8. Contadores.

Los contadores son módulos periféricos del procesador que sirven para contar pulsos internos y pulsos externos. También sirven para medir la longitud de un pulso de una señal de entrada (IC, captura de entrada), tener un cambio de estado en una salida cuando se compara un valor con conteo (OC o salida por comparación), o también para generar pulsos periódicos con ancho variable o PWM. Los contadores son módulos muy usados para medir tiempos pequeños, de unos pocos microsegundos, comparado con el módulo RTC que sirve para medir tiempos grandes, mayores a un segundo. El procesador STM32F103C8T6 es un procesador de mediana densidad que tiene siete contadores que son: Tres de 16 bits con hasta cuatro IC/ OC/ PWM (TIM2, TIM3 y TIM4), un contador avanzado de 16 bits (TIM1), especial para control de motores, dos contadores watchdog (independiente y de ventana) y un contador SysTick de 24 bits con conteo regresivo.

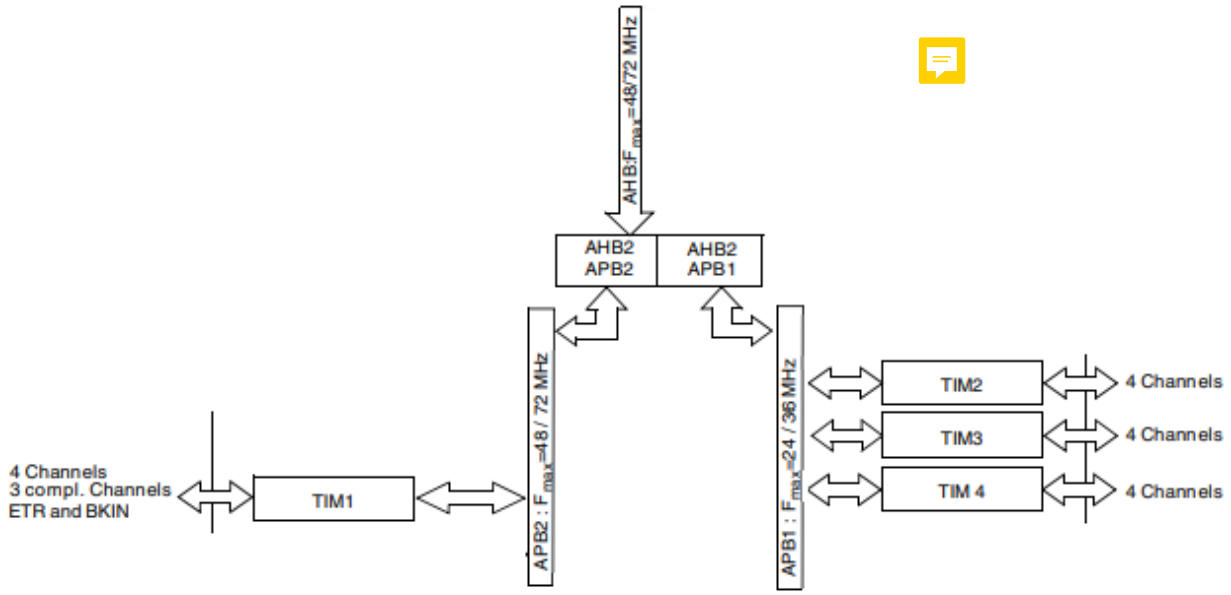

Figura 47. Contadores unidos al bus APB1 y APB2 del microcontrolador.

Como se puede ver en la figura 47, el contador TIM1 está conectado al bus APB2, mientras que los contadores TIM2, TIM3 y TIM4 están conectados al bus APB1. Para habilitar el reloj del contador TIM1 se usa la función

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE)
```

Para habilitar el reloj de los contadores TIM2, TIM3 y TIM4 se usa la función:

```
RCC_APB1PeriphClockCmd( RCC_APB1Periph_TIMx, ENABLE)
```

Donde `x` puede ser 2, para `TIM2`, 3 para `TIM3` y 4 para `TIM4`.

Cada contador tiene cuatro canales dispuestos en los siguientes pines:
TIM1: PA8, PA9, PA10 y PA11.
TIM2: PA15, PB3, PA2 y PA3.
TIM3: PA6, PA7, PB0 y PB1.

TIM4: PB6, PB7, PB8 y PB9.

Contador TIM1:

El contador TIM1 es un contador avanzado de 16 bits. Se dice que es avanzado porque tiene unas características como para generar señales para controlar motores. Este contador se puede usar para medir la longitud de un pulso de una señal de entrada, (Input Compare, IC) o para generar señales de salida en los modos de salida por comparación (Output Compare, OC), modulación por ancho de pulso (PWM) o salida de un solo pulso. El contador TIM1 puede contar hacia arriba, o sea, en incremento, hacia abajo, hacia arriba y abajo, y se puede autocargar con un valor definido. Tiene un prescaler de 16 bits, es decir, un divisor programable de la frecuencia de entrada que la puede dividir hasta por 65.536. En la figura 48 se pueden ver los diferentes registros y unidades de control de salida y entrada del contador TIM1. También se pueden ver varios pines asociados al contador TIM1, como para entrar una señal que se va a medir en tiempo o para sacar una señal del contador.

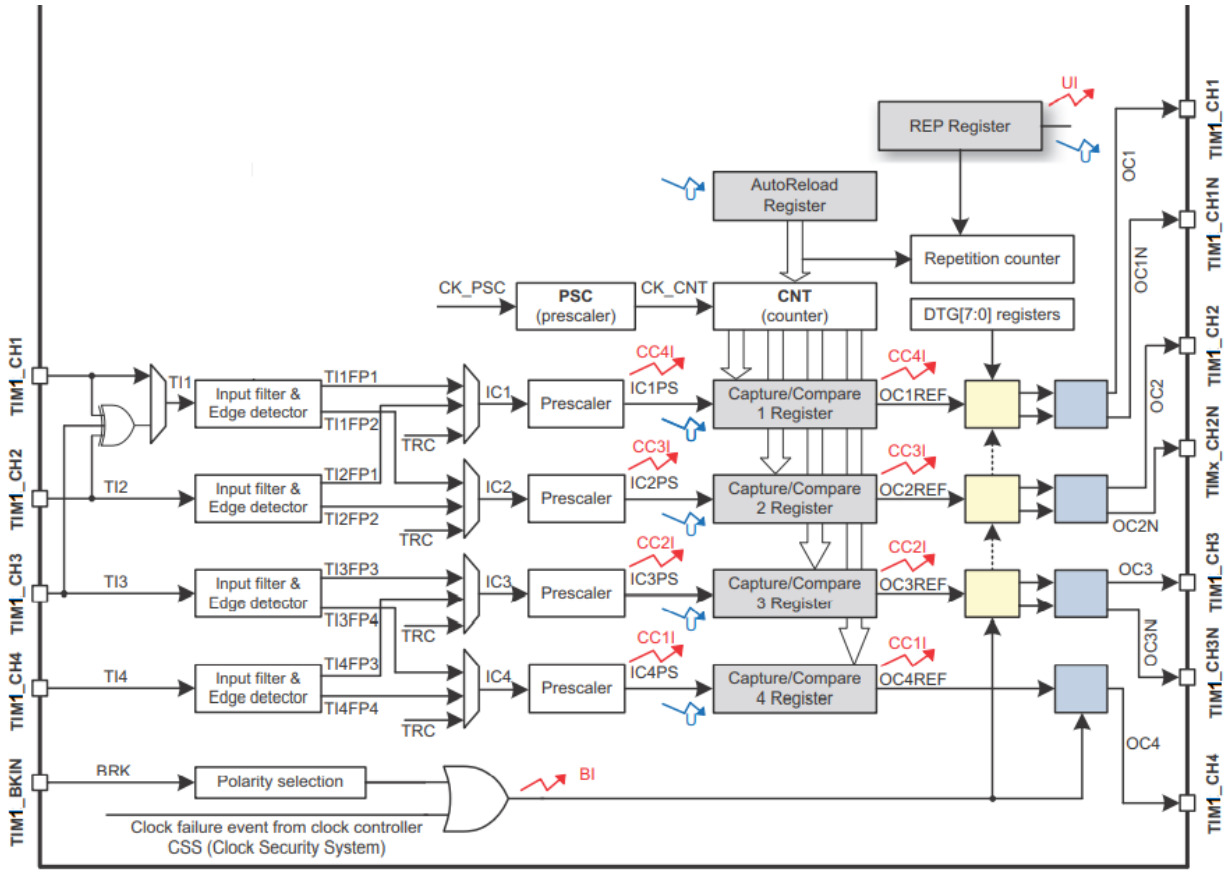

Figura 48. Descripción de los canales de entrada, salida y registros del contador TM1.

Los pulsos de conteo que entran al contador TIM1 pueden ser cualquiera de los siguientes:

- El reloj interno (CK_INT)
- Un reloj externo entrando por un pin en modo 1
- Un reloj externo entrando por un pin en modo 2, ETR.

- La salida de otro contador interno, ITRx

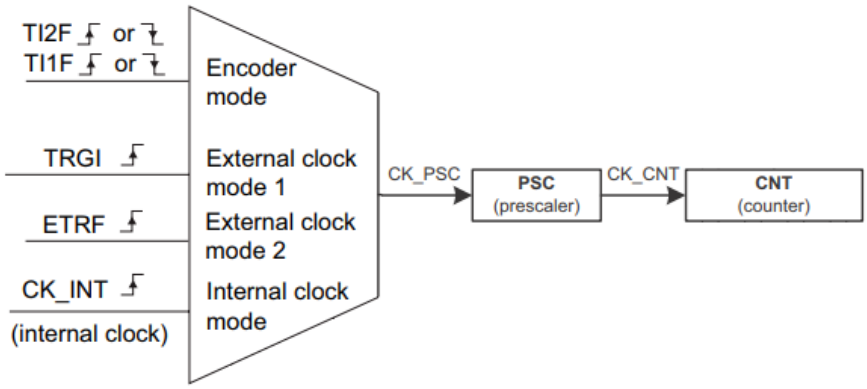

Figura 49. Entrada de pulsos al contador TIM1.

La salida del selector de la frecuencia que se escoja, CK_PSC, va a un prescaler o divisor de frecuencia programable de 16 bits, PSC, y luego al contador de pulsos, CNT.
Si se quiere escoger el reloj interno, se ejecuta la función `TIM_InternalClockConfig(TIM1).`

El reloj interno para cada uno de los contadores se muestra en la siguiente figura.

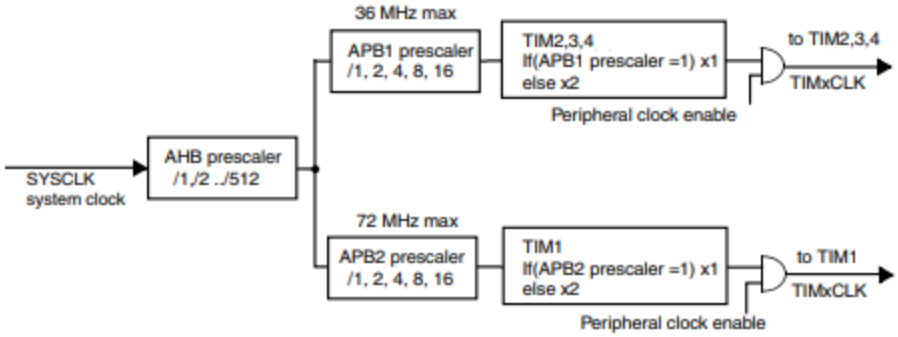

Figura 50. Frecuencia interna para cada uno de los contadores.

8.1. Contador básico.

El contador TIM1 como contador básico se configura con los pasos siguientes:

- Habilitar el reloj del periférico Tim1 que está en APB2 con la función `RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE).`
- Escoger el reloj fuente del contador. Para el caso de escoger como fuente el reloj interno, se hace con la función `TIM_InternalClockConfig(TIM1).`
- Establecer la división del reloj fuente. Para esto se debe conocer cuanto es el valor del prescaler y del contador. Tiempo= prescaler x contador/ frecuencia reloj fuente.
- Si hay interrupción, hacer la configuracion de la interrupción por actualización del contador.

Como ejemplo, suponga que se va a encender y apagar el led de la tarjeta a exactamente un segundo. Si el SysClk es de 56 Mhz, igual a la frecuencia del bus APB2, entre el preescaler y el

contador deben contar 56 millones de pulsos. Lo máximo que puede contar el preescaler es 65536 pulsos porque es de 16 bits, entonces se puede poner el prescaler en 56.000 y el contador a que cuente 1.000 pulsos, o sea, 56.000 x 1.000= 56M. Las características del contador se pueden programar con la función de inicialización de una estructura del tipo `TIM_TimeBaseInitTypeDef, TIM_TimeBaseStructInit( &TIM_TimeBaseInitStruct)`. Esta inicialización por defecto asigna las siguientes características:

- `TIM_Period = 0xFFFF`. Este es el valor de recarga del contador.
- `TIM_Prescaler = 0x0000`. Este es el valor del prescaler.
- `TIM_ClockDivision = TIM_CKD_DIV1`. Puede también ser 2 o 4.
- `TIM_CounterMode = TIM_CounterMode_Up`. Aquí cuenta hacia arriba, pero también puede contar hacia abajo.
- `TIM_RepetitionCounter = 0x0000`. Este valor es para el modo PWM.

Hay que cambiar algunos de estos valores para el conteo que se necesita:

- `TIM_TimeBaseInitStruct.TIM_Period = 1000;`
- `TIM_TimeBaseInitStruct.TIM_Prescaler = 56000;`

Con estos valores se configura el contador mediante la función `TIM_TimeBaseInit(TIM1, &TIM_TimeBaseInitStruct)` y luego hay que empezar el conteo con la función `TIM_Cmd(TIM1, ENABLE)`. El conteo es hacia arriba en esta configuración que se hizo, así que cuando el contador llegue a cero se tiene que actualizar con el valor que se puso como periodo. Una vez llegue a cero, la bandera de actualización se activa y es entonces esta bandera que se tiene que leer para saber que un segundo ha transcurrido.

```
FlagStatus TIM_GetFlagStatus(TIM1, TIM_FLAG_Update)
```

El programa completo queda de la siguiente manera:

```
int main(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
    Sysclk_56M();
    LED_Init();

    // configuración del contador
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE);
    TIM_InternalClockConfig(TIM1);
    TIM_TimeBaseStructInit(&TIM_TimeBaseInitStruct);
    TIM_TimeBaseInitStruct.TIM_Period = 1000;
    TIM_TimeBaseInitStruct.TIM_Prescaler = 56000;
    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseInitStruct);
    TIM_Cmd(TIM1, ENABLE);

  while(1)
  {
      GPIO_ResetBits(GPIOB, GPIO_Pin_12);// enciende led
      while(TIM_GetFlagStatus(TIM1, TIM_FLAG_Update) == RESET);
      TIM_ClearFlag(TIM1, TIM_FLAG_Update);
      GPIO_SetBits(GPIOB, GPIO_Pin_12);// apaga led
      while(TIM_GetFlagStatus(TIM1, TIM_FLAG_Update) == RESET);
      TIM_ClearFlag(TIM1, TIM_FLAG_Update);
```

```
  }
}
```

Este mismo programa se puede hacer usando interrupciones de forma que cada evento genere una interrupción y dentro de la función de interrupción hacer el encendido y apagado del led. Aquí se presenta como una función:

```
void Tim1Init(void)
{
    // poner el reloj al contador Tim1
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE);
    // se escoge el reloj interno, el que sale de APB2, o sea, 56 Mhz, como entrada al contador Tim1
    TIM_InternalClockConfig(TIM1);

    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct; // creacion de estructura
    TIM_TimeBaseStructInit(&TIM_TimeBaseInitStruct); // inicialización de datos de la estructura
    // son 56M de entrada, dividido en 56000, queda 1000
    TIM_TimeBaseInitStruct.TIM_Prescaler = 56000; // con este valor se divide el reloj de entrada
    TIM_TimeBaseInitStruct.TIM_Period = 1000;
    // 56.000x1.000= 56M. O sea, queda 1 hz saliendo del contador
    TIM_TimeBaseInit(TIM1, &TIM_TimeBaseInitStruct);
    TIM_Cmd(TIM1, ENABLE);// habilitación

    // configuración de la interrupción cada segundo
    TIM_ITConfig(TIM1, TIM_IT_Update, ENABLE);
    NVIC_InitTypeDef NVIC_Struct;
    NVIC_Struct.NVIC_IRQChannel = TIM1_UP_IRQn;
    NVIC_Struct.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_Struct.NVIC_IRQChannelSubPriority = 0;
    NVIC_Struct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_Struct);
    return;
}
void TIM1_UP_IRQHandler(void)
{
    // verificar que la bandera de interrupción es la de actualización del contador
    if (TIM_GetITStatus(TIM1, TIM_IT_Update) != RESET)
    {
        // cambia el estado del led
        if(GPIO_ReadOutputDataBit(GPIOB, GPIO_Pin_12)== Bit_RESET)
            GPIO_SetBits(GPIOB, GPIO_Pin_12);
        else
            GPIO_ResetBits(GPIOB, GPIO_Pin_12);
        TIM_ClearITPendingBit(TIM1, TIM_IT_Update);// borra la bandera que hizo interrumpir
    }

}
```

Dentro del while(1) no hay que observar las banderas, pues el cambio de estado del led se hace en la función de tratamiento a interrupción `TIM1_UP_IRQHandler()`.

8.2. Captura de entrada.

En este modo se mide el tiempo entre eventos de una entrada a un pin. Cada contador dispone de cuatro canales que son pines físicos, que para el caso del modo de captura de entrada, es un pin que se configura como entrada en modo flotante o con resistencias de pul-up o pull-down, y para el caso de salida de comparación, es un pin de salida, en modo push- pull o open-drain.

```
void TIM_DeInit(TIM_TypeDef* TIMx);
void TIM_TimeBaseInit(TIM_TypeDef* TIMx, TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct);
void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
void TIM_OC2Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
void TIM_OC3Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
void TIM_OC4Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
void TIM_ICInit(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct);
void TIM_PWMIConfig(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct);
void TIM_BDTRConfig(TIM_TypeDef* TIMx, TIM_BDTRInitTypeDef *TIM_BDTRInitStruct);
void TIM_TimeBaseStructInit(TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct);
void TIM_OCStructInit(TIM_OCInitTypeDef* TIM_OCInitStruct);
void TIM_ICStructInit(TIM_ICInitTypeDef* TIM_ICInitStruct);
void TIM_BDTRStructInit(TIM_BDTRInitTypeDef* TIM_BDTRInitStruct);
void TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState);
void TIM_CtrlPWMOutputs(TIM_TypeDef* TIMx, FunctionalState NewState);
void TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState NewState);
void TIM_GenerateEvent(TIM_TypeDef* TIMx, uint16_t TIM_EventSource);
void TIM_DMAConfig(TIM_TypeDef* TIMx, uint16_t TIM_DMABase, uint16_t TIM_DMABurstLength);
void TIM_DMACmd(TIM_TypeDef* TIMx, uint16_t TIM_DMASource, FunctionalState NewState);
void TIM_InternalClockConfig(TIM_TypeDef* TIMx);
void TIM_ITRxExternalClockConfig(TIM_TypeDef* TIMx, uint16_t TIM_InputTriggerSource);
void TIM_TIxExternalClockConfig(TIM_TypeDef* TIMx, uint16_t TIM_TIxExternalCLKSource,
                                uint16_t TIM_ICPolarity, uint16_t ICFilter);
void TIM_ETRClockMode1Config(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t
TIM_ExtTRGPolarity,
                             uint16_t ExtTRGFilter);
void TIM_ETRClockMode2Config(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler,
                             uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter);
void TIM_ETRConfig(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t
TIM_ExtTRGPolarity,
                   uint16_t ExtTRGFilter);
void TIM_PrescalerConfig(TIM_TypeDef* TIMx, uint16_t Prescaler, uint16_t TIM_PSCReloadMode);
void TIM_CounterModeConfig(TIM_TypeDef* TIMx, uint16_t TIM_CounterMode);
void TIM_SelectInputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_InputTriggerSource);
void TIM_EncoderInterfaceConfig(TIM_TypeDef* TIMx, uint16_t TIM_EncoderMode,
                                uint16_t TIM_IC1Polarity, uint16_t TIM_IC2Polarity);
void TIM_ForcedOC1Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);
void TIM_ForcedOC2Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);
void TIM_ForcedOC3Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);
void TIM_ForcedOC4Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);
void TIM_ARRPreloadConfig(TIM_TypeDef* TIMx, FunctionalState NewState);
void TIM_SelectCOM(TIM_TypeDef* TIMx, FunctionalState NewState);
```

```
void TIM_SelectCCDMA(TIM_TypeDef* TIMx, FunctionalState NewState);
void TIM_CCPreloadControl(TIM_TypeDef* TIMx, FunctionalState NewState);
void TIM_OC1PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
void TIM_OC2PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
void TIM_OC3PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
void TIM_OC4PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
void TIM_OC1FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
void TIM_OC2FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
void TIM_OC3FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
void TIM_OC4FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
void TIM_ClearOC1Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
void TIM_ClearOC2Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
void TIM_ClearOC3Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
void TIM_ClearOC4Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
void TIM_OC1PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
void TIM_OC1NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);
void TIM_OC2PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
void TIM_OC2NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);
void TIM_OC3PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
void TIM_OC3NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);
void TIM_OC4PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
void TIM_CCxCmd(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t TIM_CCx);
void TIM_CCxNCmd(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t TIM_CCxN);
void TIM_SelectOCxM(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t TIM_OCMode);
void TIM_UpdateDisableConfig(TIM_TypeDef* TIMx, FunctionalState NewState);
void TIM_UpdateRequestConfig(TIM_TypeDef* TIMx, uint16_t TIM_UpdateSource);
void TIM_SelectHallSensor(TIM_TypeDef* TIMx, FunctionalState NewState);
void TIM_SelectOnePulseMode(TIM_TypeDef* TIMx, uint16_t TIM_OPMode);
void TIM_SelectOutputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_TRGOSource);
void TIM_SelectSlaveMode(TIM_TypeDef* TIMx, uint16_t TIM_SlaveMode);
void TIM_SelectMasterSlaveMode(TIM_TypeDef* TIMx, uint16_t TIM_MasterSlaveMode);
void TIM_SetCounter(TIM_TypeDef* TIMx, uint16_t Counter);
void TIM_SetAutoreload(TIM_TypeDef* TIMx, uint16_t Autoreload);
void TIM_SetCompare1(TIM_TypeDef* TIMx, uint16_t Compare1);
void TIM_SetCompare2(TIM_TypeDef* TIMx, uint16_t Compare2);
void TIM_SetCompare3(TIM_TypeDef* TIMx, uint16_t Compare3);
void TIM_SetCompare4(TIM_TypeDef* TIMx, uint16_t Compare4);
void TIM_SetIC1Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
void TIM_SetIC2Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
void TIM_SetIC3Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
void TIM_SetIC4Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
void TIM_SetClockDivision(TIM_TypeDef* TIMx, uint16_t TIM_CKD);
uint16_t TIM_GetCapture1(TIM_TypeDef* TIMx);
uint16_t TIM_GetCapture2(TIM_TypeDef* TIMx);
uint16_t TIM_GetCapture3(TIM_TypeDef* TIMx);
uint16_t TIM_GetCapture4(TIM_TypeDef* TIMx);
uint16_t TIM_GetCounter(TIM_TypeDef* TIMx);
uint16_t TIM_GetPrescaler(TIM_TypeDef* TIMx);
FlagStatus TIM_GetFlagStatus(TIM_TypeDef* TIMx, uint16_t TIM_FLAG);
void TIM_ClearFlag(TIM_TypeDef* TIMx, uint16_t TIM_FLAG);
```