
```

One-pulse mode / Een-puls modus

- One-pulse mode is used to generate a pulse of a programmable length in response to an external event.
- The pulse can start as soon as the input trigger arrives or after a programmable delay. The compare 1 register (CCR1) value defines the pulse start time, while the auto-reload register (ARR) value defines the end of pulse. The effective pulse width is then defined as the difference between the ARR and CCR1 register values.



SysTick Timer

- This timer is dedicated to real-time operating systems, but could also be used as a standard downcounter. It features:
  - A 24-bit downcounter
  - Autoreload capability
  - Maskable system interrupt generation when the counter reaches 0
  - Programmable clock source.

- It is normally used as a 1ms timer that triggers an interrupt that simply increments a variable.
  - This setup is done in HAL_Init();
- HAL_Delay() implements a while loop that waits until the SysTick counter has incremented enough.
  - Must ensure that SysTick has the highest priority when using HAL_Delay() in an interrupt, otherwise system will hang.

  ⇒ **Don't do it!**

```
__weak void HAL_Delay(uint32_t Delay)
{
  uint32_t tickstart = HAL_GetTick();
  uint32_t wait = Delay;

  /* Add a freq to guarantee minimum wait */
  if (wait < HAL_MAX_DELAY)
  {
    wait += (uint32_t)(uwTickFreq);
  }

  while((HAL_GetTick() - tickstart) < wait)
  {
  }
}
```

Watchdog timer

- A watchdog timer (WDT) is a hardware timer that automatically generates a system reset if the main program neglects to periodically service it.
- It is often used to automatically reset an embedded device that hangs because of a software or hardware fault.
- Timer will typically count down from a reload value.
- If the timer reaches zero ⇒ reset the microcontroller
- To prevent timer from reaching zero, write to a register to "kick" the watchdog – causes counter register to start again with reload value.
- If the software stops working, the code that kicks the watchdog will not execute and the microcontroller will reset, restoring operation.

Watchdog timer

The STM32F4 implements two watchdog timers:

- **Independent watchdog (IWDG)**
  - The independent watchdog is based on a 12-bit downcounter and 8-bit prescaler.
  - It is clocked from an independent 32 kHz internal low-speed clock (LSI RC) and as it operates independently from the main clock.
    - It stays active even if main clock fails.
  - it can operate in Stop and Standby modes. It can be used either as a watchdog to reset the device when a problem occurs, or as a free-running timer for application timeout management. It is hardware- or software-configurable through the option bytes.
- **Window watchdog (WWDG)**
  - The window watchdog is based on a 7-bit downcounter that can be set as free-running.
  - It is prescaled from the APB1 clock which comes from the main clock.
  - Used to detect the occurrence of a software fault, usually generated by external interference or by unforeseen logical conditions, which causes the application program to abandon its normal sequence.
  - It can be used as a watchdog to reset the device when a problem occurs.
    - This watchdog can also simply trigger an interrupt instead of resetting the system.

Independent watchdog (IWDG)

- When it reaches the end of count value (0x000) a reset signal is generated (IWDG reset).
- Whenever the key value 0xAAAA is written in the IWDG_KR register, the IWDG_RLR value is reloaded in the counter and the watchdog reset is prevented.

Figure 156. Independent watchdog block diagram

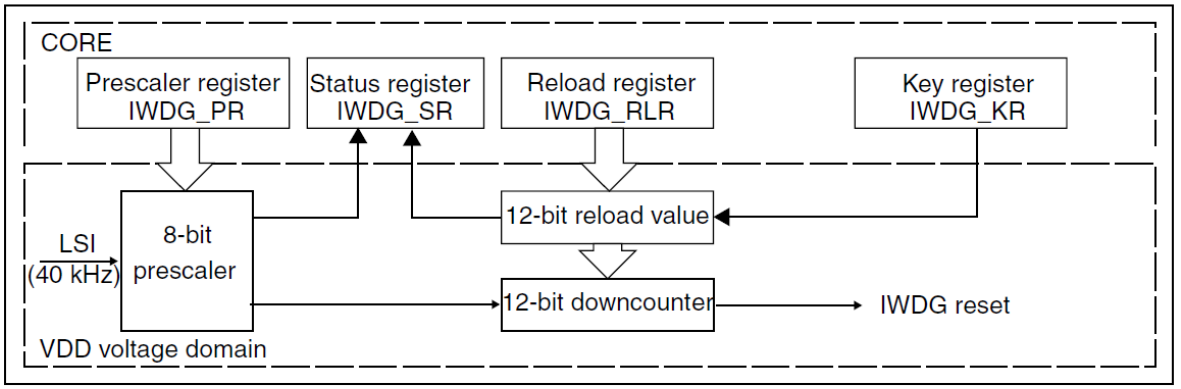


Table 61. Min/max IWDG timeout period at 32 kHz (LSI)(1)

Prescaler divider	PR[2:0] bits	Min timeout (ms) RL[11:0]= 0x000	Max timeout (ms) RL[11:0]= 0xFFF
/4	0	0.125	512
/8	1	0.25	1024
/16	2	0.5	2048
/32	3	1	4096
/64	4	2	8192
/128	5	4	16384
/256	6		32768

1. These timings are given for a 32 kHz clock but the microcontroller's internal RC frequency can vary from 30 to 60 kHz. Moreover, given an exact RC oscillator frequency, the exact timings still depend on the phasing of the APB interface clock versus the LSI clock so that there is always a full RC period of uncertainty.

Real-time clock (RTC) - revise

- The **real-time clock (RTC)** is an independent BCD timer/counter.
- Dedicated registers contain the second, minute, hour (in 12/24 hour), week day, date, month, year, in BCD (binarycoded decimal) format.
  - Correction for 28, 29 (leap year), 30, and 31 day of the month are performed automatically.
- The RTC provides a programmable alarm and programmable periodic interrupts with wakeup from Stop and Standby modes.
- It is clocked by a 32.768 kHz external crystal, resonator or oscillator, the internal low-power RC oscillator, with a typical frequency of 32 kHz, or the high-speed external clock divided by 128.
  - Internal RC oscillator is not very accurate, it's suggested to connect an external 32.768 kHz crystal oscillator (LSE) to PC14 and PC15.
- It is by default configured to generate a time base of 1 second from a clock at 32.768 kHz.

Real-time clock (RTC)

- As long as the supply voltage remains in the operating range, the RTC never stops, regardless of the device status (Run mode, low power mode or under reset).
- Digital calibration circuit (periodic counter correction)
  - 5 ppm accuracy
  - 0.95 ppm accuracy, obtained in a calibration window of several seconds

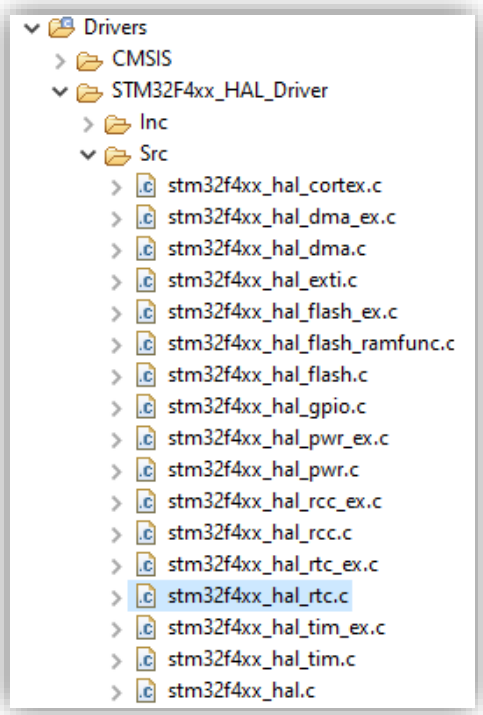
Real-time clock (RTC)

- To configure the RTC Calendar (Time and Date) use the `HAL_RTC_SetTime()` and `HAL_RTC_SetDate()` functions.
- To read the RTC Calendar, use the `HAL_RTC_GetTime()` and `HAL_RTC_GetDate()` functions.
- For example:
  - `HAL_RTC_GetTime(&hrtc, &sTime, RTC_FORMAT_BIN);`
  - `HAL_RTC_GetDate(&hrtc, &sDate, RTC_FORMAT_BIN);`

  - Where the result will be stored in `sTime` and `sDate`, which must be defined as follows:
  - `RTC_TimeTypeDef sTime = {0};`
  - `RTC_DateTypeDef sDate = {0};`

Real-time clock (RTC)

- We also need an interrupt callback function for when the alarm goes off.
- A template can be found in stm32f4xx_hal_rtc.c
  - `HAL_RTC_AlarmAEventCallback()` copy and paste your own definition in main.c (without __weak)



- Example: Turn an LED on when the alarm goes off

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(hrtc);
  /* NOTE : This function should not be modified, when the callback is needed,
            the HAL_RTC_AlarmAEventCallback could be implemented in the user file
   */

  // Turn on LED when alarm goes off
  HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, GPIO_PIN_SET);
}
```