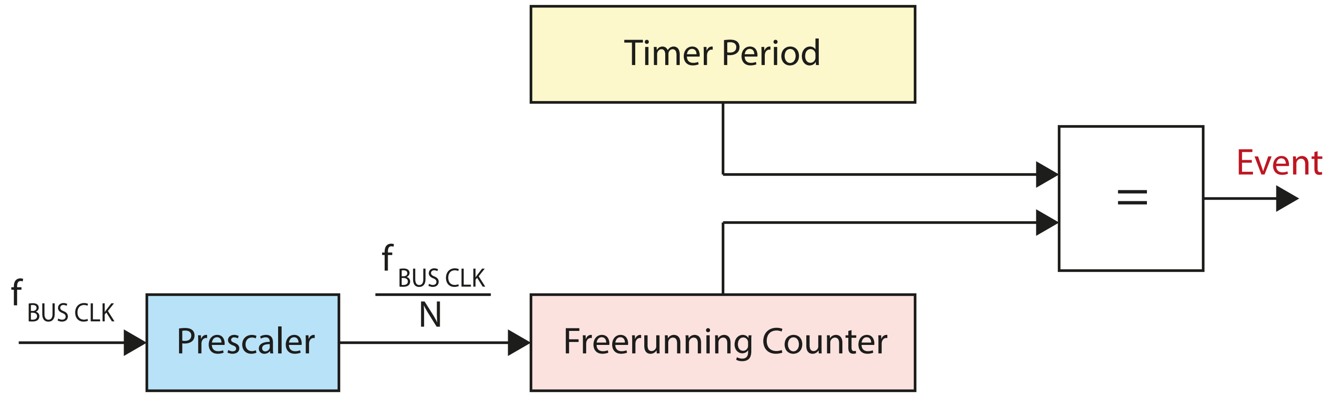
STM32H7 Basic Timers

VGRS 2023

Pa3cio Bulić

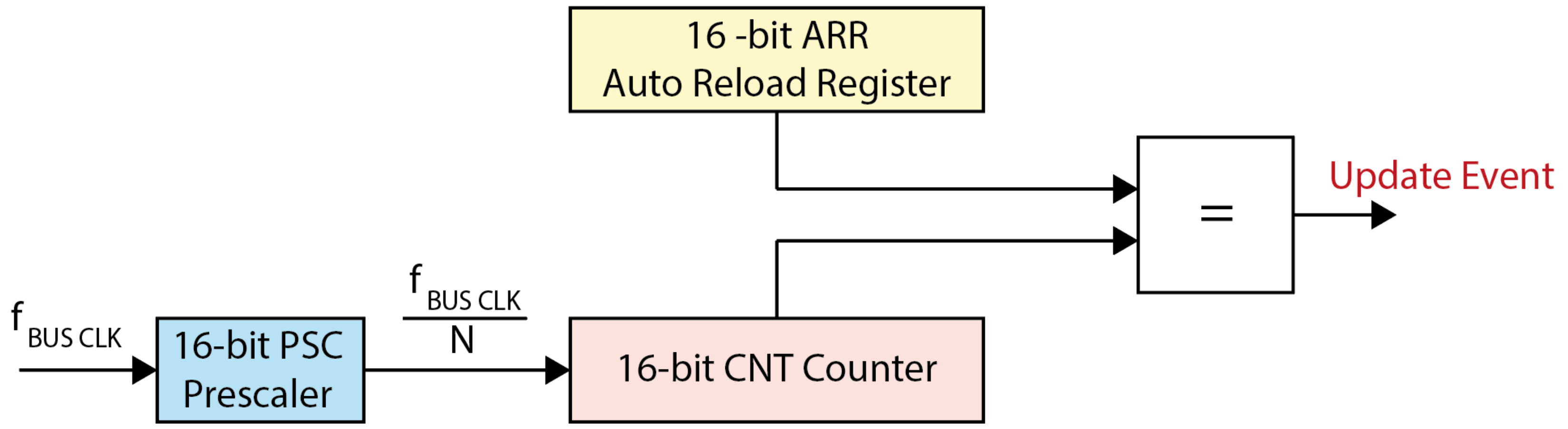
Timers

- A timer is a circuit that enables the system to have the “knowledge of time” - increments (or decrements) on the basis of a programmable clock source. A timer is essential for precise timing and control.
- A timer can generate interrupts
- The hardware of TIMER is composed by three basic ***memory-mapped*** registers:
  - FREERUNNING COUNTER: increments by one for each clock cycle
  - PRESCALER: The prescaler accumulates a prescribed number of bus clocks before issuing a clock tick to the timer.
  - PERIOD: It represents the time it takes for the timer’s counter to reach its maximum value and reset back to zero.



Basic timers in STM32H750 (TIM6, TIM7)

- STM32H7 contains two basic counter: TIM6 and TIM7.

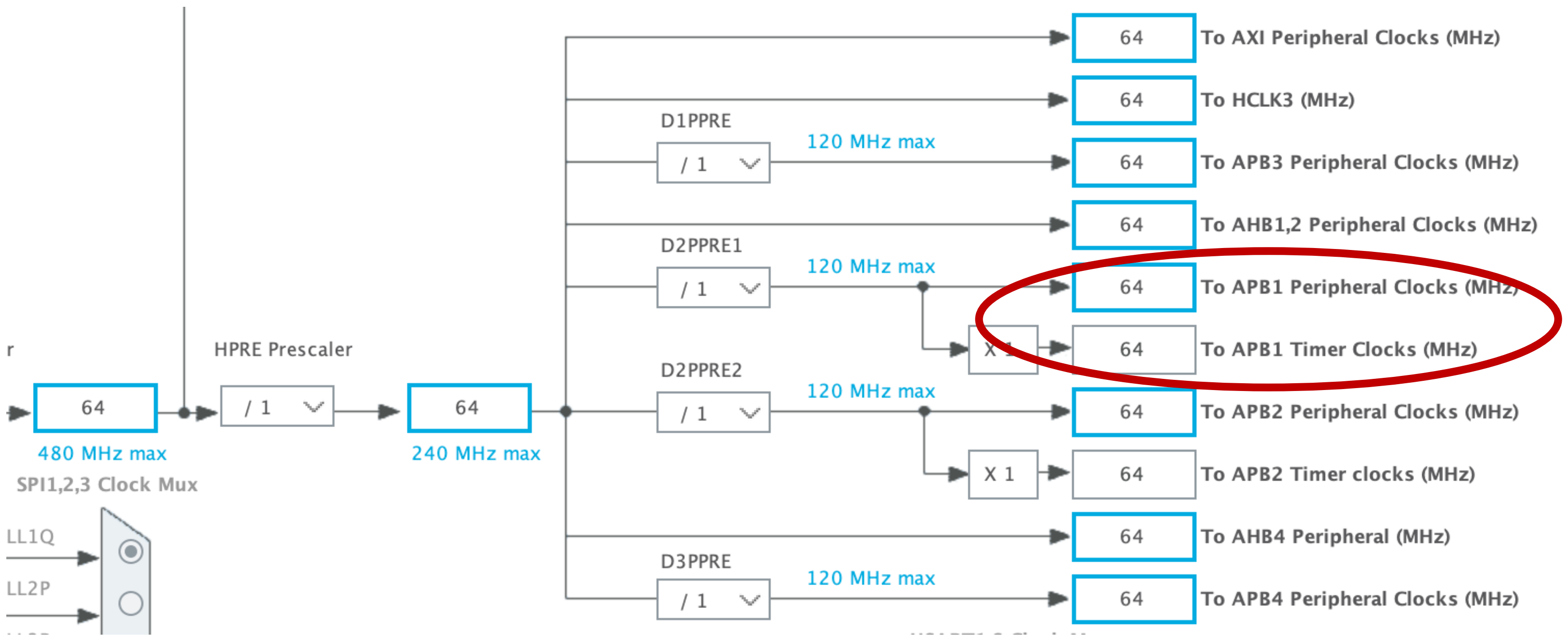


$$f_{UpdateEvent} = \frac{f_{BusClock}}{(PSC+1)(ARR+1)}$$

$$T_{UpdateEvent} = T_{BusClock} \times (PSC+1) \times (ARR+1)$$

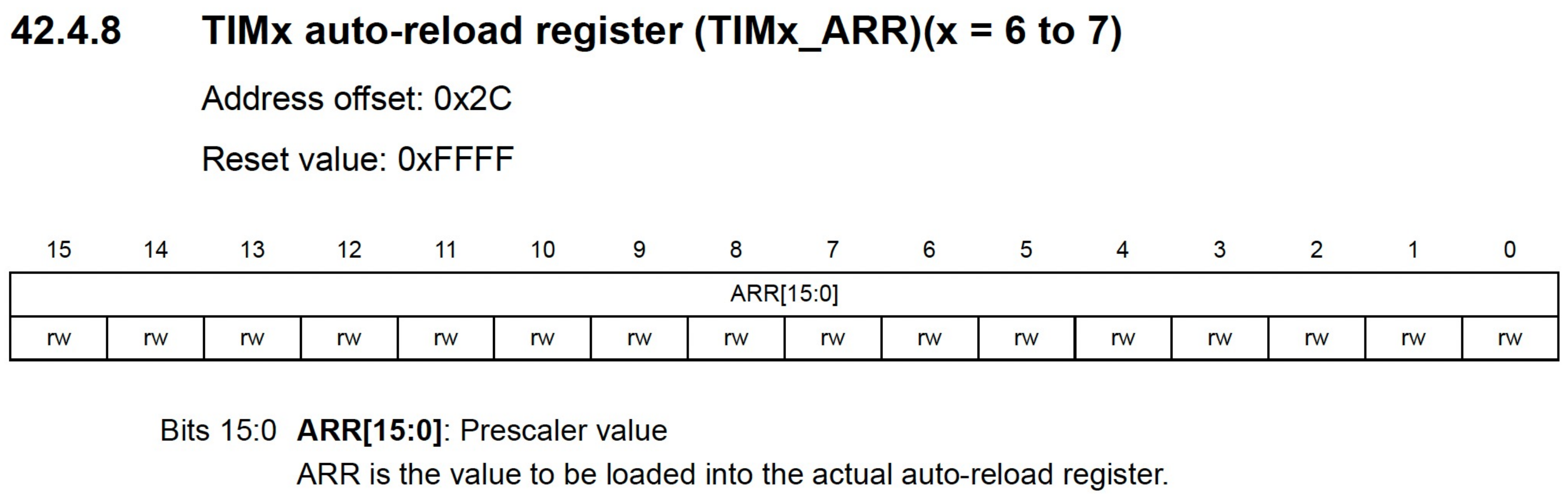
Basic timers in STM32H750 (TIM6, TIM7)

- TIM6 and TIM7 are connected to APB1 bus.



r
64
480 MHz max
SPI1,2,3 Clock Mux
LL1Q
LL2P
HPRE Prescaler
/ 1
64
240 MHz max
D1PPRE
/ 1
120 MHz max
D2PPRE1
/ 1
120 MHz max
X 1
D2PPRE2
/ 1
120 MHz max
X 1
D3PPRE
/ 1
120 MHz max
64 To AXI Peripheral Clocks (MHz)
64 To HCLK3 (MHz)
64 To APB3 Peripheral Clocks (MHz)
64 To AHB1,2 Peripheral Clocks (MHz)
64 To APB1 Peripheral Clocks (MHz)
64 To APB1 Timer Clocks (MHz)
64 To APB2 Peripheral Clocks (MHz)
64 To APB2 Timer clocks (MHz)
64 To AHB4 Peripheral (MHz)
64 To APB4 Peripheral Clocks (MHz)

Timer auto-reload register



42.4.8 TIMx auto-reload register (TIMx_ARR)(x = 6 to 7)

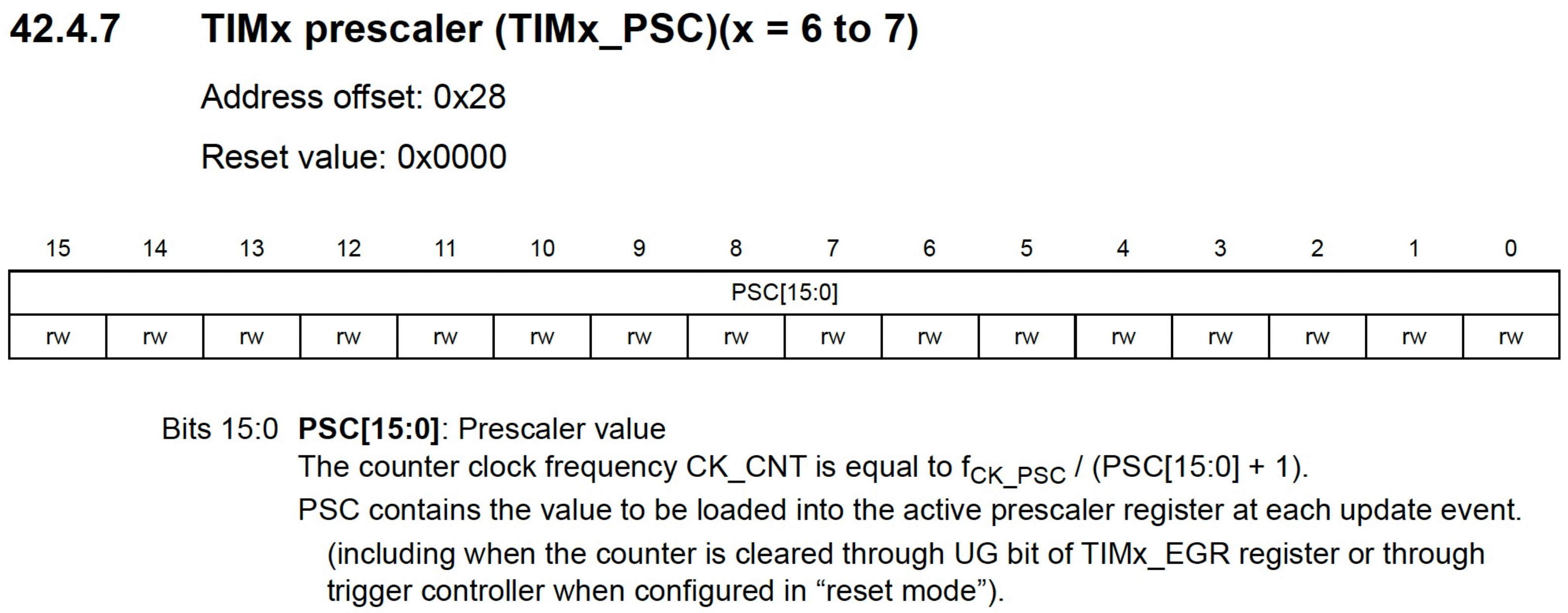
Address offset: 0x2C

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]**: Prescaler value
ARR is the value to be loaded into the actual auto-reload register.

Timer prescaler register



42.4.7 TIMx prescaler (TIMx_PSC)(x = 6 to 7)

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 into 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”).

Timer control register 1

42.4.1 TIMx control register 1 (TIMx_CR1)(x = 6 to 7)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	UIFRE MAP	Res.	Res.	Res.	ARPE	Res.	Res.	Res.	OPM	URS	UDIS	CEN
				rw				rw				rw	rw	rw	rw

Bit 1 **UDIS**: Update disable

This bit is set and cleared by software to enable/disable UEV event generation.

0: UEV enabled. The Update (UEV) event is generated by one of the following events:

- Counter overflow/underflow
- Setting the UG bit
- Update generation through the slave mode controller

Buffered registers are then loaded with their preload values.

1: UEV disabled. The Update event is not generated, shadow registers keep their value (ARR, PSC). However the counter and the prescaler are reinitialized if the UG bit is set or if a hardware reset is received from the slave mode controller.

Bit 0 **CEN**: Counter enable

0: Counter disabled
1: Counter enabled

Note: Gated mode can work only if the CEN bit has been previously set by software. However trigger mode can set the CEN bit automatically by hardware.

CEN is cleared automatically in one-pulse mode, when an update event occurs.

Bit 7 **ARPE**: Auto-reload preload enable

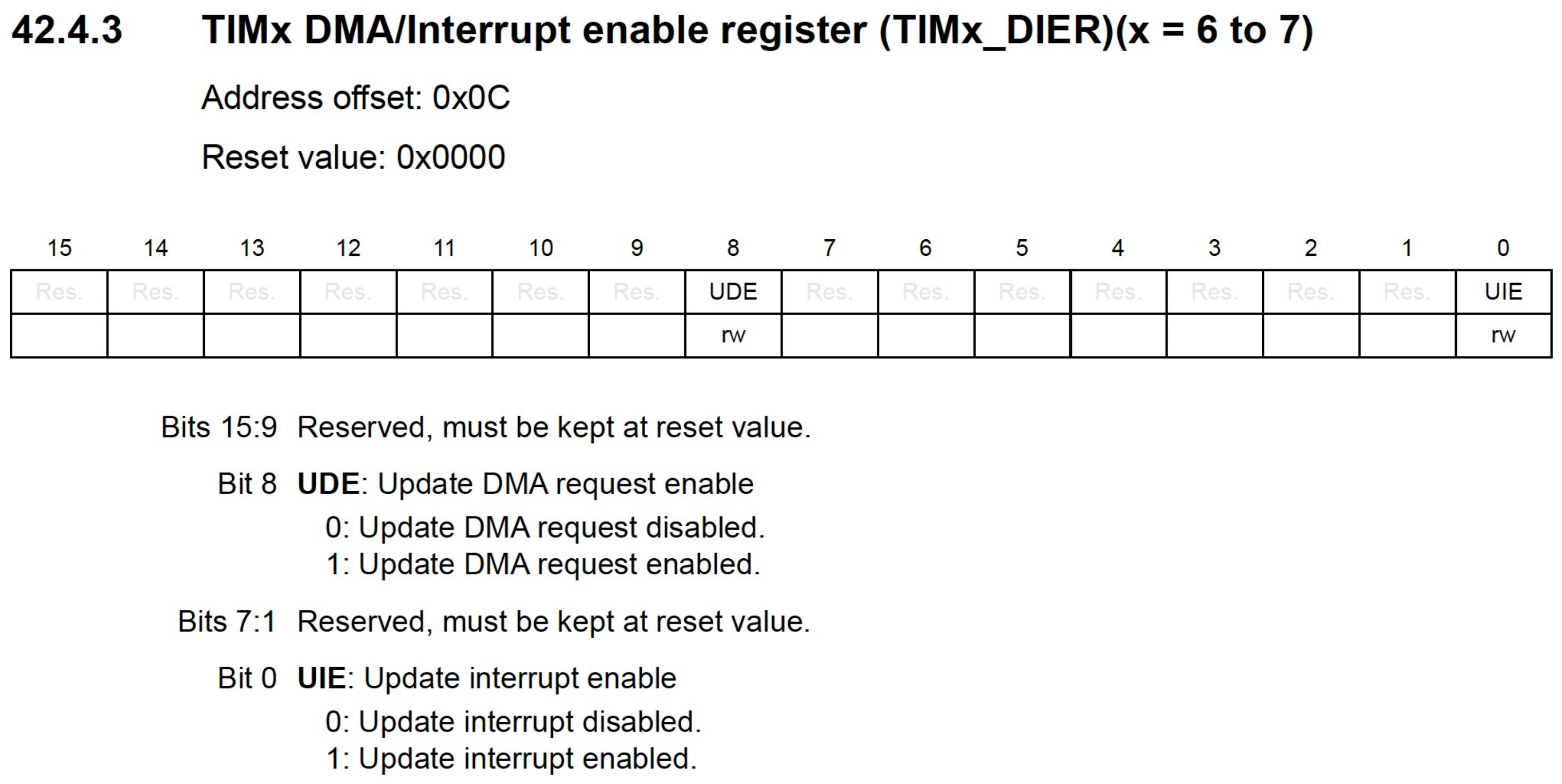
0: TIMx_ARR register is not buffered.
1: TIMx_ARR register is buffered.

Bits 6:4 Reserved, must be kept at reset value.

Bit 3 **OPM**: One-pulse mode

0: Counter is not stopped at update event
1: Counter stops counting at the next update event (clearing the CEN bit).

Timer DMA/Interrupt enable register



42.4.3 TIMx DMA/Interrupt enable register (TIMx_DIER)(x = 6 to 7)

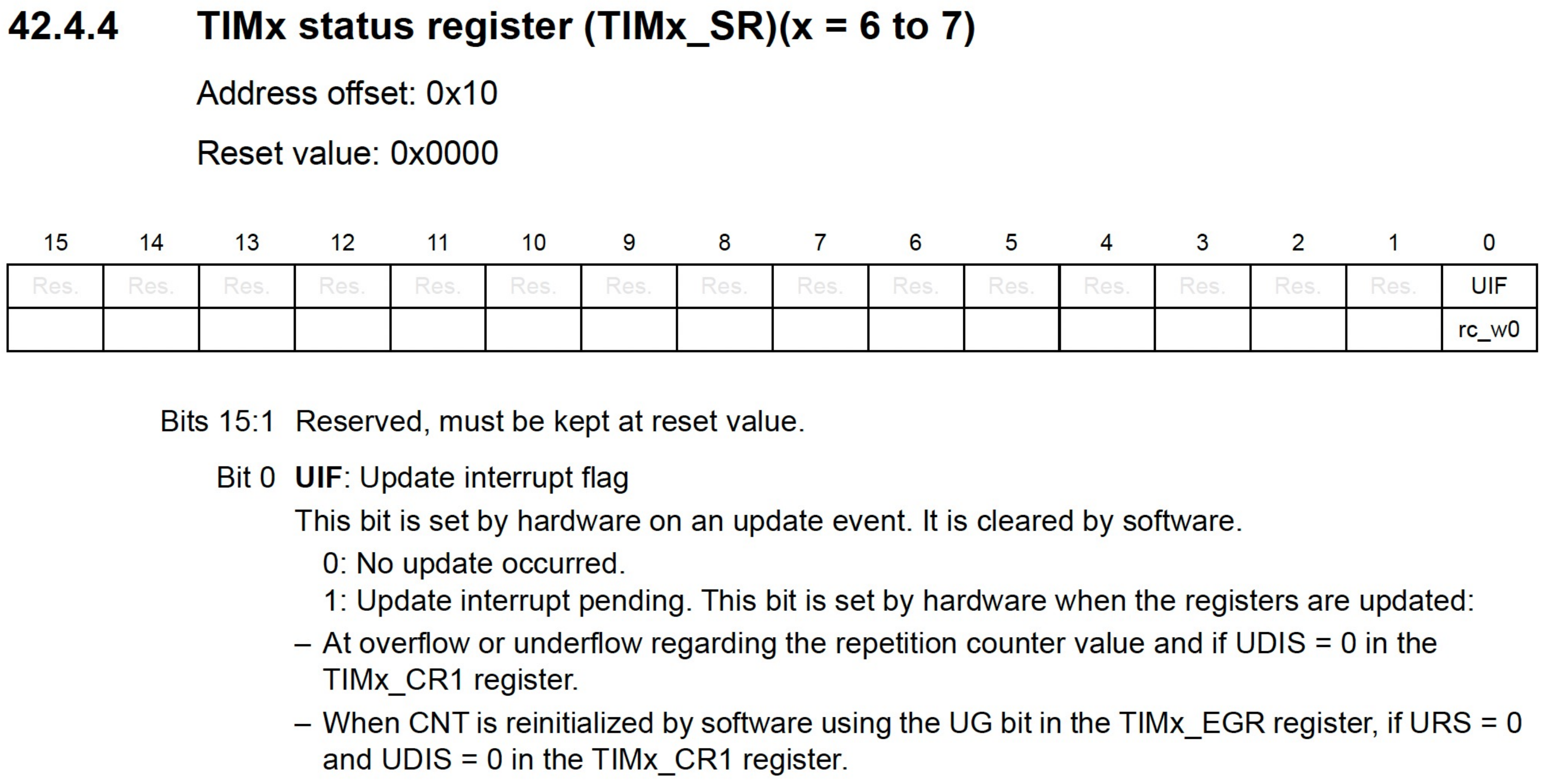
Address offset: 0x0C

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	UDE	Res.	Res.	Res.	Res.	Res.	Res.	Res.	UIE
							rw								rw

Bits 15:9	Reserved, must be kept at reset value.
Bit 8	**UDE**: Update DMA request enable 0: Update DMA request disabled. 1: Update DMA request enabled.
Bits 7:1	Reserved, must be kept at reset value.
Bit 0	**UIE**: Update interrupt enable 0: Update interrupt disabled. 1: Update interrupt enabled.

Timer status register



42.4.4 TIMx status register (TIMx_SR)(x = 6 to 7)

Address offset: 0x10

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	UIF
															rc_w0

Bits 15:1 Reserved, must be kept at reset value.

Bit 0 **UIF**: Update interrupt flag

This bit is set by hardware on an update event. It is cleared by software.

0: No update occurred.

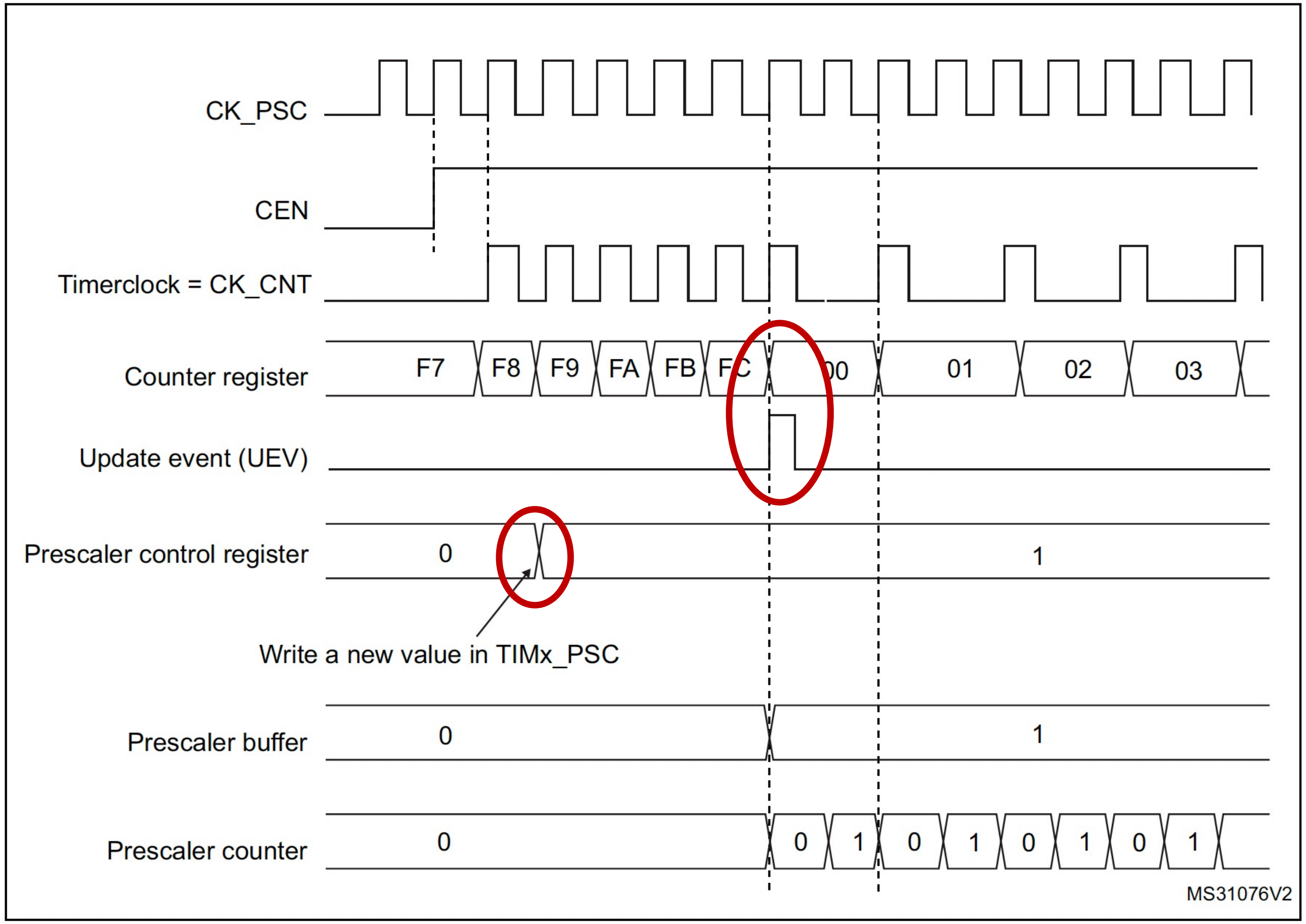
1: Update interrupt pending. This bit is set by hardware when the registers are updated:

– At overflow or underflow regarding the repetition counter value and if UDIS = 0 in the TIMx_CR1 register.

– When CNT is reinitialized by software using the UG bit in the TIMx_EGR register, if URS = 0 and UDIS = 0 in the TIMx_CR1 register.

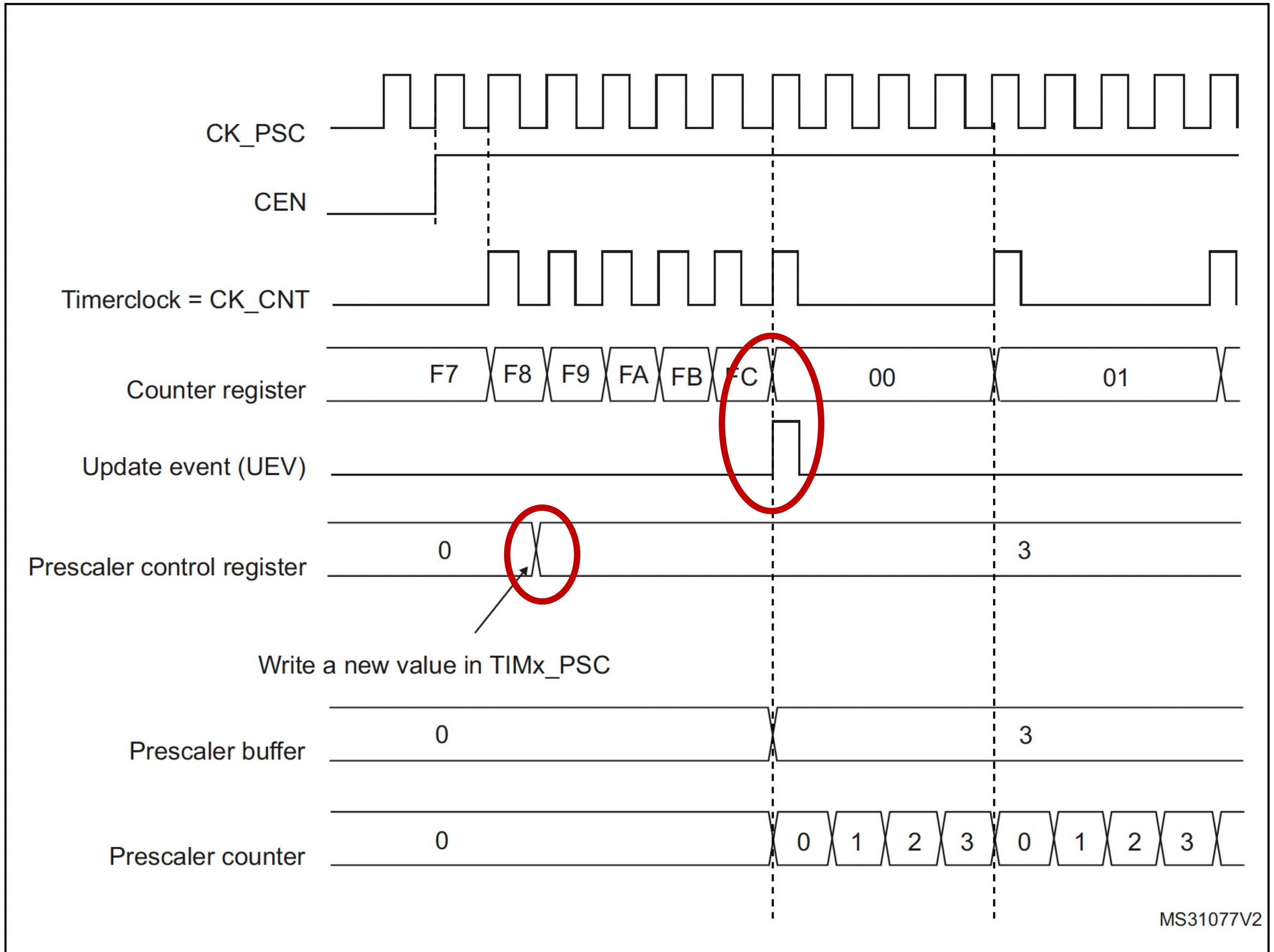
Prescaler operation

Figure 513. Counter timing diagram with prescaler division change from 1 to 2



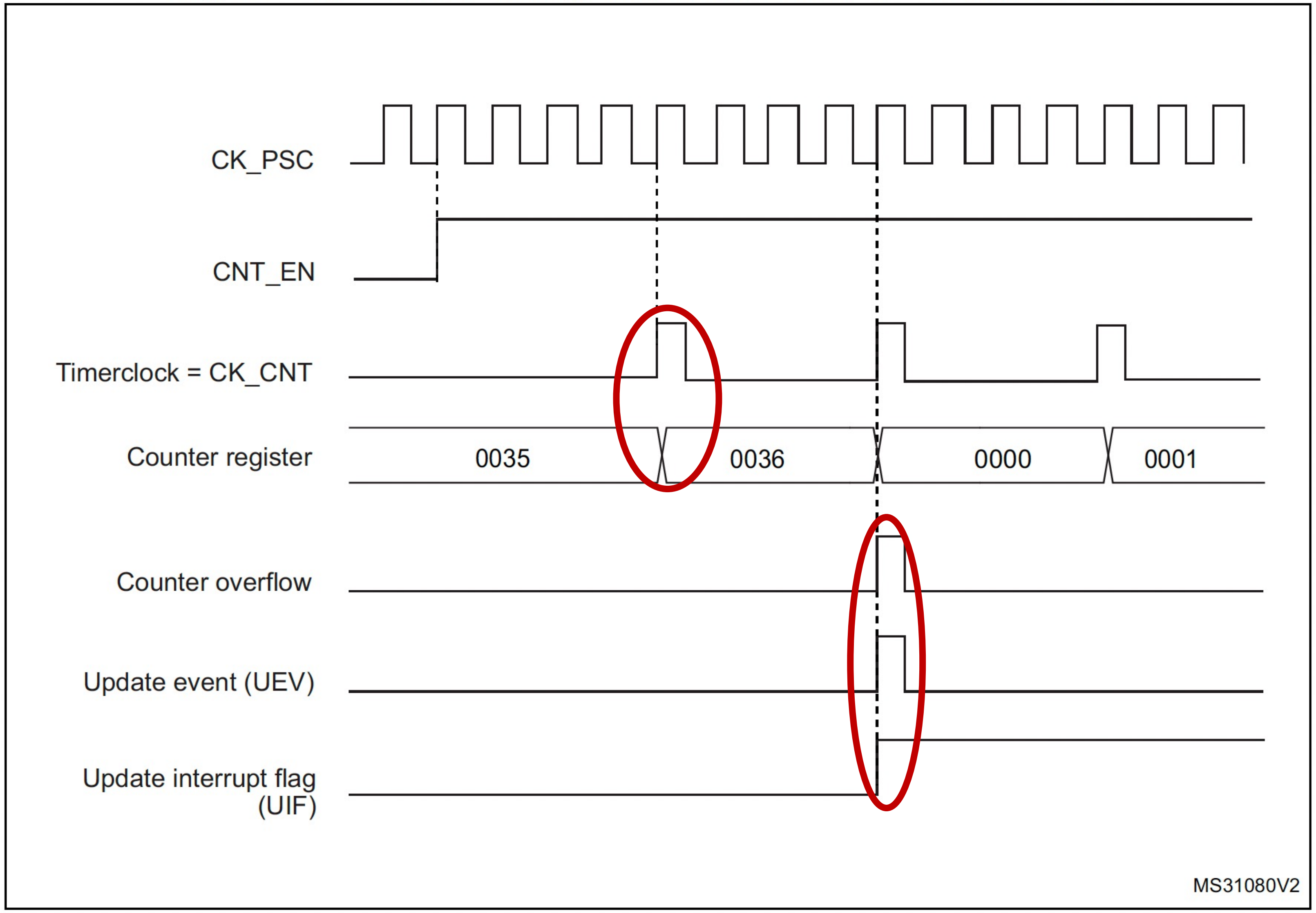
Prescaler operation

Figure 514. Counter timing diagram with prescaler division change from 1 to 4



Timer operation

Figure 517. Counter timing diagram, internal clock divided by 4



Timer operation – ARR preloading (buffering)

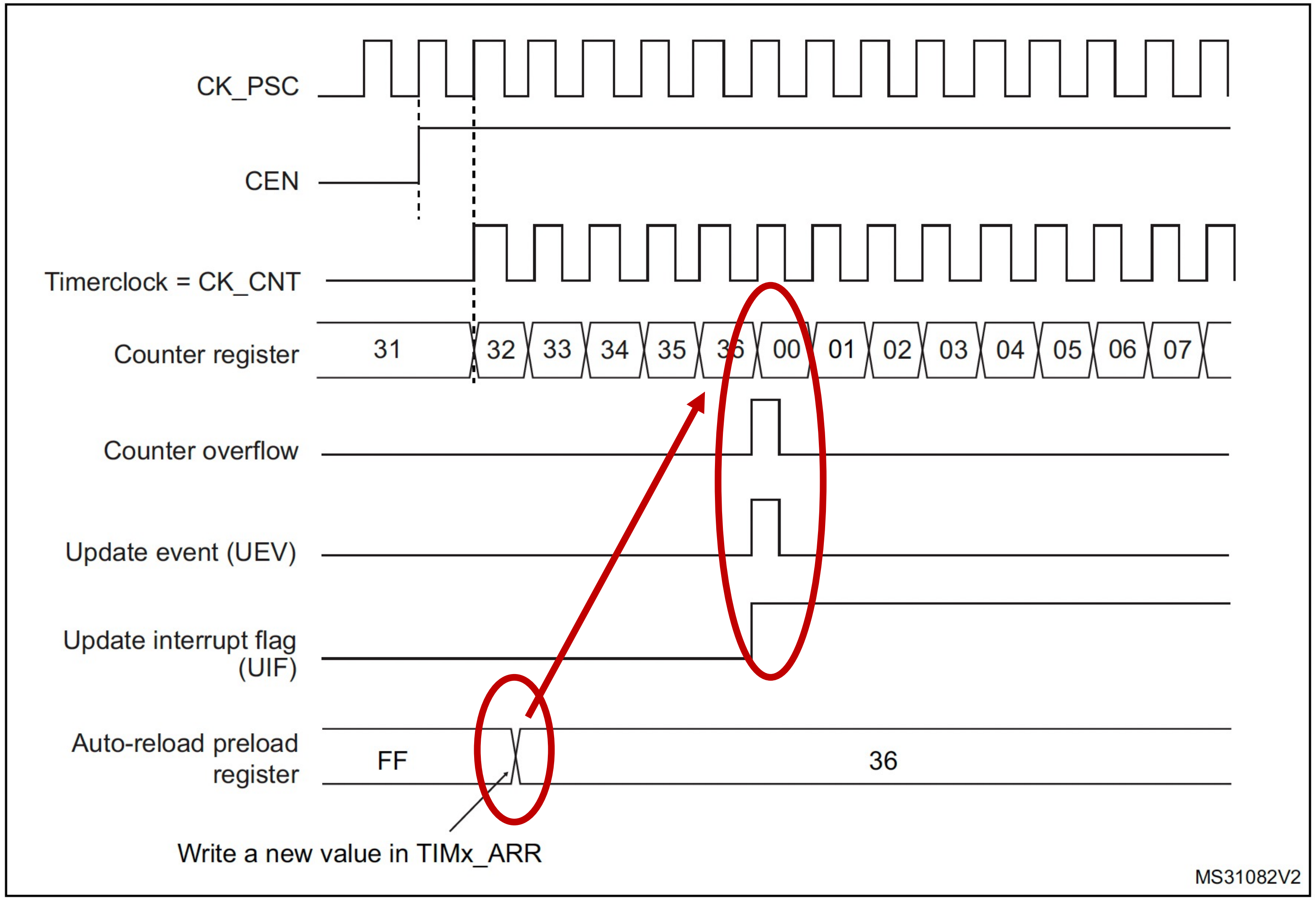
- The auto-reload register (ARR) holds the value at which the counter is automatically reloaded, creating a periodic timer behavior. When the counter reaches the value stored in the auto-reload register, it resets to zero, and the process repeats.

- ***The ARR preloading feature (buffering) allows you to load a new value into the auto-reload register without affecting the ongoing count.*** Instead of immediately taking the new value, it is only applied when an update event occurs.

- Timer ARR register preloading (buffering) provides several advantages:
    - The new value is loaded atomically, preventing glitches or inconsistencies in the timer operation.
    - Preloading allows synchronization of updates with specific timer events, ensuring precise control over the timer's behavior.
    - By updating the registers at a specific moment, preloading can help reduce timing jitter in applications where precise timing is crucial.

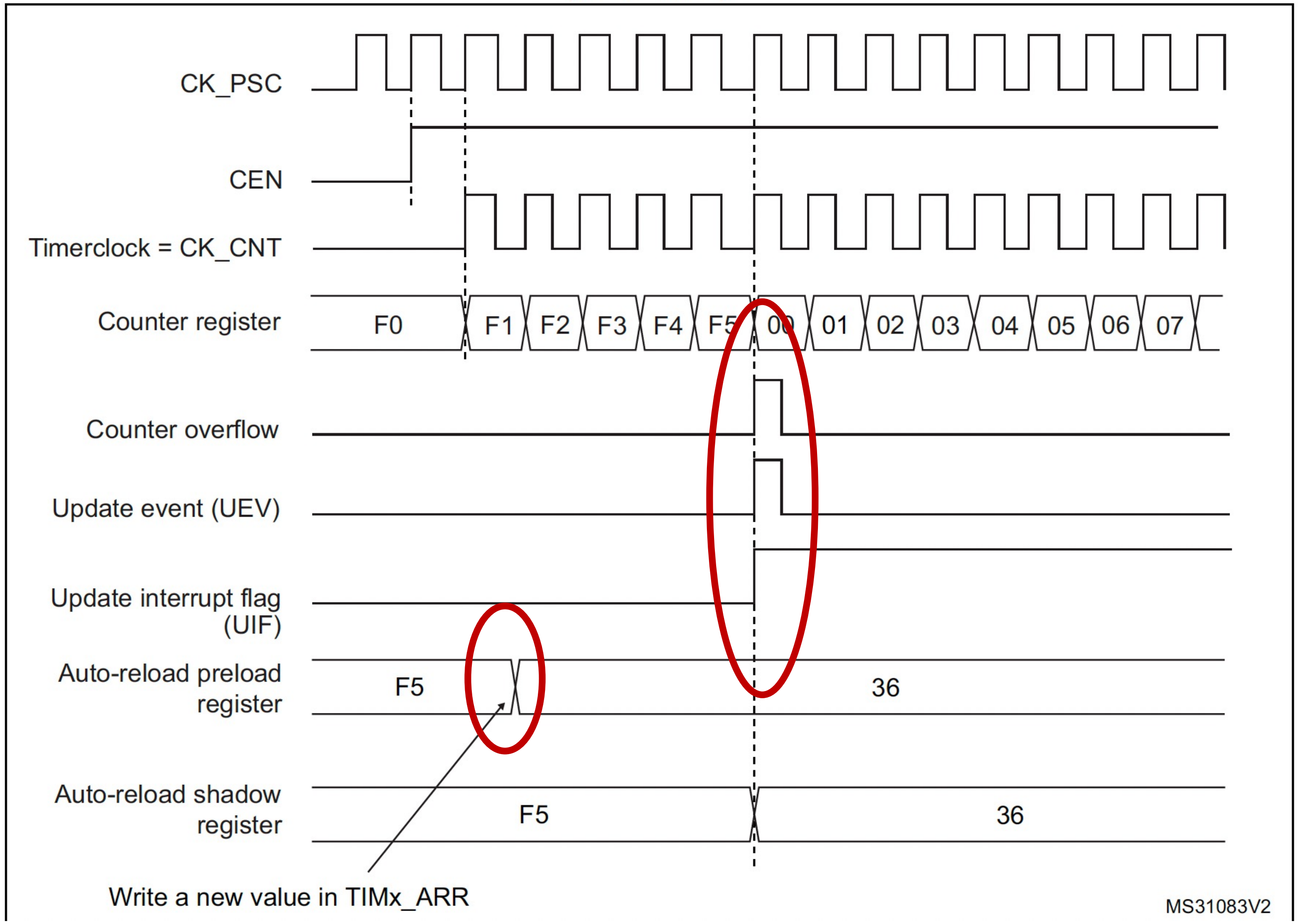
Timer operation – ARR not preloaded

Figure 519. Counter timing diagram, update event when ARPE = 0 (TIMx_ARR not preloaded)

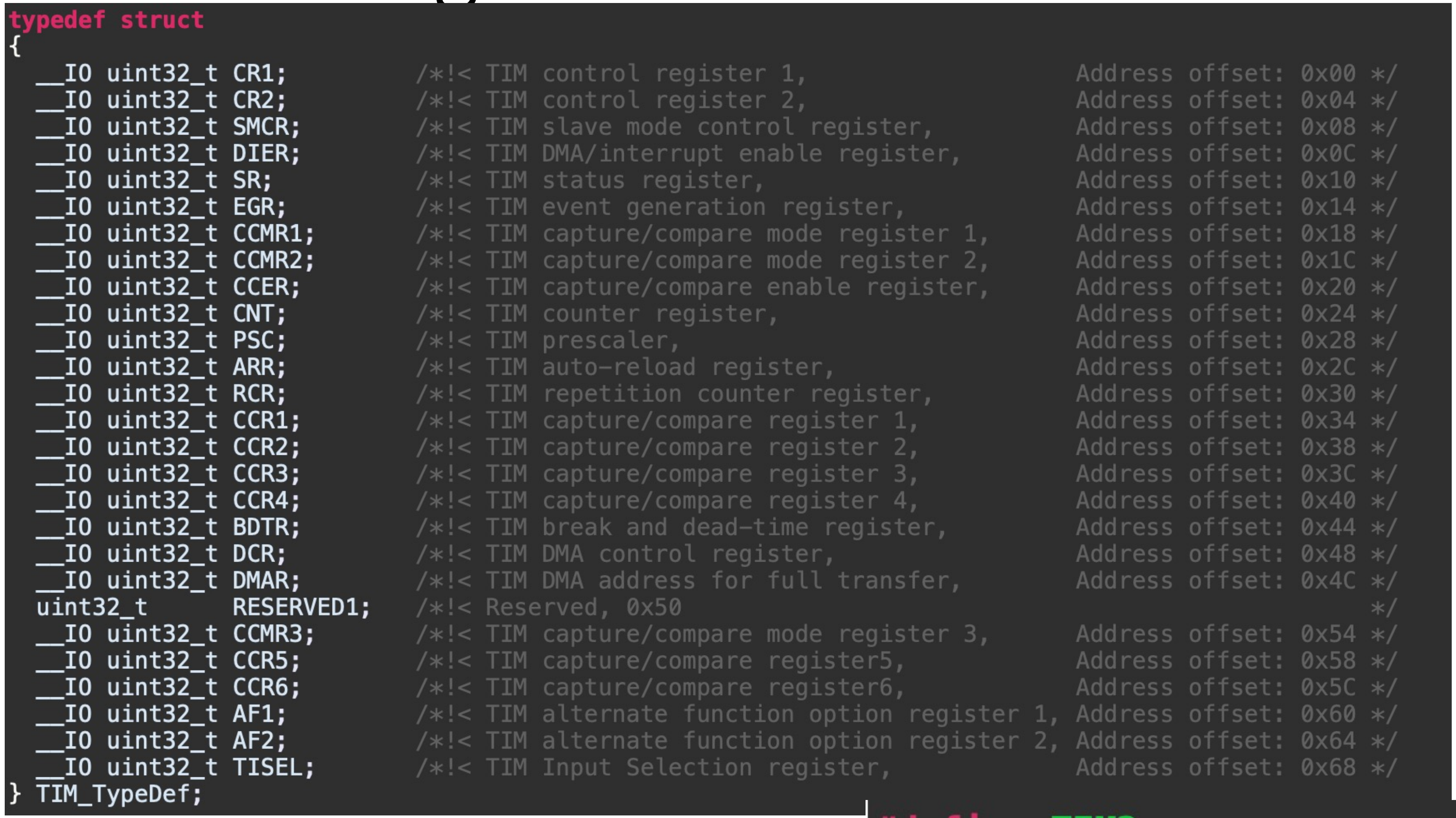


Timer operation – ARR preloaded

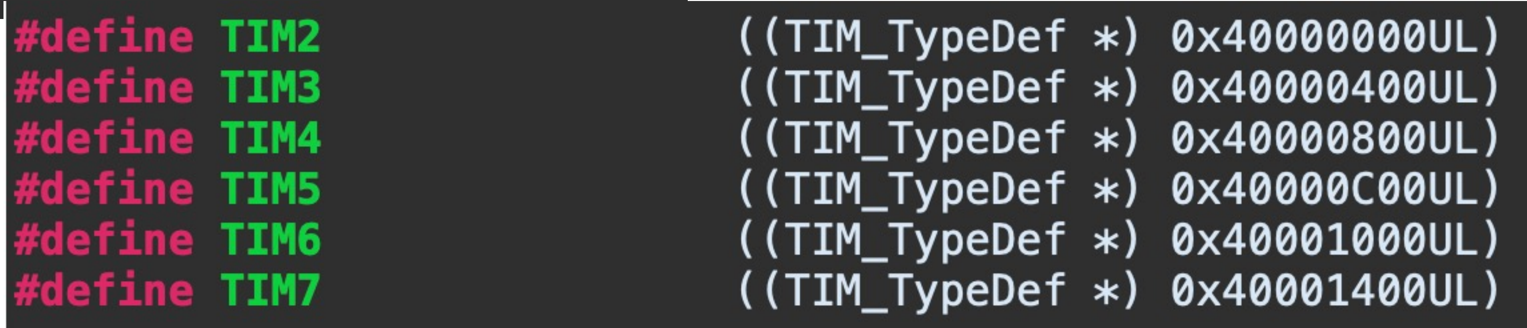
Figure 520. Counter timing diagram, update event when ARPE=1 (TIMx_ARR preloaded)



Timer registers abstraction

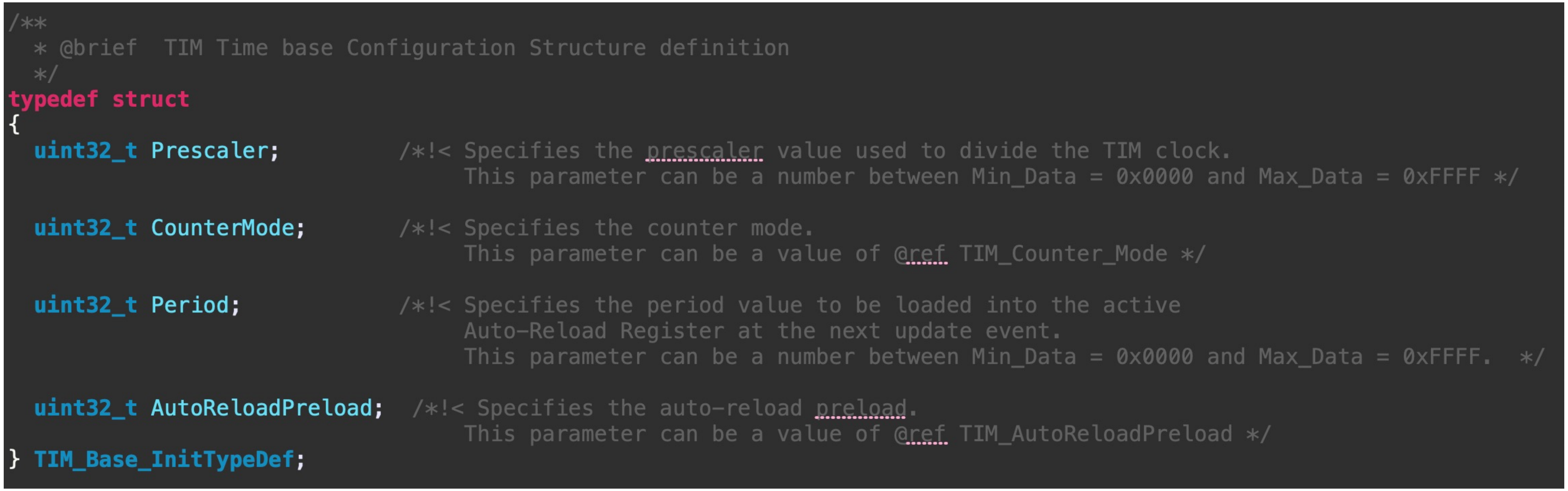


```
typedef struct
{
  __IO uint32_t CR1;         /*!< TIM control register 1,                 Address offset: 0x00 */
  __IO uint32_t CR2;         /*!< TIM control register 2,                 Address offset: 0x04 */
  __IO uint32_t SMCR;        /*!< TIM slave mode control register,        Address offset: 0x08 */
  __IO uint32_t DIER;        /*!< TIM DMA/interrupt enable register,      Address offset: 0x0C */
  __IO uint32_t SR;          /*!< TIM status register,                    Address offset: 0x10 */
  __IO uint32_t EGR;         /*!< TIM event generation register,          Address offset: 0x14 */
  __IO uint32_t CCMR1;       /*!< TIM capture/compare mode register 1,    Address offset: 0x18 */
  __IO uint32_t CCMR2;       /*!< TIM capture/compare mode register 2,    Address offset: 0x1C */
  __IO uint32_t CCER;        /*!< TIM capture/compare enable register,    Address offset: 0x20 */
  __IO uint32_t CNT;         /*!< TIM counter register,                   Address offset: 0x24 */
  __IO uint32_t PSC;         /*!< TIM prescaler,                          Address offset: 0x28 */
  __IO uint32_t ARR;         /*!< TIM auto-reload register,               Address offset: 0x2C */
  __IO uint32_t RCR;         /*!< TIM repetition counter register,        Address offset: 0x30 */
  __IO uint32_t CCR1;        /*!< TIM capture/compare register 1,         Address offset: 0x34 */
  __IO uint32_t CCR2;        /*!< TIM capture/compare register 2,         Address offset: 0x38 */
  __IO uint32_t CCR3;        /*!< TIM capture/compare register 3,         Address offset: 0x3C */
  __IO uint32_t CCR4;        /*!< TIM capture/compare register 4,         Address offset: 0x40 */
  __IO uint32_t BDTR;        /*!< TIM break and dead-time register,       Address offset: 0x44 */
  __IO uint32_t DCR;         /*!< TIM DMA control register,               Address offset: 0x48 */
  __IO uint32_t DMAR;        /*!< TIM DMA address for full transfer,      Address offset: 0x4C */
  uint32_t      RESERVED1;   /*!< Reserved, 0x50                                               */
  __IO uint32_t CCMR3;       /*!< TIM capture/compare mode register 3,    Address offset: 0x54 */
  __IO uint32_t CCR5;        /*!< TIM capture/compare register5,          Address offset: 0x58 */
  __IO uint32_t CCR6;        /*!< TIM capture/compare register6,          Address offset: 0x5C */
  __IO uint32_t AF1;         /*!< TIM alternate function option register 1, Address offset: 0x60 */
  __IO uint32_t AF2;         /*!< TIM alternate function option register 2, Address offset: 0x64 */
  __IO uint32_t TISEL;       /*!< TIM Input Selection register,           Address offset: 0x68 */
} TIM_TypeDef;
```



```
#define TIM2              ((TIM_TypeDef *) 0x40000000UL)
#define TIM3              ((TIM_TypeDef *) 0x40000400UL)
#define TIM4              ((TIM_TypeDef *) 0x40000800UL)
#define TIM5              ((TIM_TypeDef *) 0x40000C00UL)
#define TIM6              ((TIM_TypeDef *) 0x40001000UL)
#define TIM7              ((TIM_TypeDef *) 0x40001400UL)
```

Timer initalization structure

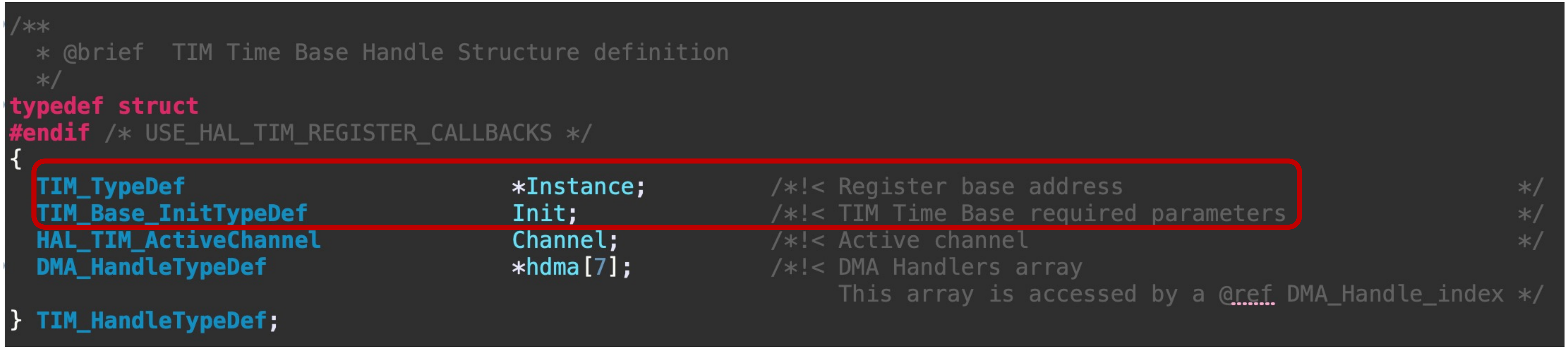


```c
/**
  * @brief  TIM Time base Configuration Structure definition
  */
typedef struct
{
  uint32_t Prescaler;         /*!< Specifies the prescaler value used to divide the TIM clock.
                                   This parameter can be a number between Min_Data = 0x0000 and Max_Data = 0xFFFF */

  uint32_t CounterMode;       /*!< Specifies the counter mode.
                                   This parameter can be a value of @ref TIM_Counter_Mode */

  uint32_t Period;            /*!< Specifies the period value to be loaded into the active
                                   Auto-Reload Register at the next update event.
                                   This parameter can be a number between Min_Data = 0x0000 and Max_Data = 0xFFFF.  */

  uint32_t AutoReloadPreload;  /*!< Specifies the auto-reload preload.
                                   This parameter can be a value of @ref TIM_AutoReloadPreload */
} TIM_Base_InitTypeDef;
```

$$f_{UpdateEvent} = \frac{f_{BusClock}}{(Prescaler+1)(Period+1)}$$

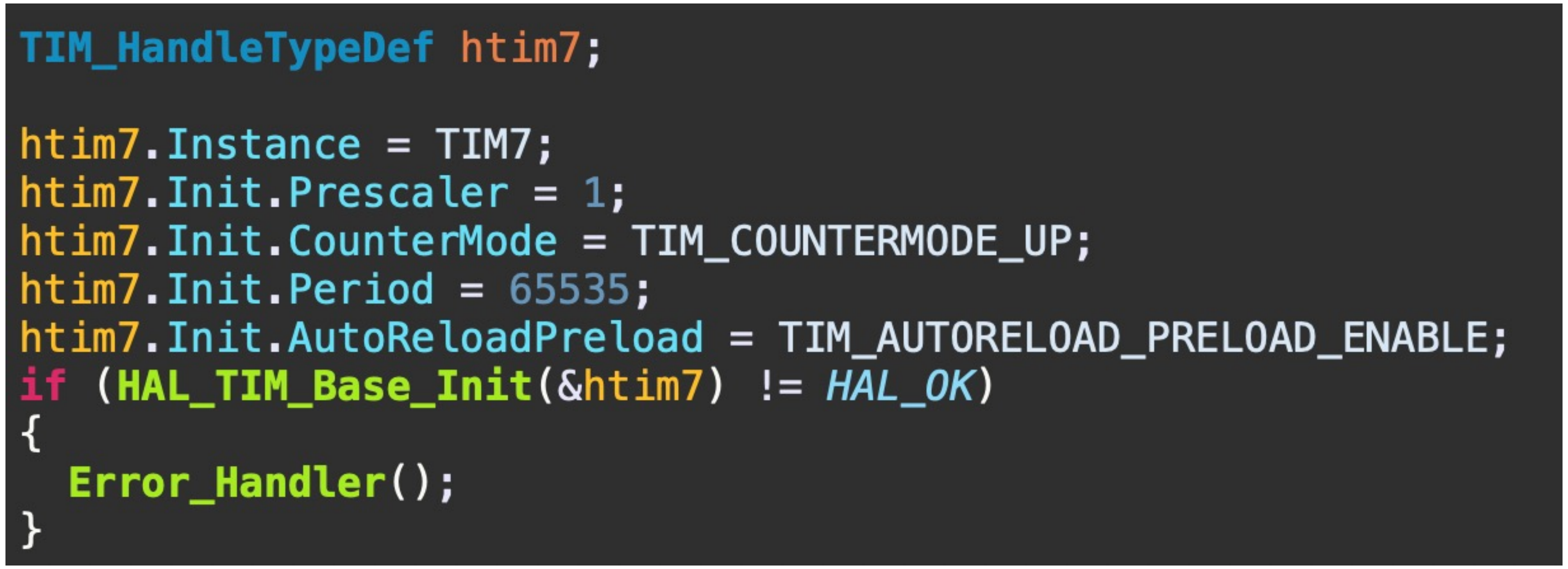
$$T_{UpdateEvent} = T_{BusClock} \times (Prescaler+1) \times (Period+1)$$

Timer handle structure



```
/**
  * @brief  TIM Time Base Handle Structure definition
  */
typedef struct
#endif /* USE_HAL_TIM_REGISTER_CALLBACKS */
{
  TIM_TypeDef                        *Instance;         /*!< Register base address                              */
  TIM_Base_InitTypeDef               Init;              /*!< TIM Time Base required parameters                  */
  HAL_TIM_ActiveChannel              Channel;           /*!< Active channel                                     */
  DMA_HandleTypeDef                  *hdma[7];          /*!< DMA Handlers array
                                                             This array is accessed by a @ref DMA_Handle_index */
} TIM_HandleTypeDef;
```

Timer initialization and start



```
TIM_HandleTypeDef htim7;

htim7.Instance = TIM7;
htim7.Init.Prescaler = 1;
htim7.Init.CounterMode = TIM_COUNTERMODE_UP;
htim7.Init.Period = 65535;
htim7.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
if (HAL_TIM_Base_Init(&htim7) != HAL_OK)
{
  Error_Handler();
}
```

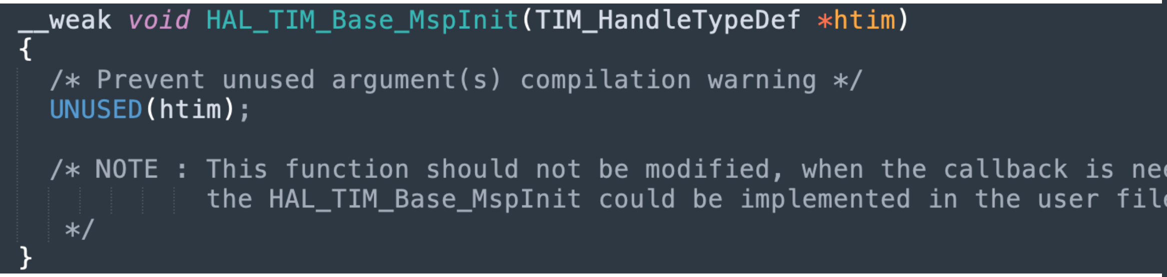
Start timer without interrupts on update events:

```
HAL_TIM_Base_Start(&htim7);
```

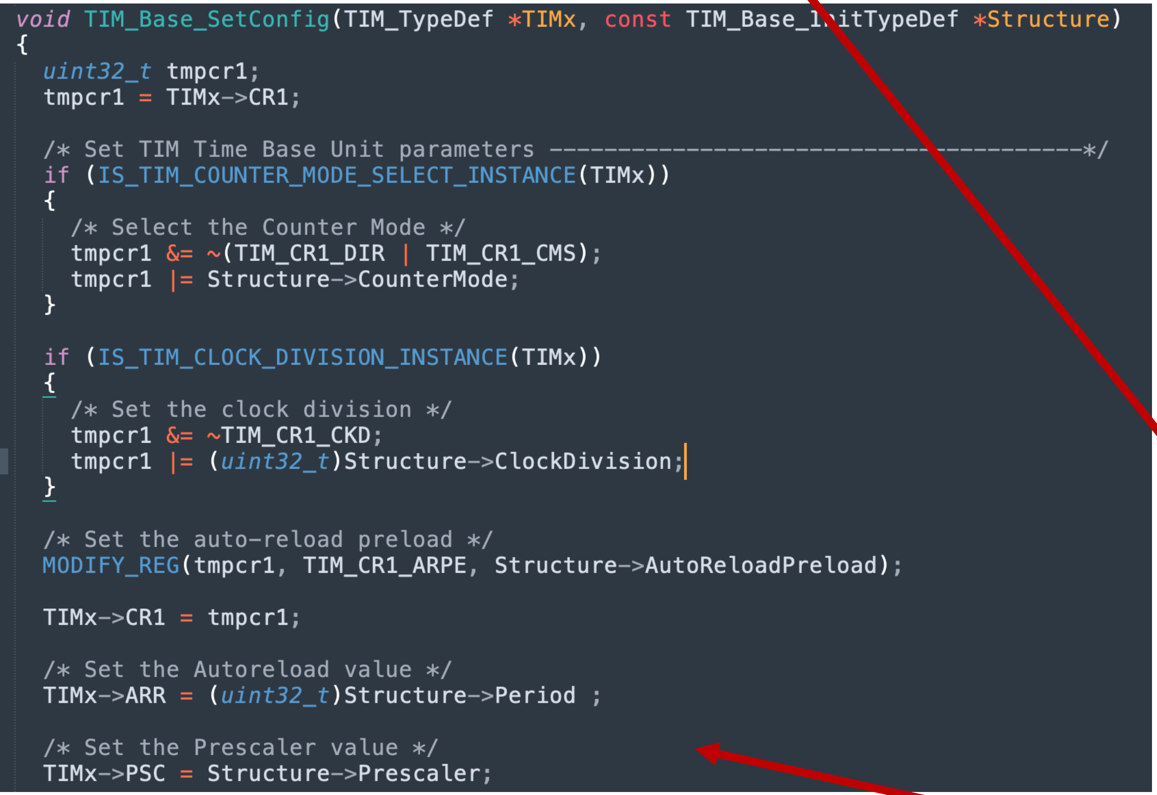
Start timer with interrupts on update events:

```
HAL_TIM_Base_Start_IT(&htim7);
```

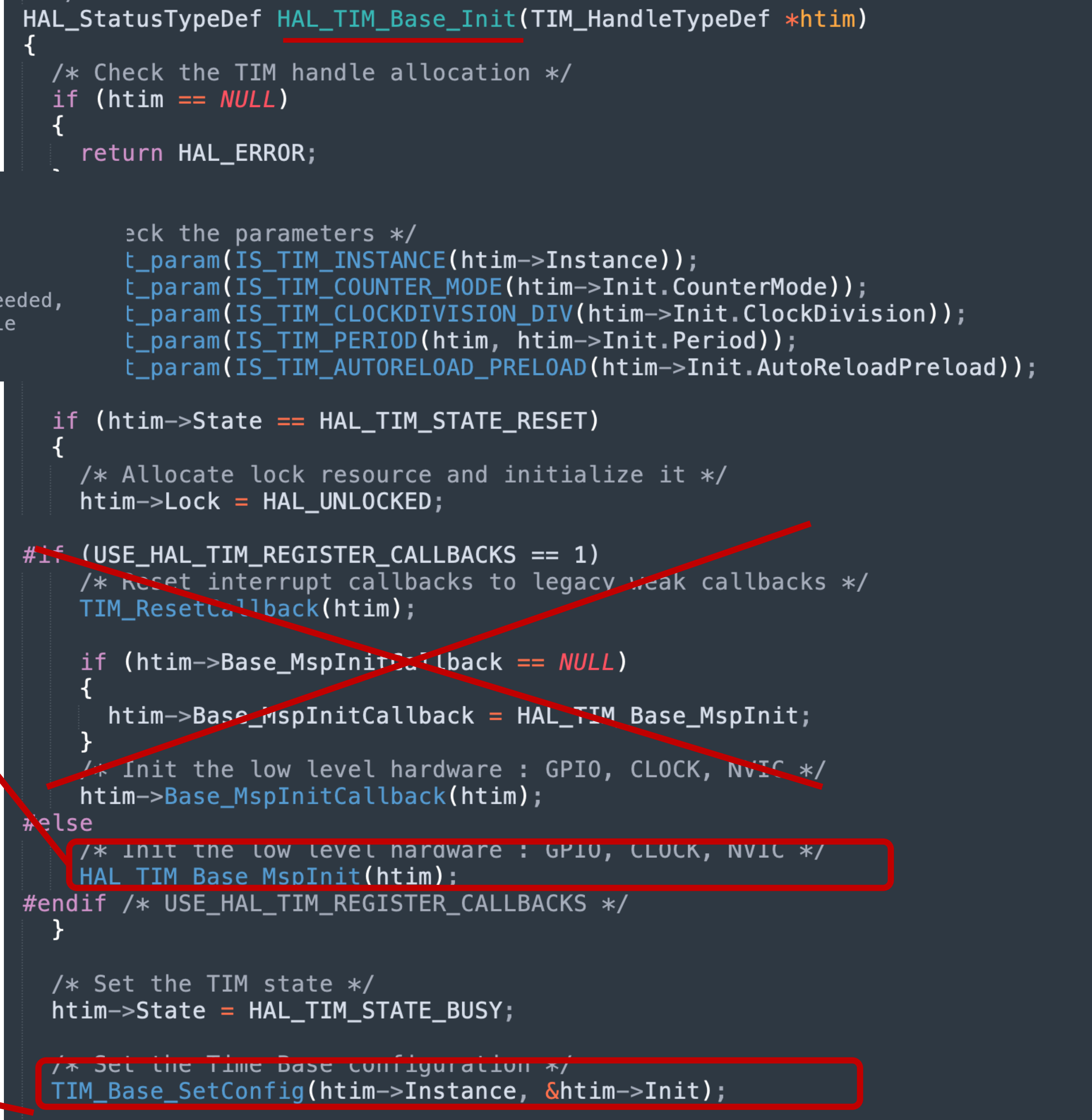
Timer initialization:



```
__weak void HAL_TIM_Base_MspInit(TIM_HandleTypeDef *htim)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(htim);

  /* NOTE : This function should not be modified, when the callback is needed,
            the HAL_TIM_Base_MspInit could be implemented in the user file
   */
}
```



```
void TIM_Base_SetConfig(TIM_TypeDef *TIMx, const TIM_Base_InitTypeDef *Structure)
{
  uint32_t tmpcr1;
  tmpcr1 = TIMx->CR1;

  /* Set TIM Time Base Unit parameters ---------------------------------------*/
  if (IS_TIM_COUNTER_MODE_SELECT_INSTANCE(TIMx))
  {
    /* Select the Counter Mode */
    tmpcr1 &= ~(TIM_CR1_DIR | TIM_CR1_CMS);
    tmpcr1 |= Structure->CounterMode;
  }

  if (IS_TIM_CLOCK_DIVISION_INSTANCE(TIMx))
  {
    /* Set the clock division */
    tmpcr1 &= ~TIM_CR1_CKD;
    tmpcr1 |= (uint32_t)Structure->ClockDivision;
  }

  /* Set the auto-reload preload */
  MODIFY_REG(tmpcr1, TIM_CR1_ARPE, Structure->AutoReloadPreload);

  TIMx->CR1 = tmpcr1;

  /* Set the Autoreload value */
  TIMx->ARR = (uint32_t)Structure->Period ;

  /* Set the Prescaler value */
  TIMx->PSC = Structure->Prescaler;
```

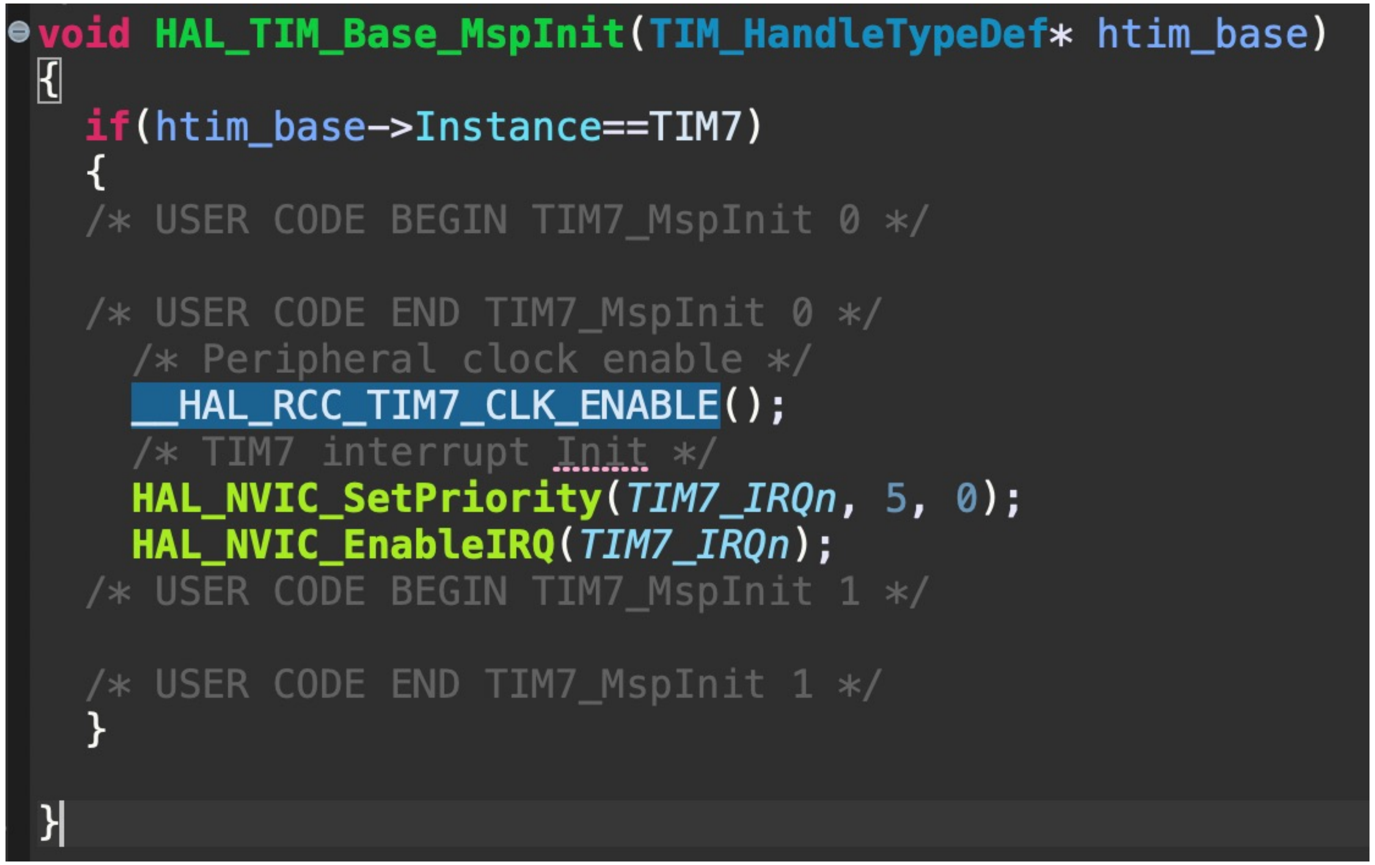


```
HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef *htim)
{
  /* Check the TIM handle allocation */
  if (htim == NULL)
  {
    return HAL_ERROR;

       eck the parameters */
       t_param(IS_TIM_INSTANCE(htim->Instance));
       t_param(IS_TIM_COUNTER_MODE(htim->Init.CounterMode));
       t_param(IS_TIM_CLOCKDIVISION_DIV(htim->Init.ClockDivision));
       t_param(IS_TIM_PERIOD(htim, htim->Init.Period));
       t_param(IS_TIM_AUTORELOAD_PRELOAD(htim->Init.AutoReloadPreload));

  if (htim->State == HAL_TIM_STATE_RESET)
  {
    /* Allocate lock resource and initialize it */
    htim->Lock = HAL_UNLOCKED;

#if (USE_HAL_TIM_REGISTER_CALLBACKS == 1)
    /* Reset interrupt callbacks to legacy weak callbacks */
    TIM_ResetCallback(htim);

    if (htim->Base_MspInitCallback == NULL)
    {
      htim->Base_MspInitCallback = HAL_TIM_Base_MspInit;
    }
    /* Init the low level hardware : GPIO, CLOCK, NVIC */
    htim->Base_MspInitCallback(htim);
#else
    /* Init the low level hardware : GPIO, CLOCK, NVIC */
    HAL_TIM_Base_MspInit(htim);
#endif /* USE_HAL_TIM_REGISTER_CALLBACKS */
  }

  /* Set the TIM state */
  htim->State = HAL_TIM_STATE_BUSY;

  /* Set the Time Base configuration */
  TIM_Base_SetConfig(htim->Instance, &htim->Init);
```

Timer initialization:

Implement your own Msp_Init function which:

- enables timer clock
- sets NVIC (priority+enable)

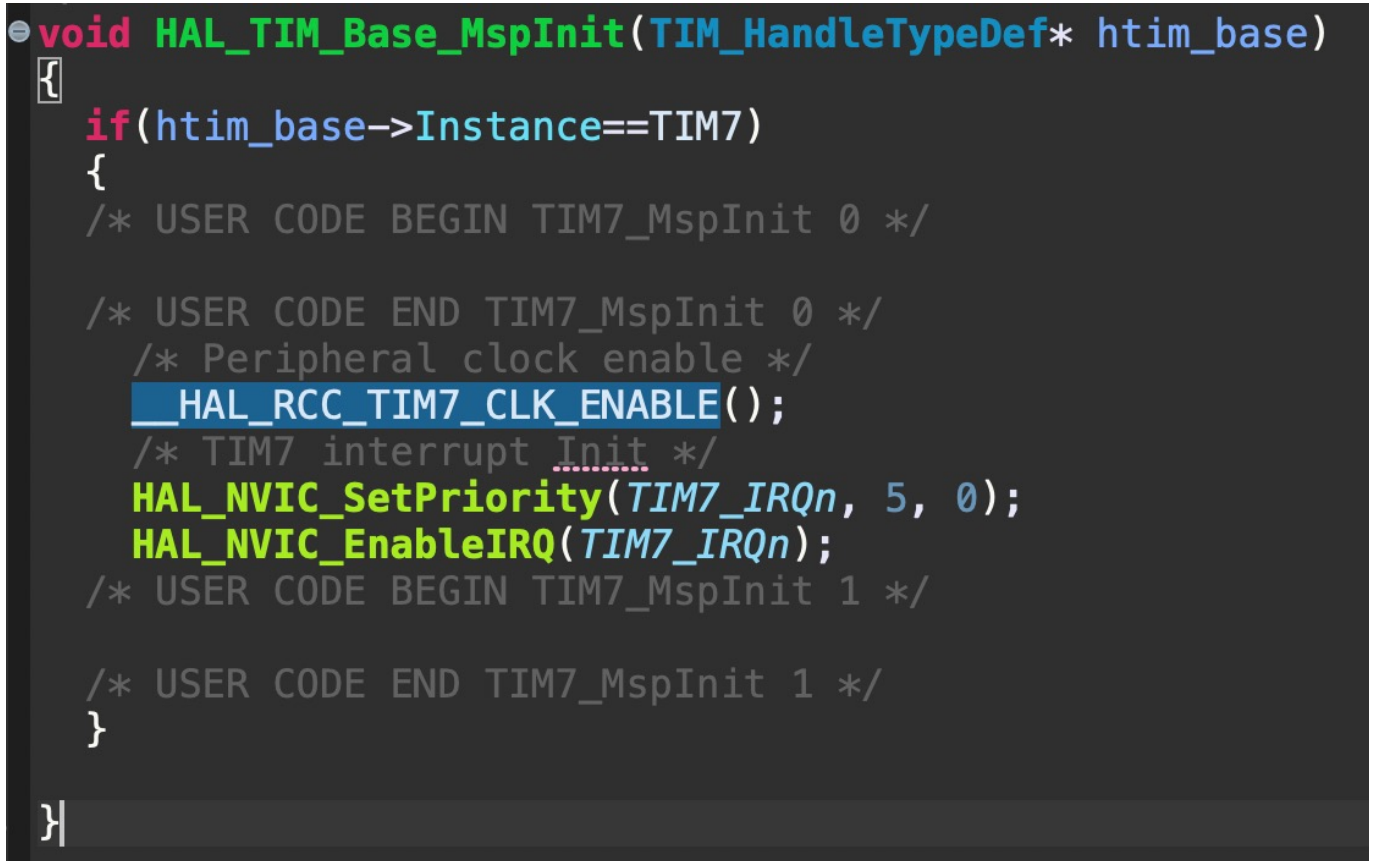


```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base)
{
  if(htim_base->Instance==TIM7)
  {
  /* USER CODE BEGIN TIM7_MspInit 0 */

  /* USER CODE END TIM7_MspInit 0 */
    /* Peripheral clock enable */
    __HAL_RCC_TIM7_CLK_ENABLE();
    /* TIM7 interrupt Init */
    HAL_NVIC_SetPriority(TIM7_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(TIM7_IRQn);
  /* USER CODE BEGIN TIM7_MspInit 1 */

  /* USER CODE END TIM7_MspInit 1 */
  }

}
```

Timer initialization:

Implement your own Msp_Init function which:

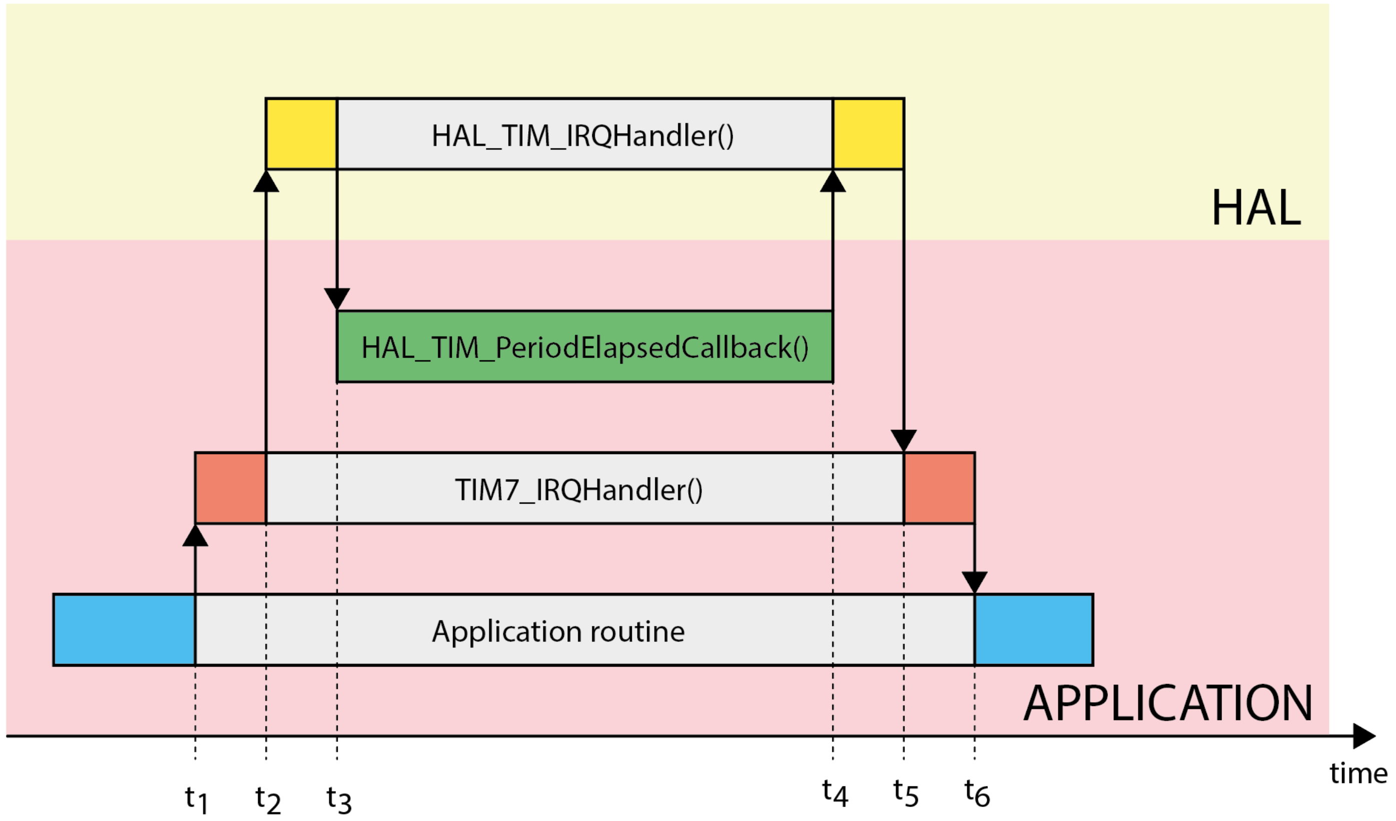
- enables timer clock
- sets NVIC (priority+enable)



```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base)
{
  if(htim_base->Instance==TIM7)
  {
  /* USER CODE BEGIN TIM7_MspInit 0 */

  /* USER CODE END TIM7_MspInit 0 */
    /* Peripheral clock enable */
    __HAL_RCC_TIM7_CLK_ENABLE();
    /* TIM7 interrupt Init */
    HAL_NVIC_SetPriority(TIM7_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(TIM7_IRQn);
  /* USER CODE BEGIN TIM7_MspInit 1 */

  /* USER CODE END TIM7_MspInit 1 */
  }

}
```

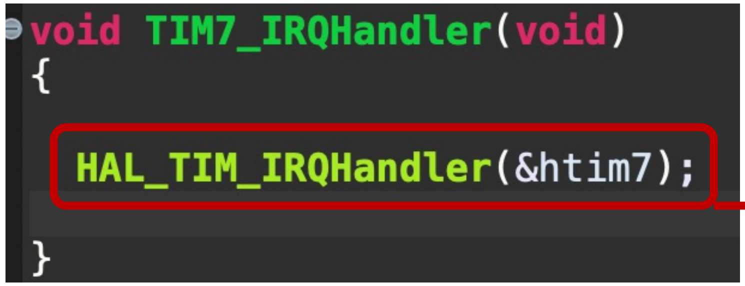
Timer interrupt processing in HAL:

- STM32 HAL library follows a callback-oriented programming model and uses callback functions:
- This approach decouples the handling of hardware events from the core library, allowing users to define their own behaviors in response to interrupts.
- It provides a flexible and modular approach to handle interrupts, making the HAL library adaptable to different application requirements.
- Users have the freedom to define specific actions in response to interrupts, ensuring that the HAL remains versatile and customizable across diverse embedded applications.

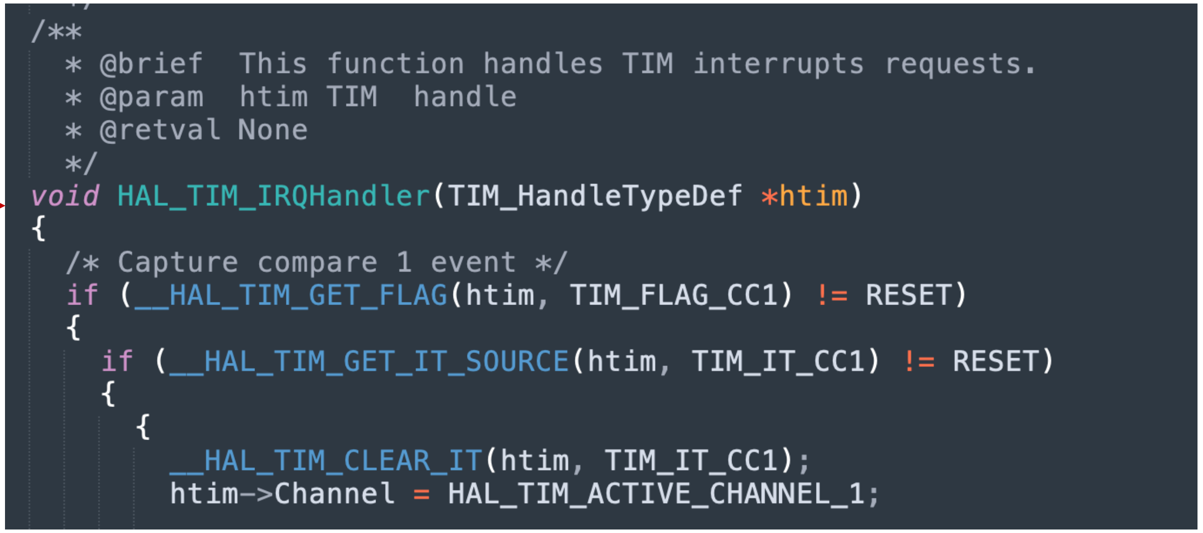


This mechanism is used by almost all IRQ handler routines inside the HAL.

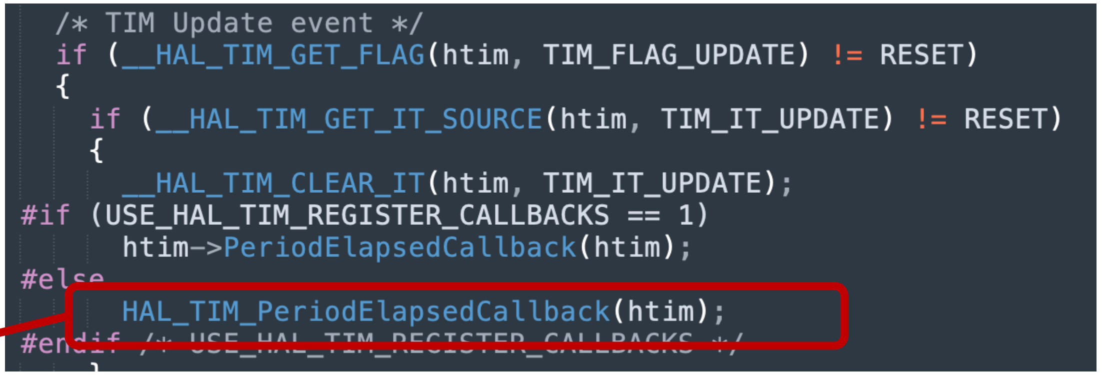
Timer interrupt processing in HAL:



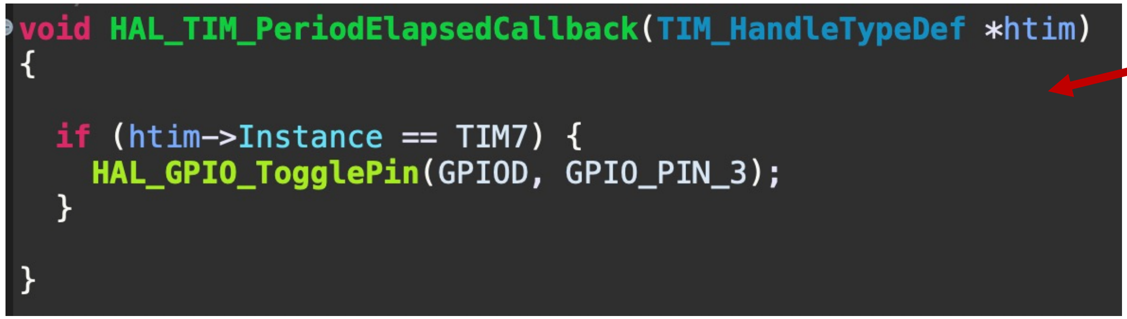
```c
void TIM7_IRQHandler(void)
{

  HAL_TIM_IRQHandler(&htim7);

}
```



```c
/**
  * @brief  This function handles TIM interrupts requests.
  * @param  htim TIM  handle
  * @retval None
  */
void HAL_TIM_IRQHandler(TIM_HandleTypeDef *htim)
{
  /* Capture compare 1 event */
  if (__HAL_TIM_GET_FLAG(htim, TIM_FLAG_CC1) != RESET)
  {
    if (__HAL_TIM_GET_IT_SOURCE(htim, TIM_IT_CC1) != RESET)
    {
      {
        __HAL_TIM_CLEAR_IT(htim, TIM_IT_CC1);
        htim->Channel = HAL_TIM_ACTIVE_CHANNEL_1;
```



```c
  /* TIM Update event */
  if (__HAL_TIM_GET_FLAG(htim, TIM_FLAG_UPDATE) != RESET)
  {
    if (__HAL_TIM_GET_IT_SOURCE(htim, TIM_IT_UPDATE) != RESET)
    {
      __HAL_TIM_CLEAR_IT(htim, TIM_IT_UPDATE);
#if (USE_HAL_TIM_REGISTER_CALLBACKS == 1)
      htim->PeriodElapsedCallback(htim);
#else
      HAL_TIM_PeriodElapsedCallback(htim);
#endif /* USE_HAL_TIM_REGISTER_CALLBACKS */
```



```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{

  if (htim->Instance == TIM7) {
    HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_3);
  }

}
```