BME3321:Introduction to Microcontroller Programming

Topic 6: Basics of Timers in ARM Cortex MCUs

Assist. Prof. Dr. İsmail Cantürk

Mastering STM32 (ch11)

STM32F407 reference manual (ch 17,18,19,20)

Timers

- MCUs can perform time-based operations. (e.g., increase int i at every 10 second).

- For really simple delays a busy loop could carry out the task, but using the CPU core to perform time-related activities is never a smart solution. E.g., HAL_Delay() function.

- For this reason, all microcontrollers provide dedicated hardware peripherals: the timers.

- STM32 microcontrollers may have different number of timers. STM32F4 has 14 independent timers.

- Timers can be grouped as : basic, general purpose and advanced timers.

Timers

- A timer is a free-running counter with a counting frequency. Counting frequency is a fraction of its source clock.

- Source clock is clock frequency of the bus where it is connected.

- The counting speed (frequency) can be reduced using a dedicated prescaler for each timer.

- Usually, a timer counts from zero up to a given value, which cannot be higher than the maximum unsigned value for its resolution (for example, a 16-bit timer overflows when the counter reaches 65535).

- Depending on the timer type, a timer can generate interrupts.

- The timers in an STM32 MCU can be used for different purposes:
  - They can be used as time base generator (which is the feature common to all STM32 timers).
  - They can be used to measure the frequency of an external signal (input capture mode).
  - To generate PWM signals

  ...

Timer categories in STM32F4

Advanced-control timers (TIM1 and TIM8): They may be used for a variety of purposes, including measuring the pulse lengths of input signals (input capture) or generating output waveforms (PWM and some more waveforms).

General-purpose timers (TIM2 to TIM14): They may be used for a variety of purposes, including measuring the pulse lengths of input signals (input capture) or generating output waveforms (output compare, PWM).

Basic timers (TIM6 and TIM7): They may be used as generic timers for time-base generation but they are also specifically used to drive the digital-to-analog converter (DAC).

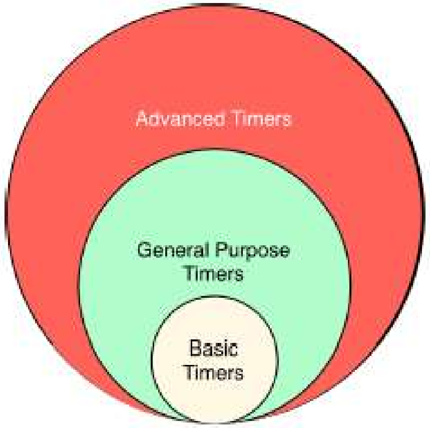


Figure 1: The relation between the three major categories of timers

C struct for Timers

```
typedef struct {
TIM_TypeDef *Instance;                  /* Pointer to timer descriptor */
TIM_Base_InitTypeDef Init;               /* TIM Time Base required parameters */
HAL_TIM_ActiveChannel Channel;          /* Active channel */
DMA_HandleTypeDef *hdma[7];             /* DMA Handlers array */
HAL_LockTypeDef Lock;                   /* Locking object */
__IO HAL_TIM_StateTypeDef State;        /* TIM operation state */
} TIM_HandleTypeDef;
```

C struct for Timers

The most important fields of this struct:

- Instance: is the pointer to the timer that we are going to use. For example, TIM6

- Init: is a nested struct which is used to configure the timer functionalities.

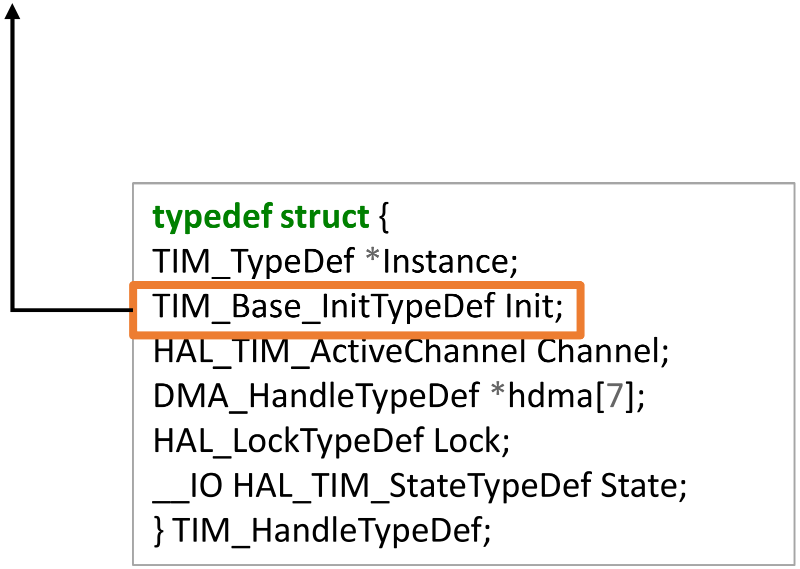
- Channel: it indicates the number of active channels in timers that provide one or more input/output channels (this is not the case of basic timers).

- hdma: this is used for DMA requests.

- State: this is used internally by the HAL to keep track of the timer state.

C struct for Timers

```
typedef struct {
uint32_t Prescaler;            /* Specifies the prescaler value used to divide the TIM clock. */
uint32_t CounterMode;           /* Specifies the counter mode. */
uint32_t Period;               /* Specifies the period value to be loaded into the active
                               Auto-Reload Register at the next update event. */
uint32_t ClockDivision;        /* Specifies the clock division. */
uint32_t RepetitionCounter;    /* Specifies the repetition counter value. */
} TIM_Base_InitTypeDef;
```



```
typedef struct {
TIM_TypeDef *Instance;
TIM_Base_InitTypeDef Init;
HAL_TIM_ActiveChannel Channel;
DMA_HandleTypeDef *hdma[7];
HAL_LockTypeDef Lock;
__IO HAL_TIM_StateTypeDef State;
} TIM_HandleTypeDef;
```

C struct for Timers

- Prescaler: it divides the timer clock by a factor ranging up to 65535 (this means that the prescaler register has a 16-bit resolution). For example, if the bus where the timer is connected runs at 48MHz, then a prescaler value equal to 47 lowers the counting frequency to 1MHz.

Ex1:

Clock frequency of the bus: 48 MHz
Prescaler : 47
Counting frequency of the timer: 48 MHz/(47+1)=1MHz

Ex2:

Clock frequency of the bus: 48 MHz
Prescaler : 47999
Counting frequency of the timer: 48 MHz/(47999+1)=1kHz

Counting of each value will take 1ms.

Counting from 0-4 will take 5ms.

C struct for Timers

- Counter Mode: it defines the counting direction of the timer. The mostly used ones are up and down modes.

  UP:The timer counts from zero up to the Period value (which cannot be higher than the timer resolution, 16/32-bit) and then generates an overflow event.

  DOWN: The timer counts down from the Period value to zero and then generates an underflow event.

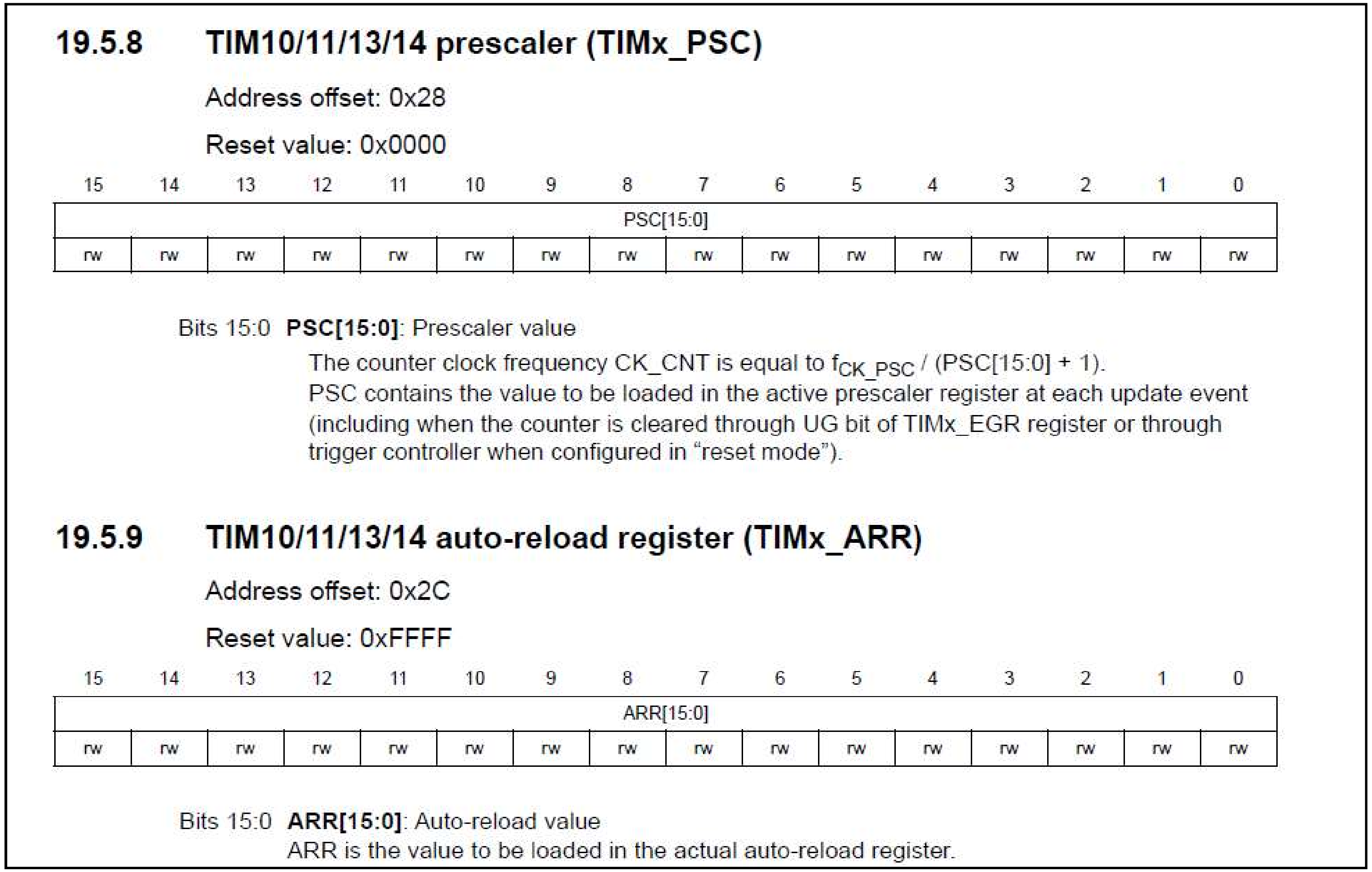
- **Period: sets the maximum value for the timer counter before it restarts counting again. This can assume a value from 0x1 to 0xFFFF (65535) for 16-bit timers, and from 0x1 to 0xFFFF FFFF for 32-bit timers.**
  If Period is set to 0x0 the timer does not start.

- Clock Division: It is used for digital filters. It is available in general purpose and advanced timers.

- Repetition Counter: every timer has a specific update register that keeps track of the timer overflow/underflow condition. The Repetition Counter says how many times the timer overflows/underflows before the update register is set. Repetition Counter is only available in advanced timers.

Prescaler and Autoreload registers

TIMx_PSC: This register defines prescaler value. TIMx_PSC=0x0000
TIMx_ARR: Auto-reload register defines period value. TIMx_ARR=0x0036



19.5.8 TIM10/11/13/14 prescaler (TIMx_PSC)

Address offset: 0x28

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **PSC[15:0]**: Prescaler value
The counter clock frequency CK_CNT is equal to f_{CK_PSC} / (PSC[15:0] + 1).
PSC contains the value to be loaded in the active prescaler register at each update event (including when the counter is cleared through UG bit of TIMx_EGR register or through trigger controller when configured in “reset mode”).

19.5.9 TIM10/11/13/14 auto-reload register (TIMx_ARR)

Address offset: 0x2C

Reset value: 0xFFFF

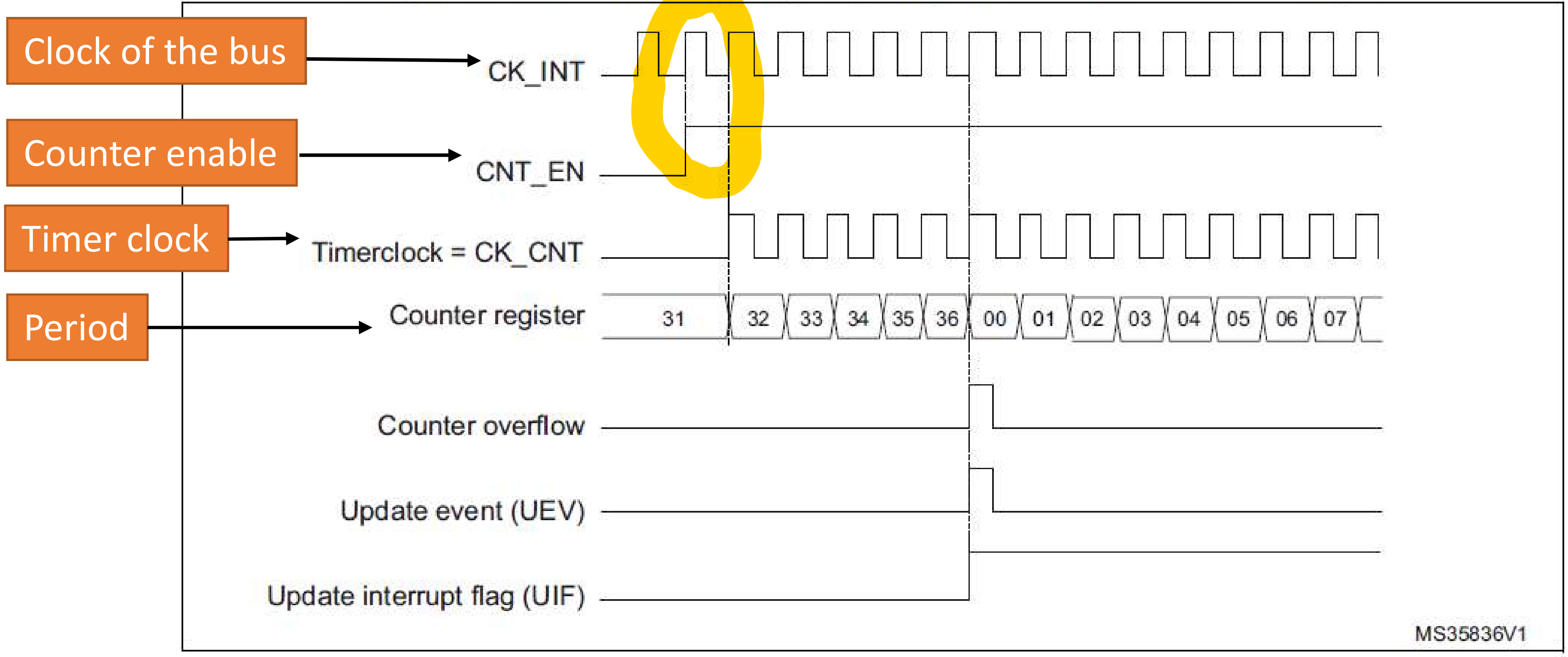
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **ARR[15:0]**: Auto-reload value
ARR is the value to be loaded in the actual auto-reload register.

Counting examples

TIMx_PSC=0x0000
TIMx_ARR=0x0036

Figure 137. Counter timing diagram, internal clock divided by 1



Clock of the bus

Counter enable

Timer clock

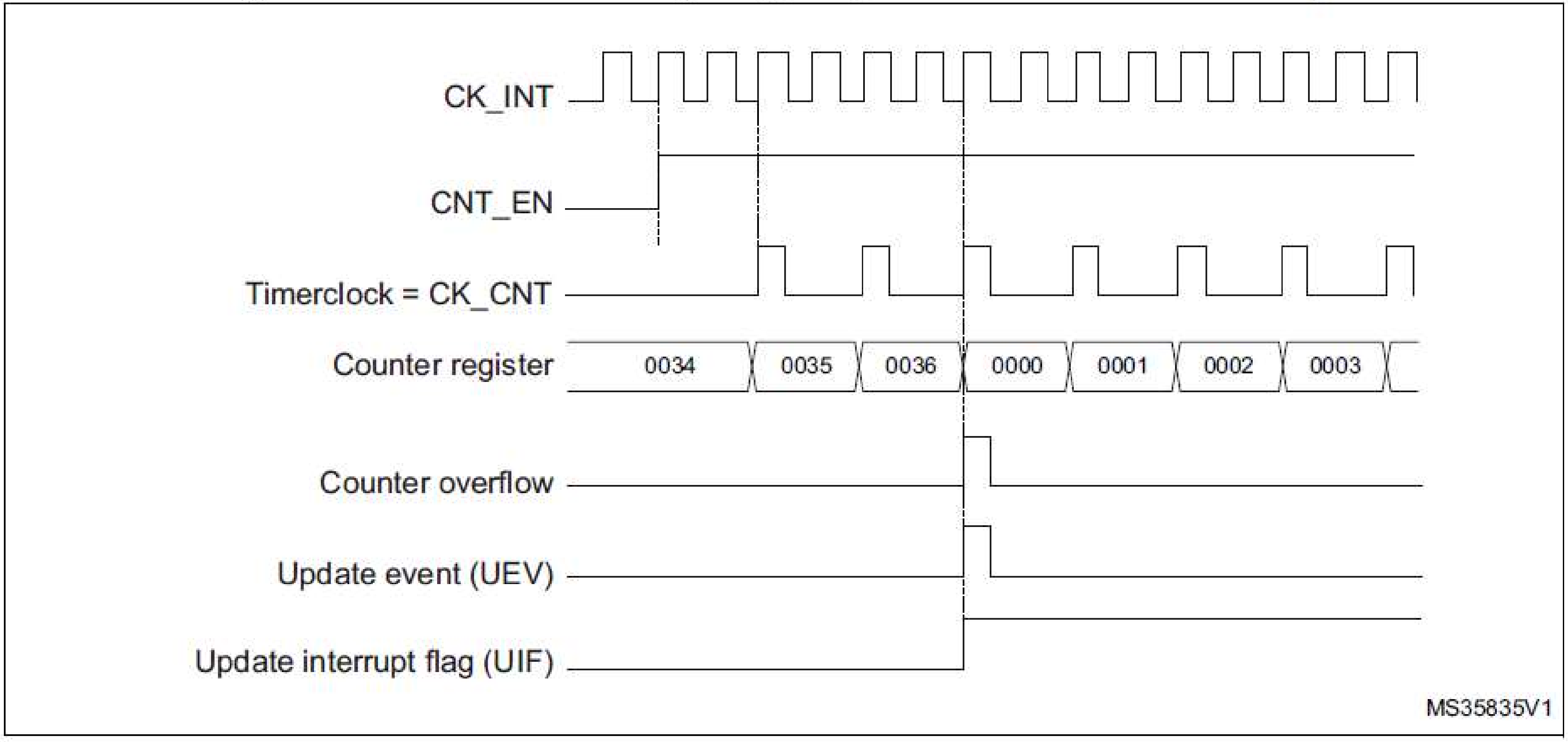
Period

Timer clock is fraction of CK_INT. Internal clock is divided by one for this example.

Counting examples

TIMx_PSC=0x0001
TIMx_ARR=0x0036

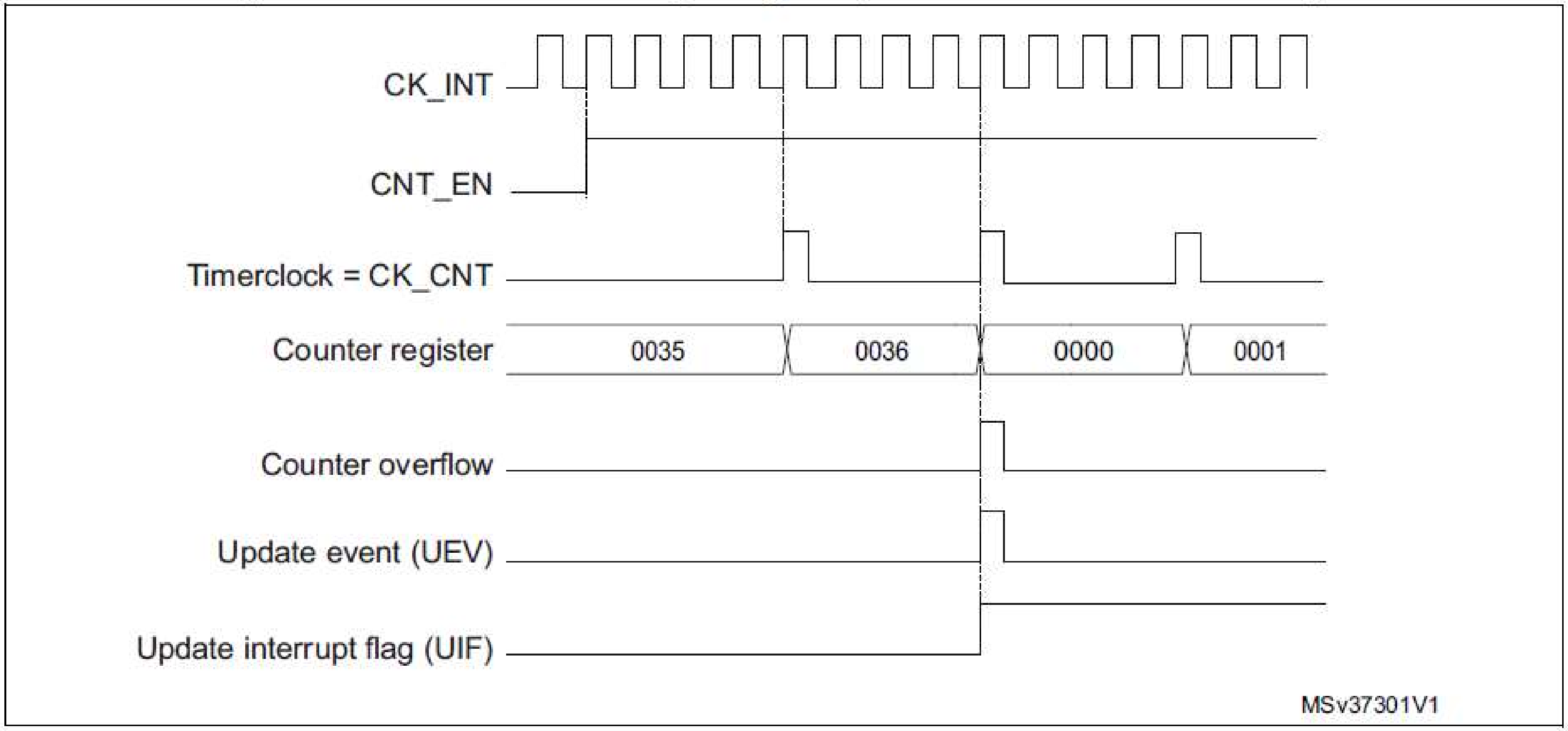
Figure 138. Counter timing diagram, internal clock divided by 2



Counting examples

TIMx_PSC=0x0003
TIMx_ARR=0x0036

Figure 139. Counter timing diagram, internal clock divided by 4



Timers

To summarize what we have seen so far. A basic timer:

- is a free-running counter, which counts from 0 up to the value specified in the Period field, which can assume the maximum value of 0xFFFF (0xFFFF FFFF for 32-bit timers);

- the counting frequency depends on the speed of the bus where the timer is connected, and it can be lowered up to 65536 times by setting the Prescaler register

- when the timer reaches the Period value, it overflows and the Update Event (UEV) flag is set; the timer automatically restarts counting again.

How to choose values for the prescaler and period

Timer 1 is connected to the APB1 bus.

Assume that clock frequency of that bus is 48 MHz.

TIMx_PSC=47999 (decimal)

Clock frequency of Timer 1= (48 MHz/48000)=1kHz. So, T=1ms. T:period of Timer

TIMx_ARR=999 (decimal) (There are 1000 values from 0 to 999)

Required time to reach the period value= 1msx1000=1s

Update event or Interrupt flag will be generated after 1s.

How to choose values for the prescaler and period

$$UpdateEvent = \frac{CK\_INT}{(Prescaler + 1)(Period + 1)}$$

$$= \frac{48MHz}{(47999 + 1)(999 + 1)}$$

$$= 1s$$

$$UpdateEvent = \frac{CK\_INT}{(Prescaler + 1)(Period + 1)(RepetitionCounter + 1)}$$

Repetition Counter is only available in advanced timers.

Repetition Counter: The Repetition Counter says how many times the timer overflows/underflows before the update register is set.

Using timers in interrupt mode

Timers can generate IRQ. When the Period has elapsed, ISR is called.

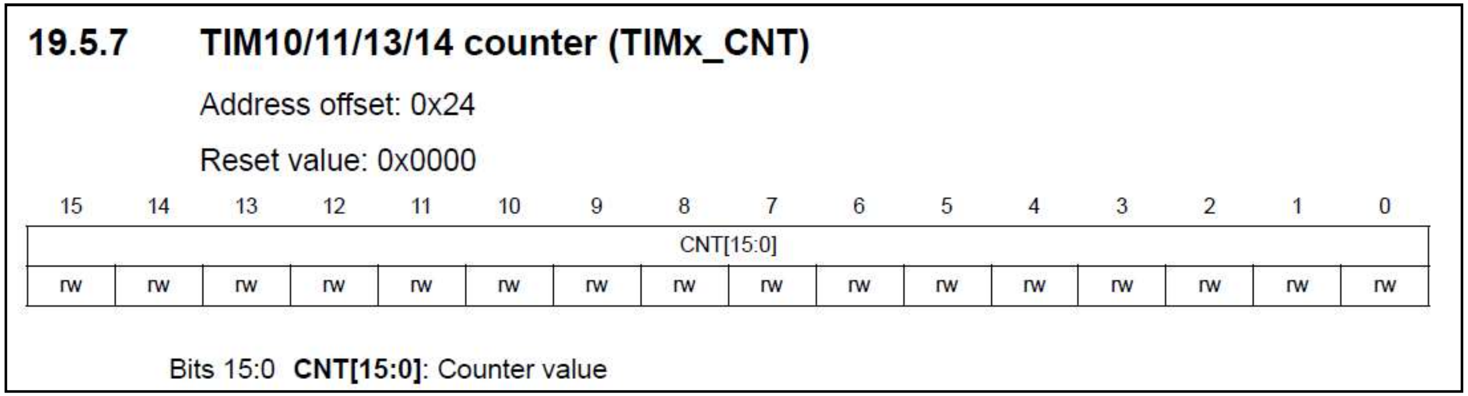
```
.
.
.

HAL_TIM_Base_Start_IT(&htim6); //Start the timer in interrupt mode
.
.
.

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
 {
// This callback is automatically called by the HAL on the UEV event
if(htim->Instance == TIM6)
HAL_GPIO_TogglePin(GPIOD, GPIO_Pin_12);
}
.
.
.
```

Using timers in polling mode

The current value of the timer can be seen by checking the TIMx_CNT register. In pooling mode, the value of this register is controlled.



19.5.7 TIM10/11/13/14 counter (TIMx_CNT)

Address offset: 0x24

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **CNT[15:0]**: Counter value

Using timers in polling mode

The best way for pooling is to check if the timer current counter value is equal or greater than the given value

```
.
.
.

HAL_TIM_Base_Start(&htim6); //Start the timer in pooling mode
.
.
.
while (1) {

value = TIM6_CNT; // or  value =  __HAL_TIM_GET_COUNTER(&htim6);

if (value ≥ desired_value)
{

.
.
.
```

The wrong way of using timers in polling mode

```
HAL_TIM_Base_Start(&htim6); //Start the timer in pooling mode
.
.
.
while (1) {
if(__HAL_TIM_GET_COUNTER(&htim6) == value)
.
.
.
```

The idea behind the polling mode is that the timer counter register (TIMx_CNT) is accessed continuously to check for a given value. But care must be taken when polling a timer.
That way to poll for a timer is completely wrong, even if it apparently works in some examples.

Timers run independently from the Cortex-M core. A timer can count really fast, up to the same clock frequency of the CPU core. But checking a timer counter for equality (that is, to check if it is equal to a given value) requires several ARM assembly instructions, which in turn need several clock cycles. There is no guarantee that the CPU accesses to the counter register exactly at the same time it reaches the configured value (this happens only if the timer runs really slow). A better way is to check if the timer current counter value is equal or greater than the given value

Stopping a timer

The CubeHAL provides functions to stop a running timer:

HAL_TIM_Base_Stop(),
HAL_TIM_Base_Stop_IT()

We pick one of these depending on the timer mode we are using .For example, if we have started a timer in interrupt mode, then we need to stop it using the HAL_TIM_Base_Stop_IT() routine.