DISCIPLINA EA701

Introdução aos Sistemas Embarcados

ROTEIRO 6: TEMPORIZADORES AVANÇADOS, PINOS MULTIPLEXÁVEIS, PWM, OUTPUT COMPARE (OC), INPUT CAPTURE (IC), RELÓGIO EXTERNO, INTERFACES (PONTE-H, ENCODER)

Profs. Antonio A. F. Quevedo e Wu Shin-Ting

FEEC / UNICAMP

Revisado e modificado em abril de 2025 por Ting com auxílio do Chatgpt
Revisado em setembro de 2024

INTRODUÇÃO

Dando continuidade ao estudo dos *timers*, exploraremos funções avançadas que estendem o *timer* padrão. A versatilidade dos temporizadores permite implementar funcionalidades que **interagem diretamente com o mundo físico**, oferecendo controle preciso de tempo e coordenação de eventos em tempo real. No Roteiro 6, exemplificamos temporizadores de pulso único, cujos sinais são acessíveis externamente. Neste roteiro, apresentaremos outras funcionalidades de interação com o mundo externo, como a **Captura de Entradas** (em inglês, *Input Capture* – IC), que registra o valor do contador no momento de um evento externo, e a **Comparação de Saídas** (em inglês, *Output Compare* – OC), que compara o valor do contador com um valor predefinido para executar ações específicas, como gerar pulsos. A **Modulação por Largura de Pulso** (em inglês, *Pulse Width Modulation* – PWM) permite a geração de sinais de saída com largura de pulso variável, controlando a potência fornecida a dispositivos. Além disso, o **Contador de Eventos Externos** (em inglês, *Pulse Counter*) realiza a contagem de pulsos ou eventos que ocorrem fora do microcontrolador.

Essas funcionalidades, padronizadas em muitos microcontroladores, facilitam o controle preciso de eventos temporais e a geração de sinais pulsados em intervalos definidos, encontrando aplicações em diversas áreas, desde o controle de LEDs e motores até sistemas de comunicação e medição de alta precisão.

Ao interagir com o mundo físico, deve-se considerar a interface entre o microcontrolador e o mundo físico, que pode envolver dispositivos com faixas de operação de tensão e corrente significativamente diferentes das que o microcontrolador pode diretamente fornecer ou tolerar. Muitos microcontroladores operam dentro de faixas de tensão e corrente bem limitadas, o que significa que suas saídas precisam ser adaptadas para controlar dispositivos externos que podem exigir tensões e correntes muito maiores ou menores. A integração bem-sucedida dessas funcionalidades avançadas dos temporizadores com o mundo físico frequentemente exige circuitos de interface, como *drivers* de potência, amplificadores ou conversores de nível, para garantir que o microcontrolador possa controlar dispositivos externos de forma segura e eficaz.

PROJETOS-EXEMPLO

Abordaremos quatro problemas específicos que ilustram o uso das funcionalidades que envolvem interações com o mundo físico externo: a geração de sinais de saída sincronizados com um sinal de referência, a medição precisa de períodos de sinais, o controle de potência de alimentação de dispositivos e a contagem de eventos externos. As técnicas envolvidas incluem, respectivamente, comparação de saídas (OC), captura de entradas (IC), modulação por largura de pulso (PWM) e contador de eventos externos.

Projeto de *Output Compare*

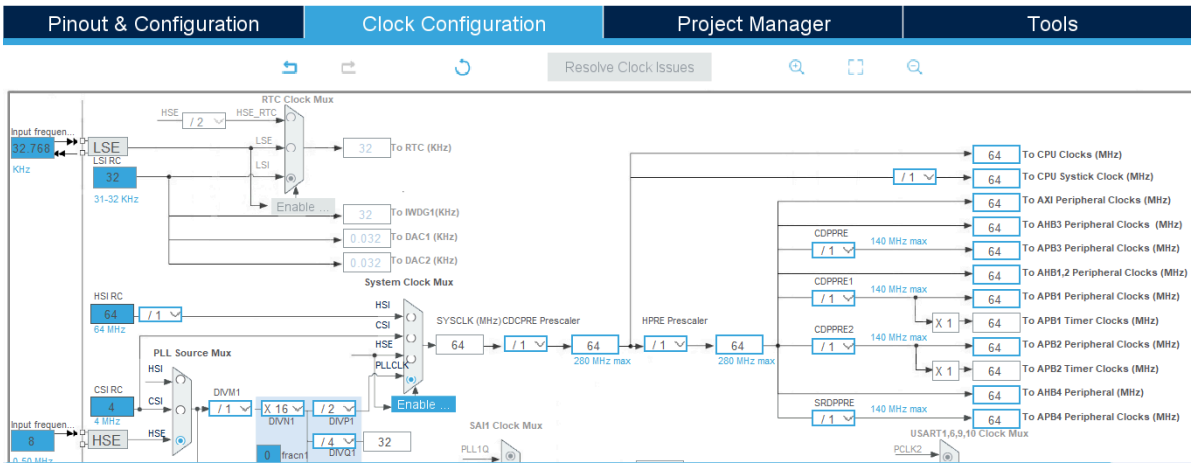
Você já pensou em gerar um sinal sincronizado com um sinal de referência? O desafio é o alinhamento preciso desses dois sinais devido às latências. Tem ideia como contornar este problema usando os temporizadores disponíveis nos microcontroladores? Vamos criar um projeto que demonstra como os temporizadores de um microcontrolador podem ser utilizados para superar esse desafio? O projeto é dividido em três partes:

1. **Geração de sinais sincronizados em baixa frequência:** Utilizaremos um temporizador para gerar no modo OC sinais em baixa frequência com 2 valores distintos de comparação. Esses sinais demonstrarão como é possível gerar múltiplos sinais sincronizados com precisão.
2. **Verificação da precisão em alta frequência:** Um segundo temporizador será configurado no modo OC para gerar um sinal em alta frequência.
3. **Demonstração da latência de interrupções:** Um terceiro temporizador será utilizado para gerar um sinal periódico numa saída GPIO através de interrupções. Esse sinal

demonstrará, em comparação com o sinal gerado no modo OC, como a latência de interrupções pode afetar a precisão da geração de sinais temporais.

Os pinos de saída dos temporizadores serão conectados aos canais de um analisador lógico, permitindo verificar a precisão da sincronização em tempo real.

1. Crie um projeto usando o *Cube* com o nome "Output_Compare", **sem inicialização dos periféricos em modo *default***. Ative o módulo *Debug* como "Serial Wire". O projeto visa comparar sinais periódicos gerados por duas abordagens distintas: uma baseada em interrupções periódicas, que envolve a alteração programática dos estados do sinal, e outra utilizando o modo *Output Compare*, onde as mudanças são realizadas diretamente em *hardware*. A comparação desses métodos permitirá avaliar a precisão de cada abordagem na geração e controle de sinais periódicos. Na aba de configuração de *clock*, localize o bloco entre "*CDCPRE Prescaler*" e "*HPRE Prescaler*", e mude o valor para 64 (MHz). O *Cube* irá calcular automaticamente os valores de multiplicação de frequência para que a frequência geral do sistema mude para o valor digitado.



2. Gere o código de configuração do sistema de sinais de relógio. Abra o arquivo `Core/Src/main.c`.

3. Comente (Remova) a declaração, a definição e a chamada da função `MX_GPIO_Init(void)`. Adicione a declaração das seguintes funções de configuração dos periféricos GPIOB, GPIOE, TIM1, TIM2 e TIM6 no escopo `/* USER CODE BEGIN PFP */`:

```
//static void MX_GPIO_Init(void);
/* USER CODE BEGIN PFP */
void PB11_PInit(void);
void PE11_PInit(void);
void PE9_PInit(void);
void TIM2C4_PInit(void);
void TIM1C2_PInit(void);
void TIM6_PInit(void);
/* USER CODE END PFP */
```

4. Vamos definir as funções de configuração dos pinos PB11 e PE11, multiplexados para os periféricos TIM2 e TIM1, respectivamente.

```
void PB11_PInit (void) {
   // Ativa GPIOB (PB11)
   RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk;
   // Configurar PB11 como função alternativa
   GPIOB->MODER &= ~GPIO_MODER_MODE11_Msk;
   GPIOB->MODER |= GPIO_MODER_MODE11_1;
   // Selecionar AF1 para PE11
   GPIOB->AFR[1] &= ~(0xF << GPIO_AFRH_AFSEL11_Pos);
   GPIOB->AFR[1] |= (1 << GPIO_AFRH_AFSEL11_Pos);
   // Configurar PB11 como push-pull
   GPIOB->OTYPER &= ~GPIO_OTYPER_OT11;
}

void PE11_PInit (void) {
   // Ativa GPIOE (PE11)
   RCC->AHB4ENR |= RCC_AHB4ENR_GPIOEEN_Msk;
   // Configurar PE11 como função alternativa
   GPIOE->MODER &= ~GPIO_MODER_MODE11_Msk;
   GPIOE->MODER |= GPIO_MODER_MODE11_1;
   // Selecionar AF1 para PE11
   GPIOE->AFR[1] &= ~(0xF << GPIO_AFRH_AFSEL11_Pos);
   GPIOE->AFR[1] |= (1 << GPIO_AFRH_AFSEL11_Pos);
   // Configurar PE11 como push-pull
   GPIOE->OTYPER &= ~GPIO_OTYPER_OT11;
}
```

De acordo com a Tabela 9 e Tabela 12 do *Datasheet*, a função alternativa para ambos os periféricos é AF1. O tipo de configuração de saída nos dois pinos é *push-pull*.

5. Vamos ainda definir a função de configuração do **pino PE9** para que ele seja um pino GPIO de saída *push-pull*, iniciando em nível baixo (que é o *default*), no escopo `/* USER CODE BEGIN 4 */`.

```
void PE9_PInit (void) {
      // Ativa GPIOE (PE9)
      RCC->AHB4ENR |= RCC_AHB4ENR_GPIOEEN_Msk;
      // PE9 como saida digital
      GPIOE->MODER &= ~(GPIO_MODER_MODE9_Msk);
      GPIOE->MODER |= GPIO_MODER_MODE9_0;
      GPIOE->OTYPER &= ~GPIO_OTYPER_OT9_Msk; // PE9 como push-pull
}
```

O pino será usado para mostrar o trem de pulsos gerado pelo TIM6 a ser configurado com a função de interrupções periódicas de 10µs.

6. Agora passamos para a configuração dos temporizadores, vamos começar com a configuração do temporizador básico, TIM6, para operar como um temporizador de

interrupção periódica (do inglês, *periodic interrupt timer* - PIT). Insira o seguinte código depois da definição da função **`PE9_PInit (void)`**.

```
void TIM6_PInit(void) {
	RCC->APB1LENR |= RCC_APB1LENR_TIM6EN; // Habilita clock de TIM6
	TIM6->EGR |= TIM_EGR_UG_Msk;     // atualizacao inicial dos registradores
	while (TIM6->EGR & TIM_EGR_UG);
	TIM6->CR1 &= ~TIM_CR1_CEN; // Desabilita o contador
	TIM6->PSC = 64-1; // Prescaler, assumindo clock de 64 MHz, timer a 1 MHz
	TIM6->ARR = 10-1; // Periodo do timer: 1/1MHz * 10 = 10us
	TIM6->CR1 &= ~TIM_CR1_UDIS; // Habilita a geracao do evento de atualizacao

	TIM6->DIER |= TIM_DIER_UIE; // Habilita a interrupcao
	// Habilitar interrupção TIM6 no NVIC
	NVIC_SetPriority(TIM6_DAC_IRQn, 1); // Define a prioridade da interrupção
	NVIC_EnableIRQ(TIM6_DAC_IRQn); // Habilita a interrupção TIM6

	TIM6->CR1 |= TIM_CR1_CEN; // Habilita o contador
}
```

Como a frequência do sinal de relógio é 64 MHz, configuramos o *Prescaler* para “64-1” e o *autoreload* para “10-1”. Além disso, habilitamos a interrupção de TIM6 no NVIC (número de vetor 54) e configuramos sua prioridade de atendimento em 1, como habilitamos a geração de eventos de interrupção no TIM6. Assim, teremos uma interrupção periódica a cada 10µs.

7. Vamos configurar o TIM1 para o seu canal 2 operar no modo *Output Compare*. Insira a seguinte função de configuração de TIM1 depois da definição da função **`TIM6_PInit(void)`**.

```
void TIM1C2_PInit(void) {
   // Habilitar o clock para o Timer TIM1
   RCC->APB2ENR |= RCC_APB2ENR_TIM1EN_Msk;
   TIM1->EGR |= TIM_EGR_UG_Msk;       // Atualização inicial dos registradores
   while (TIM1->EGR & TIM_EGR_UG);
   // Configurar PSC e ARR
   TIM1->PSC = 64-1; // Prescaler
   TIM1->ARR = 10-1; // Período
   // Configurar o modo de operacao do canal TIM1_CH2 para Output Compare:
   TIM1->CCMR1 &= ~(TIM_CCMR1_CC2S_1 | TIM_CCMR1_CC2S_0); // Modo Output Compare
   // Configurar o sinal de saída do canal TIM1_CH2
   TIM1->CCMR1 |= TIM_CCMR1_OC2M_1 | TIM_CCMR1_OC2M_0; // Modo Toggle
   // Habilitar a precarga do valor de comparacao
   TIM1->CCMR1 |= TIM_CCMR1_OC2PE;
   // Configurar o valor de comparação para 0
   TIM1->CCR2 = 0;
   // Habilitar a saida
   TIM1->BDTR |= TIM_BDTR_MOE; // Master Output Enable
   // Habilitar o canal TIM1_CH2
   TIM1->CCER |= TIM_CCER_CC2E;
   // Habilitar o Timer
```

```
    TIM1->CR1 |= TIM_CR1_CEN; // Iniciar a contagem
}
```

Em cada *match*, ou seja quando o valor do contador do TIM1 fique igual ao conteúdo do registrador de comparação `TIM1->CCR2`, o sinal é alternado (*toggle*). Ajustamos a frequência do relógio do temporizador TIM1 para 1MHz, atribuindo "64-1" ao *Prescaler*. Setamos "10-1" como contagem máxima no *autoreload*. O valor de comparação é setado em "10". Habilitamos também a pré-carga do valor de comparação, para que as modificações nos registradores de comparação sejam efetivadas apenas quando ocorrem eventos de atualização. Note que para o evento *match* ser reconhecido como um evento de interrupção, a máscara de interrupção do canal 2 precisa ser habilitada. Além disso, TIM1 precisa habilitar um *bit* de *Master Output Enable* para que as saídas dos canais sejam exteriorizadas nos pinos correspondentes. Por fim, o *timer* em si precisa ser habilitado para contar, através do *bit* CEN no registrador CR1.

Obs: A comparação do valor do contador com o registrador de "captura/comparação" (*Capture/Compare Register* ou CCR) é assíncrona, portanto aqui não existe o fator "-1" usado no *prescaler* e no *autoreload*). Isto vale tanto no OC como no IC e no PWM.

Entende por que os registradores são configurados de maneiras diferentes? Se não, sem problemas! Vamos explorar isso juntos mais adiante.

8. E adicionamos também, após a função `TIM1C2_PInit(void)`, a definição da função que configura o canal 4 do temporizador TIM2 para o modo de operação OC com a alternância do nível do sinal de saída em cada *match*.

```
void TIM2C4_PInit(void) {
  // Habilitar o clock para o Timer TIM2
  RCC->APB1LENR |= RCC_APB1LENR_TIM2EN_Msk;
  TIM2->EGR |= TIM_EGR_UG_Msk;       //Atualizacao inicial dos registradores
  while (TIM2->EGR & TIM_EGR_UG);
  // Configurar PSC e ARR
  TIM2->CR1 &= ~TIM_CR1_ARPE_Msk; //Desabilita precarga de autoreload
  TIM2->PSC = 64-1; // Prescaler
  TIM2->ARR = 65536-1; // Período
  // Configurar o modo de operacao do canal TIM2_CH4 para Output Compare:
  TIM2->CCMR2 &= ~TIM_CCMR2_CC4S_Msk; // Modo Output Compare
  // Configurar o sinal de saída do canal TIM1_CH2
  TIM2->CCMR2 &= ~TIM_CCMR2_OC4M_Msk;
  TIM2->CCMR2 |= TIM_CCMR2_OC4M_1 | TIM_CCMR2_OC4M_0; // Modo Toggle
  // Habilitar a precarga do valor de comparacao (pag 1551)
  TIM2->CCMR2 |= TIM_CCMR2_OC4PE;
  // Setar a polaridade
  TIM2->CCER &= ~TIM_CCER_CC4P;    // ativo alto
  // Configurar o valor de comparação para 10
  TIM2->CCR4 = 10;
  //Habilitar a interrupcao do canal 4
```

```
  TIM2->DIER |= TIM_DIER_CC4IE; // Habilita a interrupcao
  // Habilitar interrupção TIM1 no NVIC
  NVIC_SetPriority(TIM2_IRQn, 2); // Define a prioridade da interrupção
  NVIC_EnableIRQ(TIM2_IRQn); // Habilita a interrupção TIM2_IRQn
  // Habilitar o canal TIM1_CH2
  TIM2->CCER |= TIM_CCER_CC4E;
  // Habilitar o Timer
  TIM2->CR1 |= TIM_CR1_CEN; // Iniciar a contagem
}
```

E, habilitamos a interrupção de TIM2, com a prioridade de atendimento em 2, no NVIC (número de vetor 27) e a geração de eventos de interrupção do canal 4 no TIM2. Usaremos essas interrupções para modificar o conteúdo do registrador de comparação como veremos adiante. Para que essas atualizações não conflictem com as atualizações dos sinais de saída, é importante habilitar a pré-carga do registrador *auto-reload*. Lembre-se do papel importante dos registradores pré-carga e sombra na integridade de dados destacado no Roteiro 5?

9. Por fim, dentro da área `/* USER CODE BEGIN 2 */`, chame as funções:

```
PE9_PInit();
TIM6_PInit();
PE11_PInit();
TIM1C2_PInit();
PB11_PInit();
TIM2C4_PInit();
```

Nada será feito no *loop* infinito.

10. Precisa-se ainda implementar as definições das rotinas de serviço, `void TIM2_IRQHandler (void)` e `void TIM6_DAC_IRQHandler (void)`, predeclaradas em `Startup/startup_stm32h7a3xxq.s` para tratamento das interrupções. Abra o arquivo `Core/Src/stm32h7xx_it.c` e insira depois de `/* USER CODE BEGIN 1 */` a definição

```
void TIM6_DAC_IRQHandler (void)
{
	static uint8_t pinon = 0; // estado logico do pino
	if (TIM6->SR | TIM_SR_UIF) {
		//Alterna o estado logico do pino PE9
		TIM6->SR *= ~TIM_SR_UIF;  // limpa o bit de estado
		if(pinon) {
			pinon = 0;
			GPIOE->BSRR = GPIO_BSRR_BR9; // reset pino PE9
		} else {
			pinon = 1;
			GPIOE->BSRR = GPIO_BSRR_BS9; // set pino PE9
		}
	}
}
```

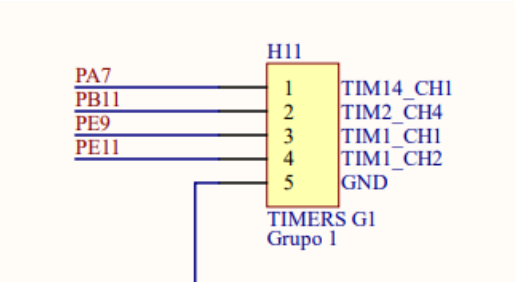
Note que é necessário limpar a *flag*, escrevendo 0 no *bit* de *Update Interrupt Flag* (veja no Manual de Referência que as *flags* deste registrador são limpas escrevendo-se o *bit* 0 (rc_w0) nos mesmos). Na sequência, inverte-se a variável de estado do pino e o nível lógico em PE9. Aqui vamos gerar uma onda quadrada no pino PE9, alternando o nível do sinal de saída em cada *match*.

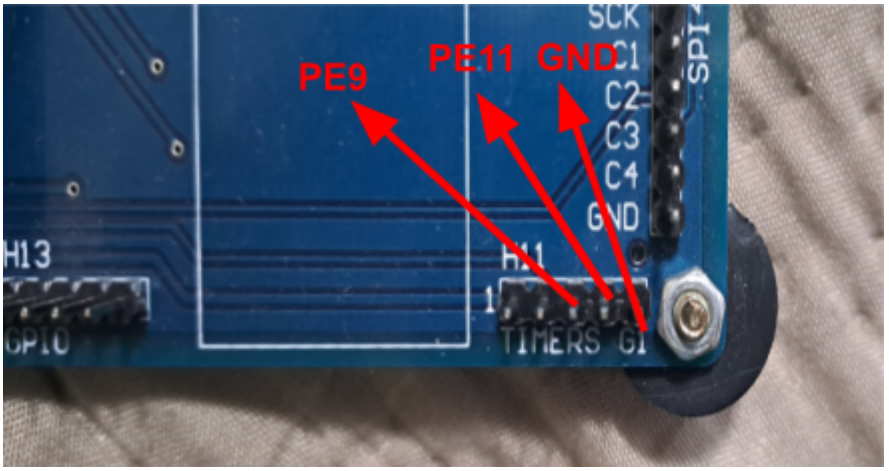
11, Em seguida, insira a definição da segunda rotina de serviço:

```
void TIM2_IRQHandler (void)
{
	static uint16_t cnt=0;
	if(TIM2->SR & TIM_SR_CC4IF) { //Flag interrupcao TIM1_CH2
		TIM2->SR &= ~TIM_SR_CC4IF; // Limpa flag
		if (TIM2->CCR4 == 10 && cnt < 3) {
			cnt++;
		} else if (TIM2->CCR4 == 10 && cnt == 3) {
			cnt = 0;
			TIM2->CCR4 = TIM2->ARR/4; // proxima transicao em 0
		} else if (TIM2->CCR4 == TIM2->ARR/4 && cnt < 3) {
			cnt++;
		} else if (TIM2->CCR4 == TIM2->ARR/4 && cnt == 3){
			cnt = 0;
			TIM2->CCR4 = 10; // proxima transicao em 5
		}
	}
}
```

A cada *match*, o *hardware* realiza o *toggle* do pino instantaneamente, e ativa uma interrupção (IRQ 28). Note que a interrupção tem uma latência, mas o *toggle* sempre ocorrerá no instante preciso. Inicialmente, a rotina de serviço verifica se ocorreu um evento de OC no canal 4 do TIM2. Se este for o caso, a *flag* é limpa segundo a política de limpeza rc_w0. Para demonstrar que o *toggle* se repita periodicamente em intervalos precisos para valores de distintos no registrador `TIM2->CCR4`, definimos 2 valores de comparação, 10 e `TIM2->ARR/4`. Esses valores são setados depois de 3 períodos de contagem máxima, seguindo a lógica programada e atualizados apenas quando ocorrem os eventos de atualização.

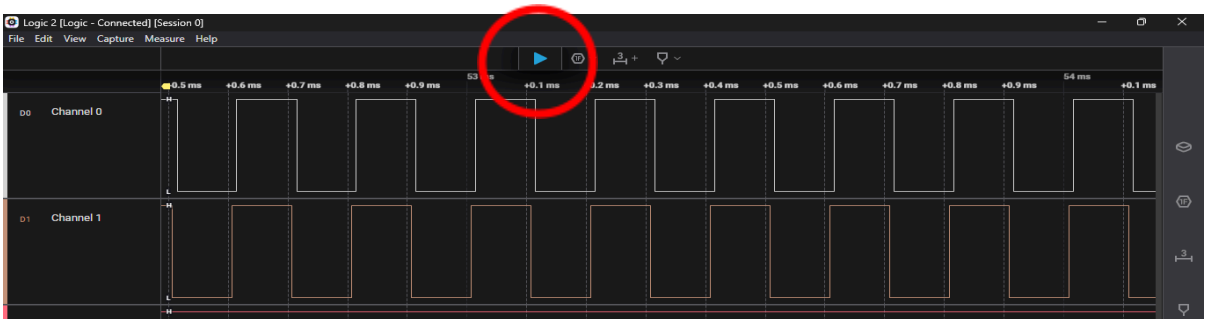
11. Realize um *Build* no código, verificando se não há erros. Transfira o código executável para o microcontrolador no modo *Debug*. Antes de realizar o *Debug*, vamos ligar o analisador lógico nos pinos de saída. Conecte o canal 2 do analisador lógico ao pino 2 do *Header* H11 (PB11), o canal 1 ao pino 4 (PE11) e canal 0 ao pino 3 do mesmo *Header* (PE9), além do pino GND do analisador ao pino 5 do mesmo *Header* (GND). As figuras abaixo mostram os pontos de conexão, sob duas distintas perspectivas.





Em seguida, conecte o analisador a uma porta USB do desktop e abra o aplicativo Logic 2.4.x.

11. Transfira o programa executável ao microcontrolador no modo *Debug*. Inicie a execução do programa. Clique na seta azul do aplicativo para inicializar a captura dos sinais nos pinos PB11, PE9 e PE11 pelo analisador lógico. Para párar a coleta, basta clicar no quadrado azul no mesmo local.



12. A frequência do sinal de relógio dos contadores dos três temporizadores, TIM6, TIM1 e TIM2, é 64MHz. Meça os períodos (larguras dos pulsos em nível alto) dos sinais capturados nos pinos e preencha a tabela abaixo

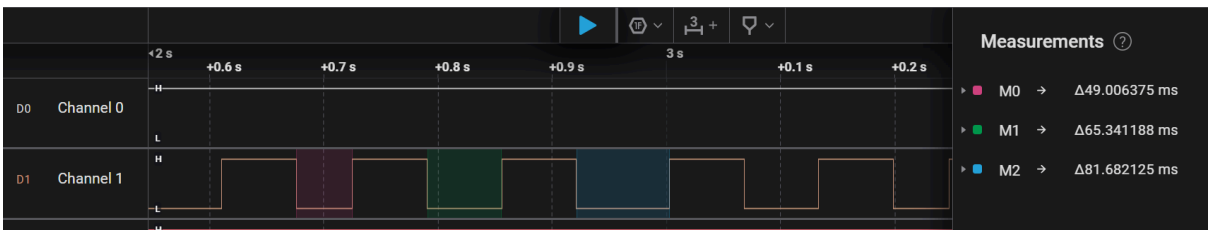
Pino	TIMx->PSC	TIMx->ARR	TIMx->CCRn	Período Estimado	Período Medido

Os valores observados (medidos) são condizentes com os esperados (estimados)? As larguras dos pulsos positivos (nível 1) dependem do valor de comparação configurados? Você saberia justificar a sua resposta? Se tiver dúvidas, continue os exercícios. A explicação será apresentada mais adiante.

13. Compare a configuração dos registradores dos temporizadores TIM1/TIM2 com TIM6. Qual(is) deles permite(m) configurar o modo de operação OC? Através de qual(is) registrador(es)?

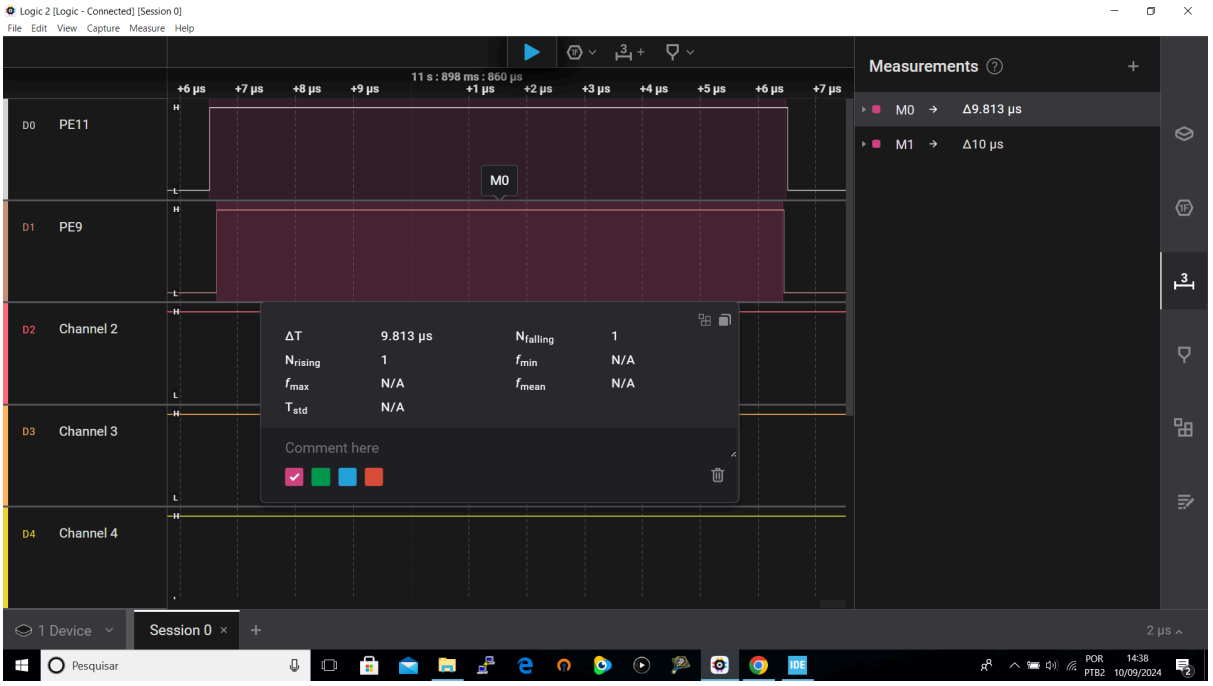
14. Na configuração do TIM2, a pré-carga do valor máximo (em inglês, *autoreload*) foi desabilitada, enquanto no TIM1, a pré-carga do valor de comparação (`TIM1->CCR2`) foi habilitada. O que é pré-carga de um registrador? O que é um evento de atualização? O que aconteceria se removêssemos essas instruções? Como você acha que o comportamento do sistema seria afetado? Não se preocupe se a resposta não vier à mente agora! Vamos desvendarmos juntos os segredos por trás dessas configurações mais adiante.

15. Você consegue identificar três pulsos em nível baixo com larguras diferentes na forma de onda capturada no pino PB11 (Canal 2)? Observe a figura abaixo e compare com suas medições. Agora, reflita: Os valores que você mediu concordam com os mostrados na imagem capturada que se segue? Qual parte do código pode estar gerando essa variação? Esses valores fazem sentido com a configuração programada? Se ainda não tem certeza, não se preocupe! As explicações mais adiante vão te ajudar a entender exatamente o que está acontecendo.



16. Agora vamos comparar o grau de precisão dos períodos dos sinais gerados de duas formas diferentes: o modo OC do canal `TIM1_CH2` (Canal 1) e as interrupções periódicas do temporizador `TIM6` (Canal 0). Sabemos que, em ambos os casos, o período do *timer* é de

10µs, o que significa que cada sinal deve permanecer 10µs em nível alto e 10µs em nível baixo, totalizando um período completo de 20µs.



Aumente a resolução (*zoom in*) da sua captura de sinal e meça os tempos em nível alto e baixo nos dois canais, como a imagem capturada. Compare os valores medidos com os apresentados na imagem capturada. Tente responder as seguintes perguntas: Qual dos dois sinais está mais próximo do valor esperado? Por quê há diferenças entre eles? O que pode estar influenciando essas variações? Se tiver dúvidas, vamos investigar juntos!

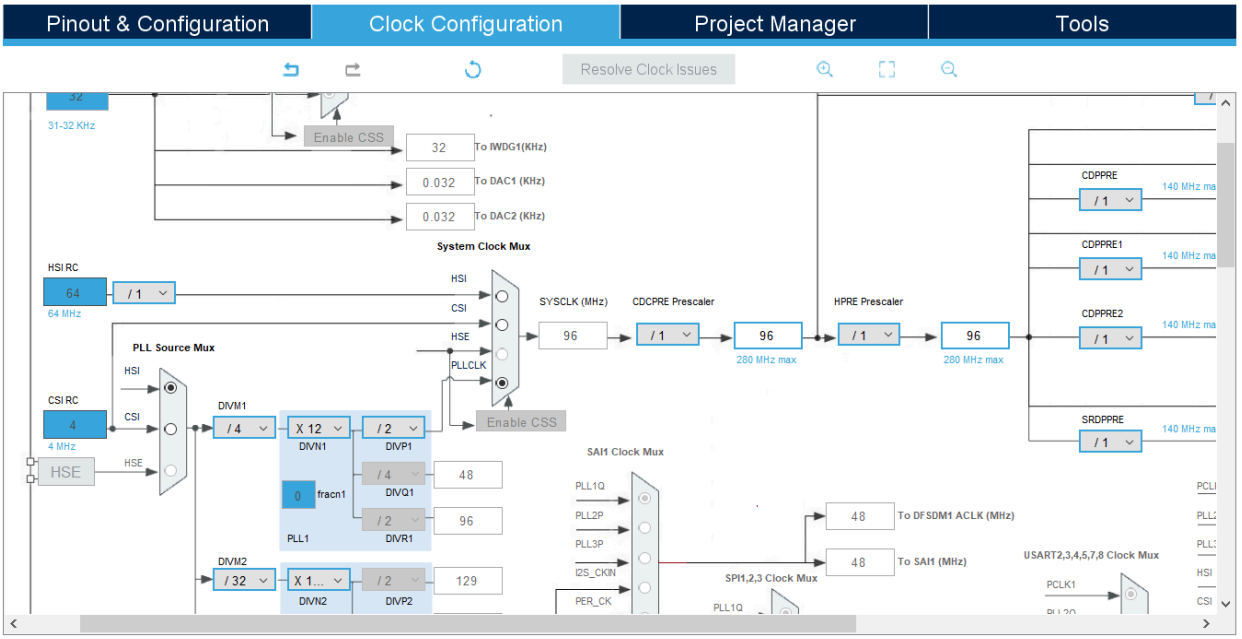
15. Compare as instruções usadas na configuração do canal TIM1_CH2 e do temporizador TIM6. Qual das duas abordagens envolve menos registradores adicionais para gerar resultados bem similares?

Projeto de *Input Capture*

Você já precisou medir intervalos de tempo com alta precisão? Em muitos projetos eletrônicos e sistemas embarcados, é essencial capturar a diferença de tempo entre eventos assíncronos, como a detecção de bordas de subida e descida de um sinal. Para isso, utilizaremos a funcionalidade *Input Capture* (IC), uma ferramenta que mede com exatidão esses intervalos. No projeto em questão, vamos gerar uma onda assimétrica usando laços simples e GPIO, medir os tempos em nível alto e baixo dessa onda, detectando as bordas de subida e descida, e comparar os valores medidos com os obtidos por um analisador lógico. O desafio está em como garantir que nossa medição seja precisa e eficiente? Será que conseguimos reduzir ao máximo os erros? Vamos descobrir com o seguinte projeto que mede

o intervalo entre dois instantes assíncronos, definidos por um par de bordas de descida/subida?

1. Crie o projeto usando o *Cube*, com o nome "Input_Capture", **sem a inicialização dos periféricos**. Ative o módulo *Debug* como "Serial Wire". Entre no editor gráfico de "Clock Configurtion" e configure a frequência do sistema em 96 MHz em uma das janelas embaixo dos quais tem o texto "280 MHz max."



2. Gere o código de configuração do sistema de sinais de relógio. Abra o arquivo `Core/Src/main.c`.

3. Comente (Remova) a declaração, a definição e a chamada da função `MX_GPIO_Init(void)`. Adecione a declaração das seguintes funções de configuração dos periféricos GPIOA, GPIOB e TIM1 e de interface entre o arquivo `main.c` e `stm32h7xx_it.c` no escopo `/* USER CODE BEGIN PFP */`:

```
void PA8_PInit (void);
void PE11_PInit (void);
void TIM1C2_PInit(void);
void le_dados_captura (uint16_t *arg1, uint16_t *arg2, uint16_t *arg3, uint8_t *arg4);
void set_last_edge (uint8_t i);
```

Observação: Evitar variáveis globais e utilizar a passagem de valores via funções são práticas essenciais para o desenvolvimento de *software* robusto, manutenível e reutilizável. Essas técnicas promovem a modularidade, o encapsulamento e facilitam a depuração e o teste do código.

4. Vamos inserir a definição da função de configuração do pino PA8 para o modo de operação GPIO com o tipo de saída *push-pull* no escopo `/* USER CODE BEGIN 4 */`

```
void PA8_PInit (void) {
      // Ativa GPIOE (PE9)
      RCC->AHB4ENR |= RCC_AHB4ENR_GPIOAEN_Msk;
      // PE9 como saida digital
      GPIOA->MODER &= ~(GPIO_MODER_MODE8_Msk);
      GPIOA->MODER |= GPIO_MODER_MODE8_0;
      GPIOA->OTYPER &= ~GPIO_OTYPER_OT8_Msk; // PA8 como push-pull
}
```

O pino PA8 servirá como nosso gerador de onda assimétrica, alimentando o pino PE11.

5. O pino PE11 é multiplexado para o canal TIM1_CH2 configurado no modo de operação IC. Insira após a definiçao de `PA8_PInit (void) a seguinte função`

```
void PE11_PInit (void) {
   // Ativa GPIOE (PE11)
   RCC->AHB4ENR |= RCC_AHB4ENR_GPIOEEN_Msk;
   // Configurar PE11 como função alternativa
   GPIOE->MODER &= ~GPIO_MODER_MODE11_Msk;
   GPIOE->MODER |= GPIO_MODER_MODE11_1;
   // Selecionar AF1 para PE11
   GPIOE->AFR[1] &= ~(0xF << GPIO_AFRH_AFSEL11_Pos);
   GPIOE->AFR[1] |= (1 << GPIO_AFRH_AFSEL11_Pos);
}
```

Observe que, servindo agora como pino de entrada, a sua configuração é ligeiramente diferente da configuração do projeto anterior.

6. Por fim, precisamos definir logo a seguir a função de configuração do canal TIM1_CH2 para operar no modo de operação IC, sensível às duas bordas (de subida e descida)

```
void TIM1C2_PInit(void) {
  // Habilitar o clock para o Timer TIM1
  RCC->APB2ENR |= RCC_APB2ENR_TIM1EN_Msk;
  TIM1->EGR |= TIM_EGR_UG_Msk;    //Atualizacao inicial dos regostradores
  while (TIM1->EGR & TIM_EGR_UG);
  // Configurar PSC e ARR
  TIM1->CR1 &= ~TIM_CR1_ARPE_Msk; //Desabilita precarga de autoreload
  TIM1->PSC = 96-1; // Prescaler
  TIM1->ARR = 65536-1; // Período
  // Configurar o modo de operacao do canal TIM1_CH2 para Input:
  TIM1->CCMR1 &= ~(TIM_CCMR1_CC2S_1 | TIM_CCMR1_CC2S_0);
  TIM1->CCMR1 |= TIM_CCMR1_CC2S_0; // PE11 mapeado na entrada TI1 (página 1522)
  // Configurar a detecção de ambas bordas
  TIM1->CCER |= (TIM_CCER_CC2NP | TIM_CCER_CC2P); // Detecção de ambas bordas
  //Habilitar a interrupcao do canal 2
  TIM1->DIER |= TIM_DIER_CC2IE; // Habilita a interrupcao
  // Habilitar interrupção TIM1 no NVIC
  NVIC_SetPriority(TIM1_CC_IRQn, 1); // Define a prioridade da interrupção
  NVIC_EnableIRQ(TIM1_CC_IRQn); // Habilita a interrupção TIM1_CC_IRQn
```

```
  // Habilitar o canal TIM1_CH2
  TIM1->CCER |= TIM_CCER_CC2E;
  // Habilitar o Timer
  TIM1->CR1 |= TIM_CR1_CEN; // Iniciar a contagem
}
```

Para configurar o *timer* TIM1 com um clock de 96 MHz, iniciamos definindo um prescaler de 96, que atua como um divisor da frequência do *clock*, seguido pela configuração de um valor máximo de contagem, conhecido como *auto-reload*, para 65536. Essa combinação de configurações garante que cada incremento no contador `TIM1->CNT` corresponda a um intervalo de tempo de 1µs. Adicionalmente, habilitamos a interrupção do TIM1 no NVIC, utilizando o vetor de interrupção 27, e asseguramos a geração de eventos de interrupção no canal `TIM1_CH2`. Por fim, para controlar a prioridade de atendimento da interrupção, definimos o nível de prioridade como 1.

7. Vamos abrir agora o arquivo `Core/Src/stm32h7xx_it.c` para incluir a rotina de serviço que calcula os intervalos de tempo entre a borda de subida e descida. Para isso, vamos declarar algumas variáveis na área `/* USER CODE BEGIN 1 */`

```
uint16_t last_capture=0;
uint16_t duration_high=0;
uint16_t duration_low=0;
uint8_t last_edge;
uint16_t current_capture;
int32_t tmp;
```

antes de inserir a rotina de serviço

```
void TIM1_CC_IRQHandler (void)
{
	if(TIM1->SR & TIM_SR_CC2IF) { //Flag interrupcao TIM1_CH2
		current_capture = (uint16_t)TIM1->CCR2; // Flag e limpo na leitura do
CCRx
		if (last_capture == 0) {
			last_capture = current_capture;
		}
		tmp = current_capture - last_capture;
		if (last_edge == 0) {
			// Borda de subida (fim de nível baixo, início de nível alto)
			if (tmp >= 0) {
				duration_low = (uint16_t) tmp;
			} else {
				duration_low = (uint16_t)(TIM1->ARR + 1 + tmp);
			}
			last_edge = 1; // Próxima borda será de descida
		} else {
			// Borda de descida (fim de nível alto, início de nível
baixo
```

```
                    if (tmp >= 0) {
                        duration_high = (uint16_t) tmp;
                    } else {
                        duration_high = (uint16_t)(TIM1->ARR + 1 + tmp);
                    }
                    last_edge = 0; // Próxima borda será de subida
                }
                // Atualize o último valor de captura
                last_capture = current_capture;
        }
}
```

Essencialmente, a rotina de serviço calcula, na captura de uma borda, a quantidade de contagens entre duas capturas sucessivas, diferenciando-a entre pulsos de nível alto (`duration_high`) e pulsos de nível baixo (`duration_low`). Inicialmente, ela verifica se o evento foi gerado em `TIM1_CH2` (PE11). Em seguida, usa a função para ler o valor presente no registrador `TIM1->CCR2`. Na sequência, dependendo da última borda, calcula o valor do tempo em nível alto ou em nível baixo, atualizando a borda e o último valor lido.

É importante ressaltar que quando ocorre um evento de IC no pino de um canal do *timer*, o valor do contador do *timer* é imediatamente (via *hardware*) copiado no registrador CCR do canal correspondente. Assim, o registrador CCR guarda a informação do momento exato em que o evento ocorreu. Mesmo que haja a latência de interrupção, o que será usado nos cálculos não é o valor atual do contador (que pode ter mudado durante a latência), mas o valor de CCR, que corresponde ao valor de contagem **no momento do evento de IC**. Isto garante a alta precisão nas medições.

Note que o registrador CCR2 é de 32 *bits*, apesar de o contador ser de 16 *bits*. Assim, é realizado um *cast* para 16 *bits* ao se copiar o valor na variável `current_capture`. Se o valor é lido como de 16 *bits* sem sinal, quando ocorrer um *overflow* no contador, o mesmo *overflow* ocorrerá no cálculo da diferença entre a captura atual e a anterior. Assim, o valor calculado será sempre o número de pulsos contabilizados entre eventos sucessivos. Como exemplo, imagine que o valor lido é 60000 e o valor anterior é 50000. A variável "duration_*" irá guardar o valor 10000, ou seja, 10ms. Imagine que mais 10000 pulsos sejam contados antes do próximo evento. Neste caso, o contador irá contar mais 10000 pulsos totalizando 700000 = 0b1 0001 0001 0111 0000 pulsos. Como o contador só tem 16 *bits*, somente 0b0001 0001 0111 0000=4464 é considerado. Isso equivale a contar 5536 pulsos e voltar a zero (*overflow*), contando mais 4464 pulsos antes do evento. Assim, no próximo cálculo, a variável "duration_*" será 4464 - 60000 = -55536 que, por *rollover*, deve ser ajustado somando 65536 que resulta em 10000.

Observe ainda que neste caso, não inserimos a instrução `TIM1->SR &= ~TIM_SR_CC2IF;` para limpar a *flag*, pois o acesso de leitura ao registrador `TIM1->CCR2` limpa automaticamente a *flag*.

8. Para a passagem dos dados entre os arquivos `main.c` e `stm32h7xx_it.c`, vamos implementar mais duas funções:

```
void le_dados_captura (uint16_t *arg1, uint16_t *arg2, uint16_t *arg3, uint8_t
*arg4) {
	*arg1 = last_capture;;
	*arg2 = duration_high;
	*arg3 = duration_low;
	*arg4 = last_edge;
}
void set_last_edge (uint8_t i) {
	last_edge = i;
}
```

e declarar os seus protótipos tanto em `main.c` (área `/* USER CODE BEGIN PFP */`) quanto em `stm32h7xx_it.c` (também na área `/* USER CODE BEGIN PFP */`).

```
void le_dados_captura (uint16_t *arg1, uint16_t *arg2, uint16_t *arg3, uint8_t
*arg4);
void set_last_edge (uint8_t i);
```

9. Agora vamos declarar as variáveis locais de `main()`, uma variável para definir o estado inicial do sinal lido em PE11 e outra variável para gerar *delays* (não muito exatos) através de laços. Na área `/* USER CODE BEGIN 2 */`, declare as variáveis:

```
GPIO_PinState pin_state;
uint32_t i;
```

10. Vamos incluir na área `/* USER CODE BEGIN 2 */` da função `main` as chamadas das funções de configuração dos periféricos

```
 PA8_PInit();
 PE11_PInit();
 TIM1C2_PInit();
```

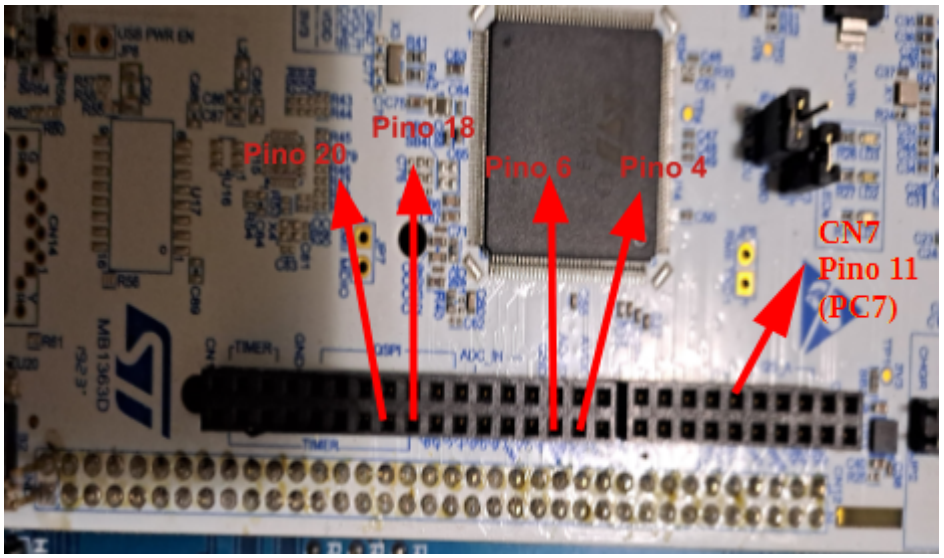
e instruções de detecção do nível inicial do sinal no pino de entrada PE11

```
 //Estado inicial de PE11
 pin_state = GPIOE->IDR & GPIO_IDR_ID11_Msk;
 if (pin_state == GPIO_PIN_SET) {
	set_last_edge (1); // Inicia em nível alto, então a primeira borda será
de descida
 } else {
	set_last_edge (0); // Inicia em nível baixo, então a primeira borda será
de subida
 }
```

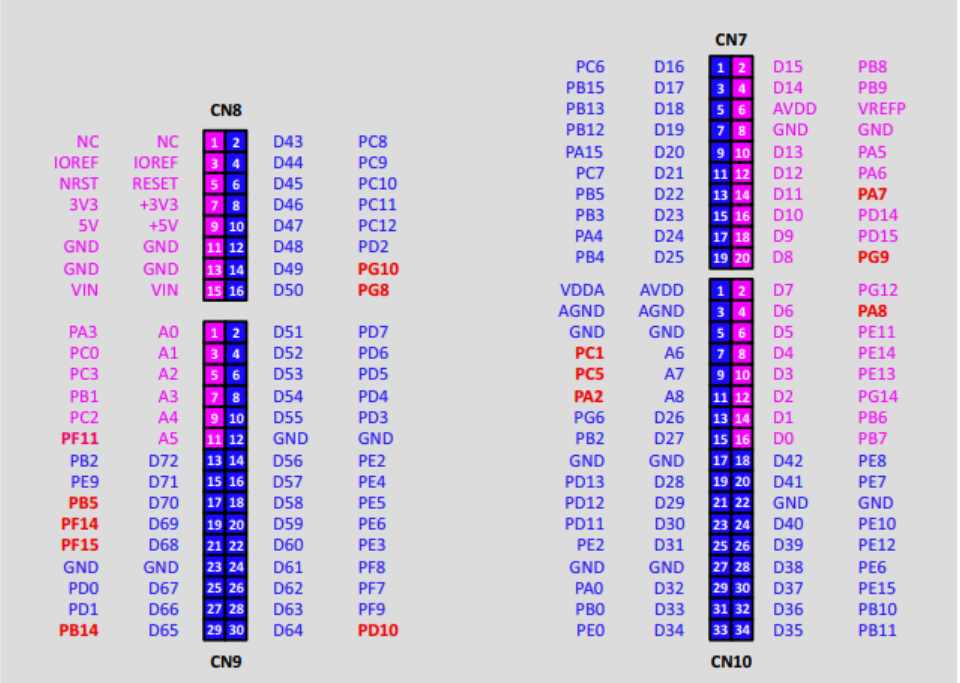
11. Para gerar um trem de pulsos no pino PA8 que vai alimentar o pino PE11 de entrada do canal TIM1_CH2, vamos incluir o seguinte bloco de instruções na área `/* USER CODE BEGIN 3 */`, dentro do laço infinito

```
for(i = 0; i < 13000; i++){}
    GPIOA->BSRR = GPIO_BSRR_BS8;
for(i = 0; i < 8000; i++){}
    GPIOA->BSRR = GPIO_BSRR_BR8;
//Atualiza os dados de captura
uint16_t arg1, arg2, arg3;
uint8_t arg4;
le_dados_captura (&arg1, &arg2, &arg3, &arg4);
```

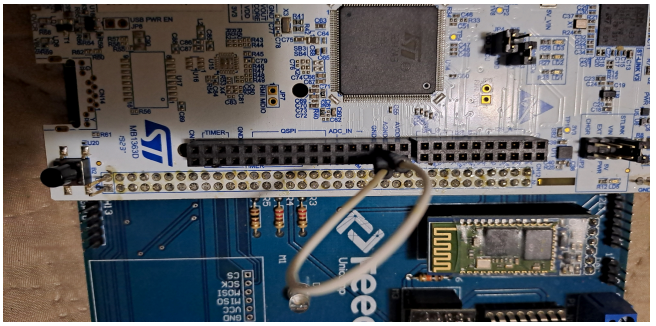
12. Realize um *Build* e verifique se não há erros. Antes de executar o programa, precisamos interligar a saída da onda (PA8) à entrada `TIM1_CH2` (PE11). Para isso, vamos usar os *headers* fêmea da placa NUCLEO, conforme mostra a figura a seguir. **Obs: Esta figura serve de referência para este projeto e para os próximos.**



Existem 4 conectores pretos de linha dupla (conectores Zio) organizados em duas fileiras na placa. O conector embaixo à direita é o CN10, e nele podemos acessar PA8 (pino 4) e PE11 (pino 6). Note que PE11 deste *header* e o PE11 do *header* H11 correspondem a dois acessos distintos ao pino físico PE11 do microcontrolador.

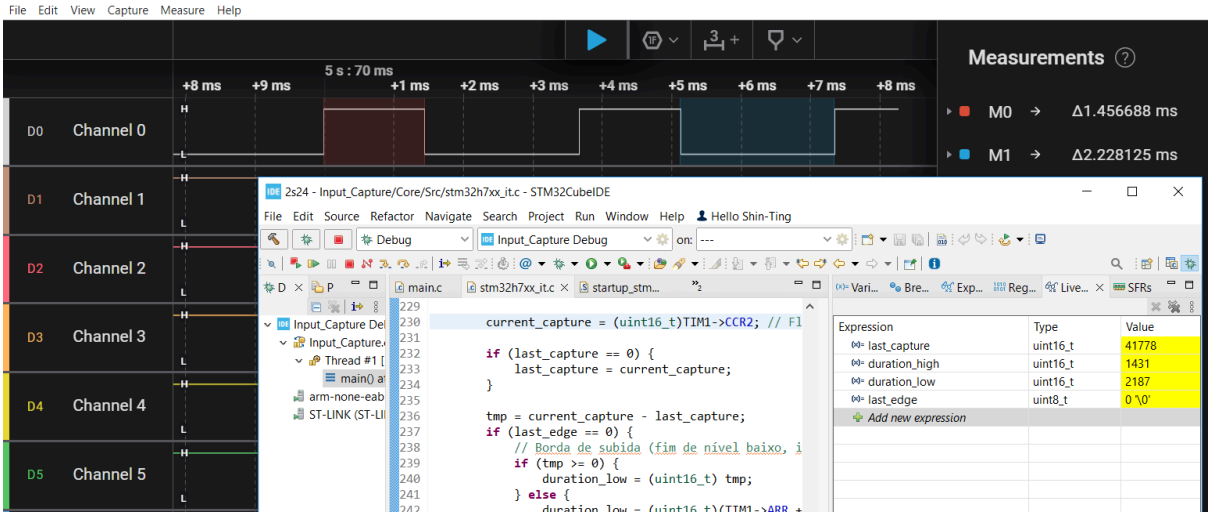


Usando um fio de conexão, interligue os pinos 4 e 6 de CN10, como indicado na figura. Para visualizar a onda gerada no pino PA8, basta manter a conexão de PE11 e GND ao analisador lógico através do *header* H11 da placa de expansão.



13. Transfira o programa executável ao microcontrolador no modo *Debug*. Inicie a execução do programa, realizando uma captura do sinal no PE11 pelo analisador lógico. Meça os tempos alto e baixo no canal do analisador em que o pino PE11está conectado.

Os valores esperados devem estar próximos dos exibidos na imagem da tela capturada. Compare as larguras dos pulsos alto e baixo medidos pelo analisador lógico com os valores calculados pelo programa, expressos em unidades de 10 µs. Na aba `Expression`, nas três últimas linhas, estão os valores calculados a partir do conteúdo lido do contador `TIM1_CNT` ao entrar na ISR, em vez dos valores capturados durante as transições. Note que, apesar da diferença ser mínima, há uma variação entre os valores obtidos por essas duas abordagens.



14. Dê uma pausa (“Pause’) na execução e veja os valores das variáveis de tempo alto e baixo (use a aba “Expressions”). **Lembre que, como o *clock* do *Timer* 1 é de 1MHz, os valores calculados nas variáveis já estão em µs.** Compare com os valores medidos no analisador. Lembre que, da mesma forma que no OC, a latência do *hardware* é imperceptível. Os recursos de OC e IC permitem que os eventos sejam executados ou detectados sem atrasos de *software*.

	Valor calculado (PE11)	Valor medido (PA8)
Nível Alto		
Nível Baixo		

15. Neste projeto, configuramos a frequência-base do sistema para 96 MHz ao utilizar o modo Input Capture (IC). Mas será que isso afeta a resolução da captura? Se aumentarmos a frequência, conseguimos medir intervalos de tempo menores com mais exatidão? Ou existe um limite para essa relação? O que você acha? Se ainda não sabe a resposta, não se preocupe! Vamos explorar essa questão mais adiante e entender como a frequência impacta a precisão da captura.

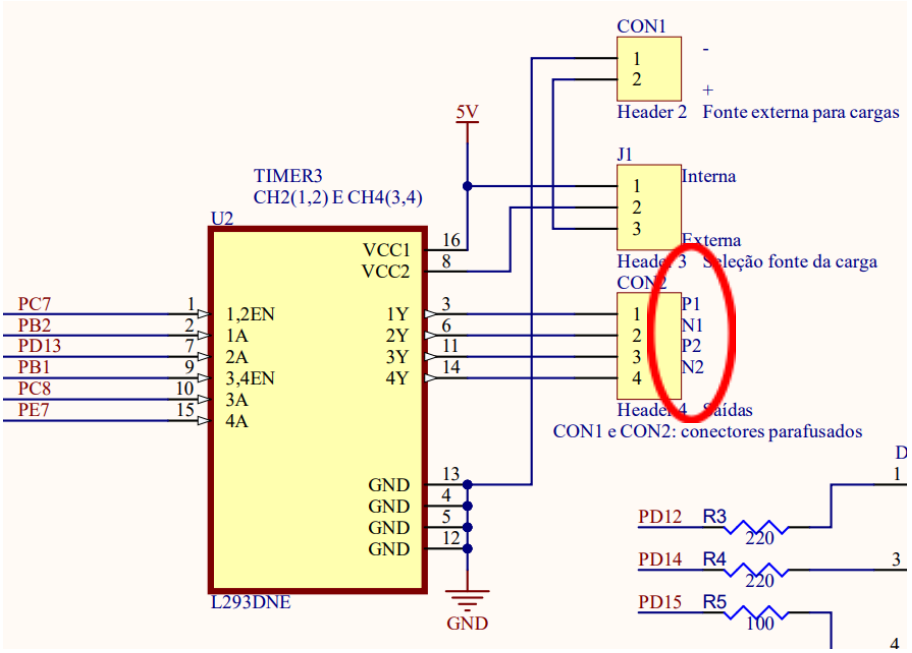
Projeto de *PWM*

Controlar a potência entregue a uma carga é um desafio em sistemas eletrônicos e automotivos. Mas como fazer isso de forma eficiente, sem desperdiçar energia ou sobrecarregar componentes? Imagine tentar controlar a velocidade de um motor DC ajustando diretamente a tensão aplicada. Esse método pode ser ineficiente, gerar aquecimento excessivo e exigir circuitos complexos. Agora, e se houvesse uma solução que permitisse um controle preciso e flexível com mínima perda de energia?

Neste projeto, você descobrirá como a modulação por largura de pulso (em inglês, *Pulse Width Modulation* – PWM) simplifica esse controle. Vamos explorar como um temporizador dedicado pode gerar um sinal PWM para regular a velocidade de um motor DC de 5V. Ao final, você verá como o uso de um PWM "pronto" reduz significativamente a complexidade do sistema e melhora sua eficiência.

1. Crie um projeto novo usando o *Cube*, com o nome "PWM", com a opção **"Initialize all peripherals with their default!" desabilitada**. Ative o módulo *Debug* como "Serial Wire". Entre na aba "Clock Configuration" e modifique a frequência do *clock* do sistema para **96MHz**. Gere o código e abra o arquivo Core/Src/`main.c`. Comente (Remova) a declaração, a definição e a chamada da função `MX_GPIO_Init(void)`.

O projeto consiste em gerar um sinal PWM no canal TIM3_CH2 para controlar a velocidade de um motor DC de 5 V, conectados nos pinos P1 e N1 mostrados no excerto do esquemático da Placa de Expansão do NUCLEO.



2. Vamos configurar o *timer* e o canal a ser utilizado com a função `PWM_PInit`. Os parâmetros de configuração são o modo de operação PWM para o canal `TIM3_CH2`, divisor (*prescaler*) em 9600, contagem máxima do contador (*autoreload)* em 100. Para isso, incluimos o protótipo da função no escopo `/* USER CODE BEGIN PFP */`

```
void PWM_PInit(void);
```

e a sua definição no escopo `/* USER CODE BEGIN 4 */`

```
/**
* @brief: Inicializacao do PWM
*/
void PWM_PInit(void) {
   // Habilitar o clock para o Timer 3
   RCC->APB1LENR |= RCC_APB1LENR_TIM3EN_Msk;
```

```
    TIM3->EGR |= TIM_EGR_UG_Msk;
    while (TIM3->EGR & TIM_EGR_UG);
    // Configurar PSC e ARR
    TIM3->PSC = 9600-1; // Prescaler
    TIM3->ARR = 100-1; // Período
    TIM3->CCR2 = 0; // Duty cycle 0%
    // Configuração do modo PWM
    TIM3->CCMR1 &= ~(TIM_CCMR1_CC2S_Msk); // Direcao do canal Output
    TIM3->CCMR1 &= ~(TIM_CCMR1_OC2M_Msk); // Modo PWM 1
    TIM3->CCMR1 |= (TIM_CCMR1_OC2M_1 |
                    TIM_CCMR1_OC2M_2 |
                    TIM_CCMR1_OC2M_3); // Modo PWM 1 assimetrico
     TIM3->CCMR1 |= TIM_CCMR1_OC2PE; // Habilita preload para o canal
2
    TIM3->CCMR1 &= ~TIM_CCMR1_OC2FE; // Desabilita modo rapido
    //Direcao de contagem
    TIM3->CR1 &= ~TIM_CR1_DIR_Msk;
}
```

3. Para controlar o motor, utilizaremos uma Ponte-H, um circuito eletrônico que permite controlar a direção e a velocidade de motores DC. Nossa placa de expansão possui dois circuitos integrados com Pontes-H, e utilizaremos a Ponte-H número 1. O controle da velocidade do motor será feito através de um sinal PWM gerado no pino PC7 do microcontrolador. Este sinal modulará a potência aplicada ao motor, permitindo um controle preciso da velocidade. Além da velocidade, a direção do motor é controlada pela polaridade da tensão aplicada, que é definida por duas entradas digitais da Ponte-H. Para garantir o funcionamento correto, estas entradas devem estar sempre em estados lógicos opostos: uma em nível alto e a outra em nível baixo. Na Ponte-H 1, estas entradas estão conectadas aos pinos PB2 e PD13 do microcontrolador. Para configurar corretamente estes pinos, implementaremos a função `GPIO_PInit`. Esta função será responsável por inicializar os pinos PC7, PB2 e PD13 com as configurações necessárias para o controle da direção do motor. O protótipo desta função será inserido na área `/* USER CODE BEGIN PFP */` do arquivo `main.c`

```
void GPIO_PInit(void);
```

e a sua definição no escopo `/* USER CODE BEGIN 4 */`

```
/**
* @brief Inicializacao de PA7
*/
void GPIO_PInit(void) {
    // Habilitar o clock para o GPIO
    RCC->AHB4ENR |= (RCC_AHB4ENR_GPIOBEN |
            RCC_AHB4ENR_GPIOCEN |
            RCC_AHB4ENR_GPIODEN );
```

```
    // Configurar o pino correspondente ao TIM3_CH2 (PC7)
    GPIOC->MODER &= ~(GPIO_MODER_MODE7_Msk); // Limpar modos
    GPIOC->MODER |= GPIO_MODER_MODE7_1; // Modo alternativo
    // Selecionar a função alternativa para TIM3_CH2 (AF2)
    GPIOC->AFR[0] &= ~(GPIO_AFRL_AFSEL7); // AF2 para PC7
    GPIOC->AFR[0] |= (GPIO_AFRL_AFSEL7_1); // AF2 para PC7
    // Configurar o pino correspondente a ponte H
    GPIOB->MODER &= ~(GPIO_MODER_MODE2_Msk); // Limpar modos
    GPIOB->MODER |= GPIO_MODER_MODE2_0; // GP Output
    GPIOB->OTYPER &= ~GPIO_OTYPER_OT2_Msk; // Push-pull
    GPIOD->MODER &= ~(GPIO_MODER_MODE13_Msk); // Limpar modos
    GPIOD->MODER |= GPIO_MODER_MODE13_0; // GP Output
    GPIOD->OTYPER &= ~GPIO_OTYPER_OT13_Msk; // Push-pull
  }
```

Os pinos de controle da direção são configurado como GPIO do tipo de saída *push-pull*, enquanto os pinos de sinais PWM são multiplexados para o canal `TIM3_CH2`, assumindo uma função alternativa.

4. Adicionamos ainda na função `GPIO_PInit` a configuração do pino PC13 em que o botão azul da placa NUCLEO está conectado. É um pino de entrada no modo GPIO.

```
    // Configurar o botao azul
    // Configurar o pino correspondente a ponte H
    GPIOC->MODER &= ~(GPIO_MODER_MODE13_Msk); // Limpar modos
```

5. Na área `/* USER CODE BEGIN PFP */`, vamos declarar o protótipo da função que vamos criar para iniciar a contagem no *timer*:

```
void Iniciar(void);
```

Inserimos a sua definição na área `/* USER CODE BEGIN 4 */`

```
void Iniciar(void) {
    GPIOB->BSRR = GPIO_BSRR_BS2_Msk; // P1 da ponte H em nivel alto
    GPIOD->BSRR = GPIO_BSRR_BR13_Msk; // N1 da ponte em nivel baixo
    TIM3->CCER |= TIM_CCER_CC2E_Msk; // Habilitar saída do canal 2
    // Habilitar o Timer
    TIM3->CR1 |= TIM_CR1_CEN_Msk; // Iniciar o timer
}
```

6. Vamos também declarar as variáveis dentro do contexto da função `main`. Na área `/* USER CODE BEGIN 1 */`, inserimos

```
uint8_t increase_motor = 1; // 1 - cresce velocidade; 0 - reduz velocidade
uint8_t pwm_motor = 0; // Valor PWM para o motor
uint8_t dir_motor = 0; // Sentido de rotacao
```

7. Agora vamos iniciar o *timer* e seu canal. Na área `/* USER CODE BEGIN 2 */`, chame a função de inicialização:

```
PWM_PInit();
GPIO_PInit();
Iniciar();
```

Inicialmente, colocamos o pino P1 da ponte (PB2) em nível alto, mantendo o pino N1 no nível baixo padrão na inicialização. Assim, o motor pode girar em um sentido, com velocidade dependente do ciclo de trabalho do PWM gerado pelo `TIM3_CH2`.

8. No *loop* infinito vamos fazer um *polling* no botão azul da placa (PC13). A cada apertar do botão, o *Duty Cycle* do PWM do motor aumenta em 10%. Ao chegar a 100%, o DC passa a diminuir 10% a cada apertar do botão. Ao retornar a 0%, o sentido de rotação é invertido e volta-se a aumentar o DC em 10%. Após a linha `/* USER CODE BEGIN 3 */`, escreva o código:

```
while(!(GPIOC->IDR & GPIO_IDR_ID13));
if(increase_motor) {
	pwm_motor += 10;
	if(pwm_motor == 100) {
		increase_motor = 0;
	}
} else {
	pwm_motor -= 10;
	if(pwm_motor == 0) {
		increase_motor = 1;
		TIM3->CCR2 = 0;
		if(dir_motor) {
			dir_motor = 0;
			GPIOB->BSRR = GPIO_BSRR_BS2; // P1 da ponte em nivel alto
			GPIOD->BSRR = GPIO_BSRR_BR13; // N1 da ponte em nivel baixo
		} else {
			dir_motor = 1;
			GPIOB->BSRR = GPIO_BSRR_BR2; // P1 da ponte em nivel baixo
			GPIOD->BSRR = GPIO_BSRR_BS13; // N1 da ponte em nivel alto
		}
	}
}
TIM3->CCR2 = pwm_motor;
while(GPIOC->IDR & GPIO_IDR_ID13);
```

Note que a maior parte do código envolve o *polling* do botão e o cálculo do novo valor de DC, com a eventual mudança entre aumento ou redução do DC, além da mudança de sentido de rotação. O detalhe importante é que o novo valor de DC é guardado na variável "pwm_motor".

9. Realize o *Build*. Conecte o motor DC nos terminais "P1" e "N1" do conector CON2. Transfira o código executável para o microcontrolador e inicie a execução do programa. Pressione o botão azul várias vezes e veja o comportamento do motor.

10. Conecte o analisador lógico no pino PC7 do conector CN7 do NUCLEO (que corresponde ao PWM do motor). Veja a figura usada no projeto de *Input Capture*. Pressione o botão até uma velocidade intermediária no motor. Faça uma captura no analisador, registre o valor do `TIM3_CCR2` e meça o período total e o tempo em nível alto, calculando o *Duty Cycle*. Compare com os valores calculados em função dos parâmetros do *Timer* 3. Faça isto para 3 valores de *Duty Cycle*.

15. Faça uma pausa no programa e mude manualmente o valor no registrador `TIM3_CCR2` para que o *Duty Cycle* seja de 30% a mais e 30% a menos. Retorne a execução do programa e faça nova captura com o analisador lógico, verificando se o DC foi modificado corretamente. Há alguma relação entre o ciclo de trabalho e a velocidade do motor?

16. Modifique o *preescaler* do Timer 3 (`TIM3->PSC`) para obter um período PWM cinco vezes maior (defina `TIM3->PSC` para 48000, frequência em 20Hz) e cinco vezes menor (defina `TIM3->PSC` para 1920, frequência em 500z). Mantendo o ciclo de trabalho constante (mesma potência média), analise se há alguma relação entre a frequência do sinal PWM e o momento em que o motor começa a girar. Com base na teoria de funcionamento de motores DC, como você interpreta o comportamento observado?

17. Neste e nos dois projetos anteriores, tivemos oportunidade de configurar os canais dos temporizadores para 3 modos de operação mais aplicados, IC, OC e PWM. Mas você já parou para observar o padrão que se repete em todas essas configurações? Embora cada modo de operação tenha suas particularidades, a base da configuração permanece a mesma! Apenas a especificação do modo de operação se altera, como se estivéssemos trocando as lentes de uma câmera para capturar diferentes perspectivas do mesmo cenário. Nesta perspectiva, quais registradores permitem a distinção entre os modos IC, OC e PWM? Quais *bits* específicos são os responsáveis?

Projeto de Relógio Externo

Em sistemas embarcados, a necessidade de contar eventos com precisão é uma demanda comum em diversas aplicações, como medição de fluxo em sensores industriais,

monitoramento de pulsos em encoders e controle de motores. No entanto, implementar essa contagem via *software* pode ser ineficiente, especialmente em sistemas que precisam responder rapidamente a outros eventos críticos. Uma solução é o uso de um contador de eventos externos dedicado, capaz de realizar a contagem automaticamente e com alta precisão, sem sobrecarregar o processador principal. Além de simplificar a implementação, esse método reduz a latência e melhora a confiabilidade do sistema, pois evita a necessidade de interrupções constantes ou *polling* intensivo. Alguns desses contadores são equipados de filtros, que ajudam a atenuar interferências e ruídos nos sinais de entrada, para garantir que as contagens permaneçam exatas mesmo na presença de interferências.

Neste projeto, implementaremos um contador de eventos externos usando o temporizador TIM1. O pino PE7 receberá um sinal digital externo. Esse pino é especificamente definido para a função de *trigger* de *clock* externo para o contador integrado ao TIM1, garantindo a contagem apenas de sinais válidos, ideal para aplicações industriais e científicas. Como realimentação visual do estado do contador, o LED verde (PB0) pisca a cada incremento do contador TIM1, o LED amarelo acende quando a contagem atinge 10 e apaga em 20, o LED vermelo acende em 20 e o contador é resetado. Se o botão azul for pressionado, apaga-se o LED vermelho e reinicia o ciclo de contagem.

1. Crie um projeto novo usando o *Cube*, com o nome "Contador_Externo", com a opção **"Initialize all peripherals with their default!" desabilitada**. Ative o módulo *Debug* como "Serial Wire". Entre na aba "Clock Configuration" e modifique a frequência do *clock* do sistema para **96MHz**. Gere o código e abra o arquivo Core/Src/`main.c`. Comente (Remova) a declaração, a definição e a chamada da função `MX_GPIO_Init(void)`.

2. Vamos declarar as funções de configuração dos periféricos utilizado no projeto na área `/* USER CODE BEGIN PFP */`

```
void GPIO_Perif_PInit (void);
void PE8_PInit (void);
void PE7_PInit (void);
void TIM1_PInit(void);
```

3. E vamos definir cada uma das funções na área `/* USER CODE BEGIN 4 */`. Em primeiro lugar, a seguinte função de configuração dos periféricos no NUCLEO para atuação e realimentação visual:

```
void GPIO_Perif_PInit (void) {
	// LED verde
	// Ativa GPIOB (PB0)
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk;
	// PB0 como saida digital
	GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk);
	GPIOB->MODER |= GPIO_MODER_MODE0_0;
	GPIOB->OTYPER &= ~GPIO_OTYPER_OT0_Msk; // PB0 como push-pull
	// LED vermelho
	// Ativa GPIOB (PB14)
	// PB14 como saida digital
	GPIOB->MODER &= ~(GPIO_MODER_MODE14_Msk);
```

```
        GPIOB->MODER |= GPIO_MODER_MODE14_0;
        GPIOB->OTYPER &= ~GPIO_OTYPER_OT14_Msk; // PB14 como push-pull
        // LED amarelo
        // Ativa GPIOE (PE1)
        RCC->AHB4ENR |= RCC_AHB4ENR_GPIOEEN_Msk;
        // PE1 como saida digital
        GPIOE->MODER &= ~(GPIO_MODER_MODE1_Msk);
        GPIOE->MODER |= GPIO_MODER_MODE1_0;
        GPIOE->OTYPER &= ~GPIO_OTYPER_OT1_Msk; // PB0 como push-pull
        // Botao azul
        // Ativa GPIOC (PC13)
        RCC->AHB4ENR |= RCC_AHB4ENR_GPIOCEN_Msk;
        // PC13 como entrada digital
        GPIOC->MODER &= ~(GPIO_MODER_MODE13_Msk);
        // Configuracao de entrada
        GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPD13_Msk); //sem pull
}
```

4. Em seguida, a configuração do pino de entrada PE7 para a função alternativa de *External TRigger input* (ETR) do temporizador TIM1 (AF1)

```
void PE7_PInit (void) {
        // Ativa GPIOE (PE7)
        RCC->AHB4ENR |= RCC_AHB4ENR_GPIOEEN_Msk;
    // Configurar PE7 como função alternativa
    GPIOE->MODER &= ~GPIO_MODER_MODE7_Msk;
    GPIOE->MODER |= GPIO_MODER_MODE7_1;
    // Selecionar AF1 para PE7 (TIM1_ETER)
    GPIOE->AFR[0] &= ~(0xF << GPIO_AFRL_AFSEL7_Pos);
    GPIOE->AFR[0] |= (1 << GPIO_AFRL_AFSEL7_Pos);
}
```

5. Precisamos agora criar um caminho físico para que o sinal gerado pelo botão azul no PC13 chegue ao pino PE7. Para isso, configuramos o pino PE8 com a função GPIO de saída, refletimos por *software* o sinal no pino PC13 no pino PE8 e conectamos o pino PE7 com o pino PE8 através do concector Zio CN10. Insira a seguinte função de configuração do pino PE8 depois da definição da função **`PE7_PInit (void)`**

```
void PE8_PInit (void) {
        // Ativa GPIOE (PE9)
        RCC->AHB4ENR |= RCC_AHB4ENR_GPIOEEN_Msk;
        // PE8 como saida digital
        GPIOE->MODER &= ~(GPIO_MODER_MODE8_Msk);
        GPIOE->MODER |= GPIO_MODER_MODE8_0;
        GPIOE->OTYPER &= ~GPIO_OTYPER_OT8_Msk; // PE8 como push-pull
}
```

6. Vamos agora configurar o temporizador TIM1 para operar no modo de relógio externo 2 (em inglês, *External Clock Mode 2*), ou seja, no modo em que a fonte externa do sinal de

relógio do contador vem de um pino dedicado ETR, seguindo o procedimento recomendado no Manual.

```
void TIM1_PInit(void) {
  // Habilitar o clock para o Timer TIM1
  RCC->APB2ENR |= RCC_APB2ENR_TIM1EN_Msk;
  TIM1->EGR |= TIM_EGR_UG_Msk;   //Atualizacao inicial dos registradores
  while (TIM1->EGR & TIM_EGR_UG);

  // Configurar PSC e ARR
  TIM1->CR1 &= ~TIM_CR1_ARPE_Msk; //Desabilita precarga de autoreload
  TIM1->PSC = 0; // Prescaler
  TIM1->ARR = 20-1; // Período
  // Configurar a fonte de sinal externo (pag. 1443)
   TIM1->SMCR |= TIM_SMCR_ECE; // Habilita o modo ETR externo (modo de relógio
externo 2)
  TIM1->SMCR &= ~(TIM_SMCR_SMS_Msk); // Desabilita o modo escravo
   TIM1->AF1 &= ~(TIM1_AF1_ETRSEL_Msk);  // pinos I/O sao fonte de sinais externos
(PE7)
  // Configurar o modo de contagem usando ETR2 (Fig. 338 e pag. 1443)
  TIM1->SMCR &= ~(TIM_SMCR_ETF_Msk); // Sem filtro ETR2
  TIM1->SMCR &= ~TIM_SMCR_ETPS_Msk; // Prescaler ETR2 (nenhum prescaler)
  TIM1->SMCR |= TIM_SMCR_ETP; // Sensivel a borda de descida
  // Habilitar interrupção TIM1 no NVIC
  NVIC_SetPriority(TIM1_UP_IRQn, 1); // Define a prioridade da interrupção
  NVIC_EnableIRQ(TIM1_UP_IRQn); // Habilita a interrupção TIM1_UP_IRQn
   TIM1->DIER |= TIM_DIER_UIE; // Habilitar geracao de interrupcao por evento de
atualizacao.
  // Reinicializar o contador
  TIM1->CNT = 0;
  // Habilitar o Timer
  TIM1->CR1 |= TIM_CR1_CEN; // Iniciar a contagem
}
```

Esta função configura o temporizador TIM1 como um contador de eventos externos. Para isso, habilita o *clock* do TIM1, desabilita a pré-carga do *autoreload* e define o *prescaler* como 0, resultando em um incremento do contador a cada evento externo. A contagem máxima (*autoreload*) é configurada para 20. O modo de relógio externo 2 é ativado sem filtro, e o contador é configurado para detectar bordas de descida no sinal de entrada. A interrupção de atualização, que ocorre quando o contador atinge o valor máximo, é habilitada com prioridade 1 no NVIC. Finalmente, o contador é reinicializado e a contagem é iniciada.

Consegue imaginar o que seria um modo de relógio externo 2? E um modo 1? Provavelmente não. Mas não se preocupe, exploraremos isso em detalhes mais adiante.

7. Inserimos as chamadas dessas funções na área `/* USER CODE BEGIN 2 */` da função `main` para inicialização dos periféricos

```
GPIO_Perif_PInit ();
PE8_PInit ();
```

```
PE7_PInit ();
TIM1_PInit();
```

6. No laço infinito de `main`, realizamos o *polling* do botão azul (PC13). Se o botão estiver pressionado, é gerado um pulso no pino PB0 onde o LED verde está conectado e no pino PE8 que será ligado ao pino PE7. Além disso é verificado o valor do contador para controlar o estado do LED amarelo. Abaixo da linha `/* USER CODE BEGIN 3 */`, insira o seguinte bloco de código:

```
uint16_t i;
if(GPIOC->IDR & GPIO_IDR_ID13_Msk) { // Botao pressionado
	if (TIM1->CNT < 19 && (GPIOB->IDR & GPIO_IDR_ID14)) {
		GPIOB->BSRR = GPIO_BSRR_BR14; // Apaga LD3 (vermelho)
	}
	GPIOE->BSRR = GPIO_BSRR_BS8; // PULSO em nivel alto
	GPIOB->BSRR = GPIO_BSRR_BS0; // LD1 aceso
	for (i=0; i<5000;i++);       // mantem LD1 aceso por um intervalo de tempo
	GPIOE->BSRR = GPIO_BSRR_BR8; // PULSO em nivel baixo
	GPIOB->BSRR = GPIO_BSRR_BR0; // LD1 apagado
	while(GPIOC->IDR & GPIO_IDR_ID13); // Espera soltar botao
}



if (TIM1->CNT > 9 && !(GPIOE->IDR & GPIO_IDR_ID1)) { // Chegou a 10
	GPIOE->BSRR = GPIO_BSRR_BS1; // Acende LD2 (amarelo)
} else if (TIM1->CNT <= 9 && (GPIOE->IDR & GPIO_IDR_ID1)){
	GPIOE->BSRR = GPIO_BSRR_BR1; // Apaga LD2
}
```

Dentro do bloco de código, observe instruções adicionais que controlam o estado do LED vermelho. Essas instruções garantem que o LED vermelho permaneça aceso após o ciclo de contagem de 20 eventos ser concluído. O LED só será apagado e um novo ciclo de contagem iniciado quando o botão azul for pressionado.

7. Agora vamos implementar a ISR do evento de interrupção *Update*. Abra o arquivo "stm32h7xx_it.c" e localize área `/* USER CODE BEGIN 1 */`. Insira a rotina de serviço:

```
void TIM1_UP_IRQHandler (void) {
	if(TIM1->SR & TIM_SR_UIF) { // Update Interrupt
		TIM1->SR &= ~TIM_SR_UIF; // Limpa flag
		GPIOB->BSRR = GPIO_BSRR_BS14; // Acende LED vermelho
	}
}
```

Além de limpar a *flag*, é aceso o LED vermelho.

8. Interligue com um fio os pinos PE7 e PE8, correspondentes aos pinos 18 e 20 do conector CN10, para que os pulsos gerados pelo botão azul sejam usados como *triggers* externos do temporizador TIM1. Veja a figura no projeto de *Input Capture*.

9. Realize o *Build* e transfira o código para o microcontrolador. Execute o código. Aperte o botão algumas vezes e veja o LED verde acender a cada aperto. Quando o número de apertos chegar a 10, o LED amarelo acenderá. Continue a apertar o botão, em um total de 19 vezes. Quando chegar a 20, o LED vermelho acenderá. O que acontece se pressionarmos mais uma vez o botão azul?

10. Termine e reinicie a execução do program. Faça uma pausa (“Pause”) após três pressionamentos do botão e verifique o valor do registrador `TIM1_CNT`. Retome a execução (“Resume”) e pressione o botão mais uma vez. Faça uma nova pausa (“Pause”) e verifique novamente o valor no registrador `TIM1_CNT`. Continue a execução (“Resume”) e continue pressionando o botão até que o LED amarelo acenda. Faça uma nova pausa (“Pause”) e verifique novamente o valor no registrador `TIM1_CNT`. Continue a execução (“Resume”) e continue pressionando o botão até que o LED vermelho acenda. vermelho seja acionado. Após acionar o LED vermelho, faça uma pausa (“Pause”) e leia o valor do `TIM1_CNT`.

	Quantidade de apertos	Valor no TIM1_CNT
Primeira pausa		
Segunda pausa		
LED amarelo muda de estado (apagado para aceso)		
Depois do LED vermelho acender		

Após o LED vermelho acender, observe atentamente o valor do registrador TIM1_CNT. O que você nota? Tente desvendar a relação entre cada pressionamento do botão e os valores armazenados no registrador. Se o funcionamento desse mecanismo ainda não for claro para você, não se preocupe! Vamos dissecar o circuito responsável pelo comportamento observado mais adiante.

FUNDAMENTOS TEÓRICOS

No Roteiro 5, exploramos temporizadores especializados, como RTC e *Watchdog*, e temporizadores de uso geral para tarefas básicas de temporização. Neste roteiro, introduziremos os **temporizadores avançados**, projetados para aplicações que demandam maior precisão e flexibilidade. Para aumentar a **precisão**, os temporizadores avançados podem integrar desmembrar registradores de módulo (auto-recarga) em registradores sombra

e pré-carga. Além da precisão, os temporizadores avançados oferecem maior **flexibilidade** em diversas áreas:

- modos de operação avançados: Permitem gerar sinais complexos e realizar tarefas de controle precisas.

- controle de saída flexível: Permitem configurar as saídas para gerar sinais com diferentes formas de onda, frequências e larguras de pulso, sincronizando-as com eventos externos.

- sincronização e interconexão: Podem ser sincronizados com outros temporizadores ou eventos externos, e interconectados com periféricos como ADCs e DMAs, para realizar tarefas de controle e aquisição de dados de forma eficiente.

Atualmente, temporizadores avançados são componentes cruciais em sistemas eletrônicos que demandam alta segurança e confiabilidade operacional. Para tanto, alguns incorporam circuitos de *break*, que monitoram continuamente o sistema em busca de falhas, e circuitos geradores de *dead-time*, que previnem a sobreposição de atividade em componentes de operação mutuamente exclusiva. Isso os torna cada vez mais apropriados para serem aplicados no **controle de sistemas de alta potência**, pois permitem o controle preciso de comutação, reduzem a necessidade de componentes externos e aumentam a vida útil dos componentes de potência.

Temporizadores avançados se destacam pela capacidade de interagir com o mundo físico através de múltiplos canais de entrada e saída, característica essencial para sistemas de controle complexos e precisos, como os utilizados em controle de motores e aplicações industriais. Os canais de entrada permitem capturar eventos externos, como pulsos de sensores, através da funcionalidade de **captura de entrada** (em inglês, *input capture* – IC), que registra o valor do contador no momento exato do evento. Já os canais de saída permitem gerar sinais de controle para dispositivos externos, como motores, LEDs e relés. A **comparação de saída** (em inglês, *output compare* – OC) compara o valor do contador com um valor predefinido, executando ações específicas quando há coincidência, enquanto a **modulação por largura de pulso** (em inglês, *pulse width modulation* – PWM) ajusta a potência fornecida aos dispositivos através da variação da largura do pulso.

É importante notar que, apesar de funcionarem de forma independente, todos os canais compartilham o **mesmo contador**, portanto sujeitos à **mesma frequência de *clock* e ao mesmo módulo de contagem.** Isso garante sincronia e precisão nas operações temporais. Como interagem com o ambiente externo, a configuração dos pinos é essencial para que os canais capturem eventos ou enviem sinais.

PRELIMINARES

Os temporizadores avançados são componentes essenciais em sistemas eletrônicos que exigem controle preciso do tempo. Além das funções básicas de contagem e geração de sinais, eles podem incorporar recursos adicionais que aumentam a segurança e a flexibilidade do sistema. Os temporizadores avançados envolvem conceitos que aumentam a flexibilidade e a precisão do controle sobre o tempo, permitindo a sincronização de eventos e a geração de sinais com alta precisão.

Nos temporizadores avançados, o **registrador de módulo**, ou registrador de auto-recarga, e o **registrador *prescaler*** podem ser implementados com dois registradores: **pré-carga** e **sombra** (ou ativo). Ao escrever no pré-carga, o valor é armazenado, mas a alteração efetiva no sombra (em inglês, *shadow register*), que define o valor de contagem ou o valor de pré-escala, ocorre apenas no evento de atualização. A transferência do valor do pré-carga para o sombra, conhecida como **recarga**, permite modificar os parâmetros do temporizador sem interromper a sua operação, assegurando transições suaves e sincronizadas.

Um **evento de atualização** ocorre quando o contador atinge o *overflow* (ou *underflow*, em contagem regressiva) ou quando solicitado por *software* (via *bit* de habilitação) ou por um *trigger* externo. Esse evento determina precisamente o instante de **pré-carga**, ou seja o instante em que ocorrem as alterações nos parâmetros do temporizador. Ao sincronizar a pré-carga, garante-se que as mudanças nos parâmetros do temporizador ocorram de forma sincronizada e sem gerar *glitches* (eventos transitórios) ou comportamentos inesperados.

A frequência dos eventos de atualização é controlada pelo contador de repetição (em inglês, *repetition counter*), otimizando o desempenho. O **contador de repetição** é um registrador que especifica quantas vezes o contador deve atingir *overflow* ou *underflow* antes que um evento de atualização seja gerado. Quando ele estiver desabilitado, o evento de atualização é gerado a cada estouro do contador. Diferente do *postscaler* que modifica a frequência de contagem do contador, o contador de repetição apenas altera a periodicidade dos eventos de atualização.

Em **aplicações de alta potência**, como controle de motores e conversores de energia, a proteção dos componentes contra danos é fundamental. Para isso, temporizadores avançados podem integrar um circuito de *break*, que monitora falhas como sobrecorrente ou curto-circuito e desativa imediatamente os sinais de saída em caso de detecção. Em sistemas que utilizam dispositivos de chaveamento, como MOSFETs e IGBTs, o *dead-time* é fundamental. Ele garante que um dispositivo seja completamente desligado antes que o outro seja ligado, evitando um curto-circuito conhecido como "*shoot-through*"[1]. O **circuito gerador de *dead-time*** insere um pequeno intervalo de tempo entre o desligamento e a ligação dos dispositivos, garantindo a segurança e a confiabilidade do sistema.

[1] *Shoot-through* é um curto-circuito indesejado que ocorre quando dois dispositivos de chaveamento em um circuito de ponte são ativados simultaneamente, causando um fluxo excessivo de corrente e potencial dano aos componentes.

Temporizadores avançados específicos podem gerar sinais com **polaridades configuráveis**, permitindo aplicações como controle bidirecional de motores e geração de formas de onda para testes eletrônicos. Além disso, a **modulação do estado das saídas** em momentos precisos possibilita a criação de sinais complexos, adaptando o temporizador a aplicações especializadas.

As configurações de entrada e saída, juntamente com as polaridades e a possibilidade de desabilitação, aumentam significativamente a flexibilidade dos temporizadores avançados. A alta resolução e baixa latência desses temporizadores garantem respostas rápidas e seguras, adaptando-se a diferentes condições e requisitos, viabilizando controle preciso de motores com pontes H e circuitos de acionamento, incluindo proteção via circuito de *break*, geração de pulsos PWM com *dead-time* ajustável para conversores DC-DC e inversores, com proteção contra sobrecorrente, criação de sinais personalizados (forma de onda, frequência, polaridade) para controle de iluminação, áudio e testes eletrônicos, medição precisa de posição, velocidade e outros parâmetros com leitura de *encoders* e sensores, e sincronização de sinais de saída com eventos externos para controle temporal preciso em automação.

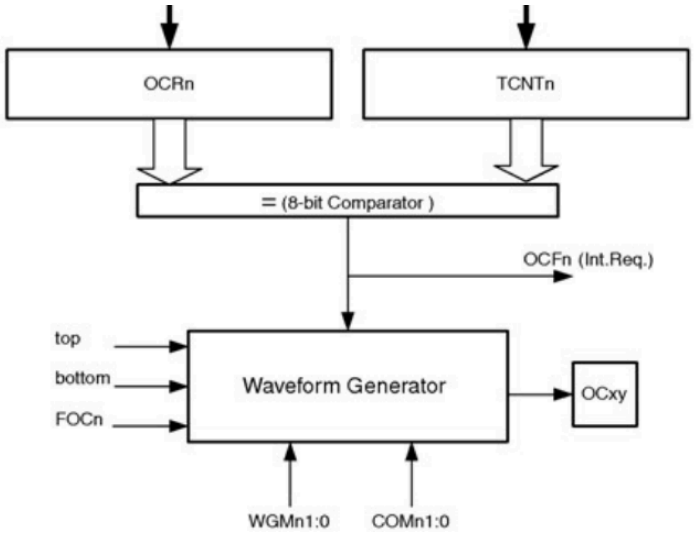
OUTPUT COMPARE

Um uso importante dos temporizadores é a geração de transições precisas de níveis lógicos em saídas digitais em momentos exatos. Porém, ao usar *polling* para ler o valor do contador em um *loop* e executar um comando para alterar o nível lógico de um pino de saída digital, pode haver uma incerteza no tempo. Isso ocorre porque, após o contador atingir o valor desejado, o processador precisa executar várias instruções para ler o contador, comparar seu valor com o valor de referência e atualizar o pino de saída. Esse atraso entre o momento em que o contador atinge o valor desejado e a atualização do estado do pino é conhecido como **latência**.

Uma forma de **reduzir essa latência** é utilizar interrupções. Ao configurar uma interrupção para ocorrer quando o contador atinge um valor específico, você pode ajustar o valor do registrador de módulo do temporizador para o tempo desejado e programar uma rotina de serviço (ISR) que altera o estado lógico do pino de saída. Embora o uso de interrupções reduza a latência, ela não é completamente eliminada, pois vimos no Roteiro 3 que o processador ainda precisa concluir a execução da instrução atual, realizar a troca de contexto e então executar a ISR para alterar o estado do pino.

Para **minimizar a latência**, temporizadores podem realizar a comparação do contador e a alteração do estado lógico dos pinos diretamente por ***hardware***. No modo de **comparação de saída** (em inglês, *Output Compare* – OC), o temporizador compara o valor do contador de n *bits* (TCNTn) com um valor de referência predefinido (OCRn). Quando a comparação é verdadeira, o *hardware* aciona automaticamente eventos, como interrupções (OCFn) e a alteração de estados nos pinos de saída (OCxy), sem intervenção do *software*. Geralmente, utiliza-se um contador de modo livre (em inglês, *free-run counter*), que opera de forma

cíclica, incrementando até o valor máximo e reiniciando automaticamente.



INPUT CAPTURE

Em sistemas embarcados, a sincronização precisa de sinais e a captura exata de eventos externos são fundamentais. Para registrar a ocorrência desses eventos, utiliza-se comumente um pino de entrada digital para detectar transições de nível lógico que sinalizam a ocorrência de um evento. Similarmente ao modo de *Output Compare*, o uso de *polling* ou interrupções externas sensíveis à borda pode introduzir **latência**, o que pode ser significativo dependendo da aplicação.

Para capturar o instante exato em que um evento externo acontece, pode-se utilizar o modo **captura de entrada** (em inglês, *Input Capture*) de um temporizador. Esse modo permite que o ***hardware*** **detecte e registre** com precisão o momento em que um evento externo altera o nível lógico do pino de entrada. Conforme ilustrado na Figura 31-81, o sinal de captura do valor do registrador de módulo/*auto-reload* é proveniente do detector de bordas. Esse modo de operação é especialmente útil para medir a duração de pulsos ou o intervalo entre pulsos, e é essencial em aplicações que exigem medições de frequência precisas.

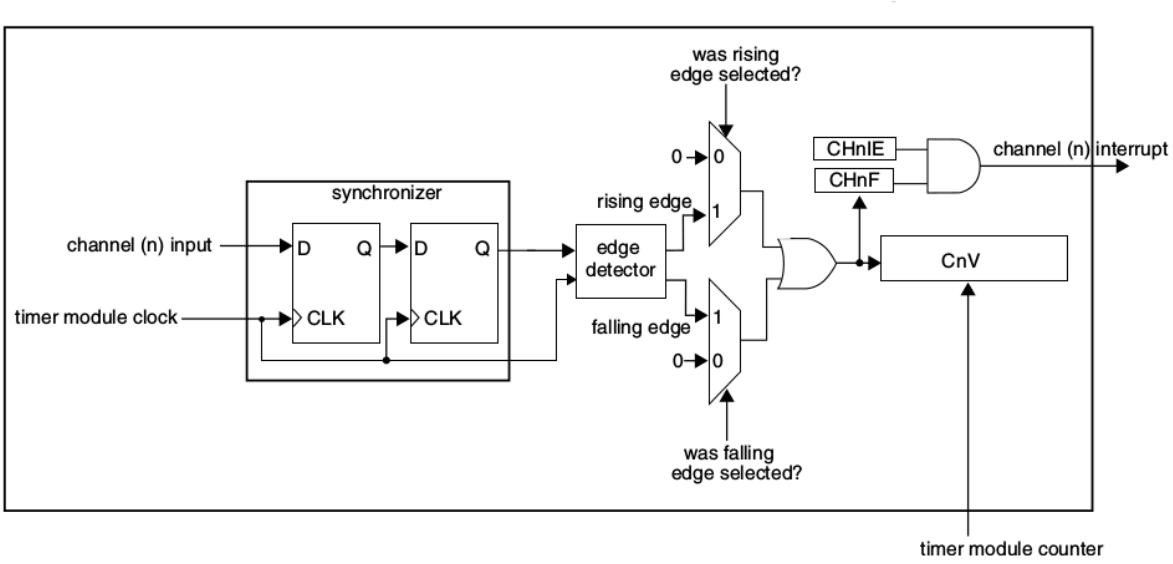


Figure 31-81. Input capture mode

A precisão das capturas realizadas pelo temporizador depende diretamente da frequência do seu contador. A relação entre a resolução do temporizador e sua frequência de *clock* é **linear** e **direta**, sendo determinada pelo período do *clock* (T = 1/f), onde f é a frequência. Quanto **maior a frequência** do *clock*, menor será o período, permitindo que o temporizador registre eventos com **maior precisão**.

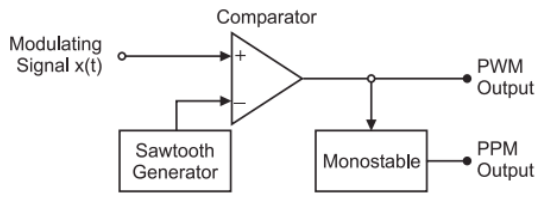
Esse aumento na frequência do temporizador reduz o intervalo de tempo entre duas contagens consecutivas, aumentando assim a resolução da captura. Como consequência, eventos rápidos podem ser medidos com maior exatidão, o que é especialmente relevante para sinais de alta frequência ou eventos de curta duração. Dessa forma, a escolha adequada da frequência do temporizador é essencial para garantir medições confiáveis e precisas em aplicações críticas.

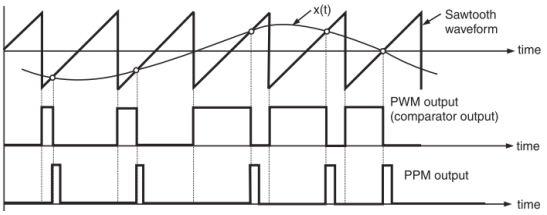
As aplicações práticas do modo *Input Capture* são diversas, incluindo a medição da frequência de sinais de sensores, a captura de pulsos de encoders em sistemas de posicionamento e o controle de eventos em processos de automação industrial. Sua relevância é particularmente distinguível em interfaces de *encoders*, onde a precisão na obtenção de informações sobre o movimento angular é fundamental.

PWM

A **modulação por largura de pulso** (em inglês, *Pulse Width Modulation* - PWM) é uma técnica amplamente utilizada na eletrônica para controlar a potência fornecida a dispositivos como motores, lâmpadas e sinais de áudio. O princípio básico do PWM é ajustar a largura dos pulsos em um sinal digital dentro de uma janela de tempo predefinida, conhecido como **janela de modulação**, a fim de simular uma saída analógica proporcional, como ilustra a figura a seguir. Na figura, o comparador gera um sinal de saída que está alto quando o sinal de modulação é maior que o sinal de dente de serra e baixo quando é menor. O resultado da

comparação é o sinal PWM. Quando o sinal PWM passa por um circuito monoestável, o circuito produz um pulso de duração fixa em resposta a cada transição de borda de descida do sinal PWM. Essa operação resulta em um padrão de pulsos de largura fixa com espaçamento variável entre eles, que é característico do PPM (do inglês *Pulse Position Modulation*).





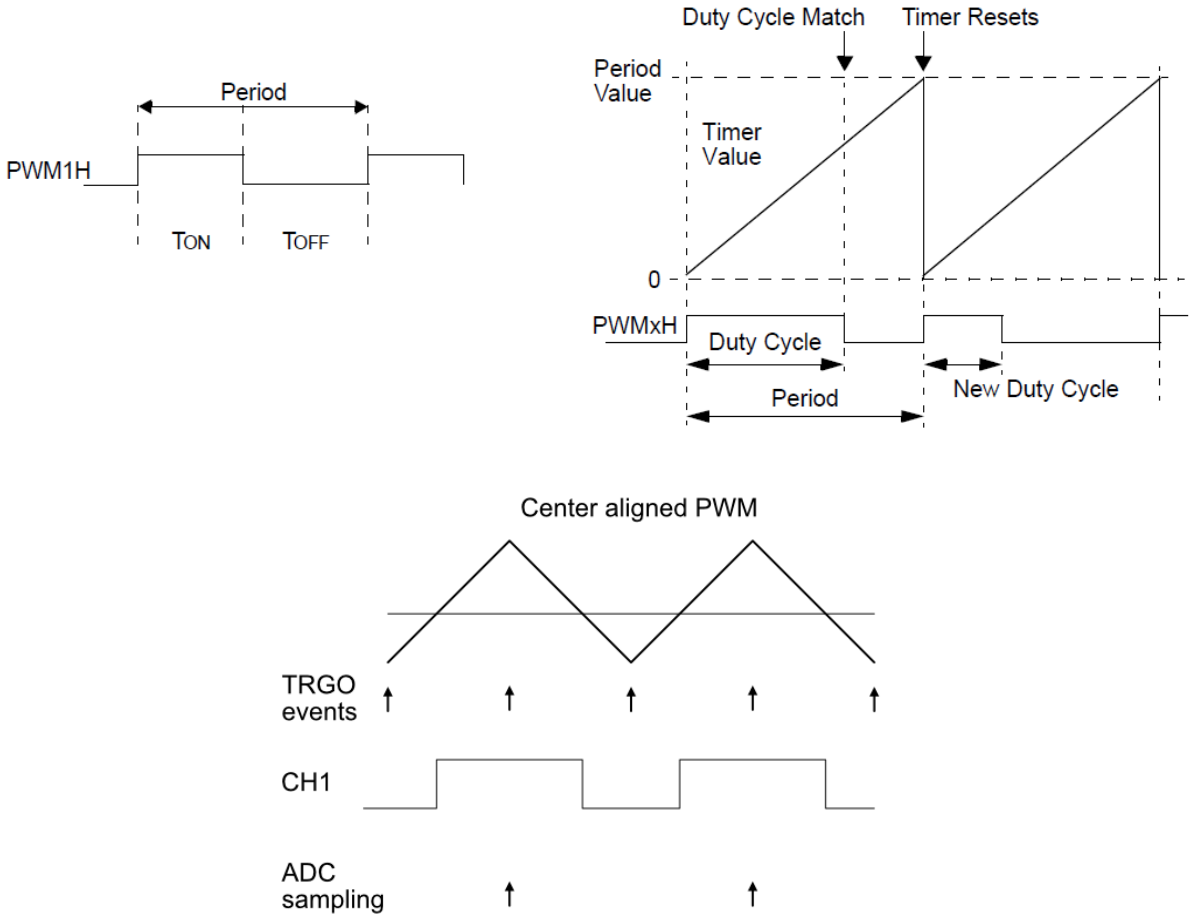
Neste roteiro, focaremos em sinais PWM. O PWM é uma técnica que gera uma onda quadrada de frequência constante, onde a largura do pulso (w) é ajustada em relação ao período total da onda (p). Essa relação, conhecida como **ciclo de trabalho** (em inglês, *duty cycle* – DC), é expressa em porcentagem (%) e calculada pela fórmula:

$$DC(\%) = (w / p) \times 100$$

Ao ajustar o *duty-cycle*, altera-se a tensão média da onda quadrada. Por exemplo, um *duty-cycle* de 100% resulta em uma tensão ativa (V_A) aplicada durante todo o período do ciclo, enquanto um *duty-cycle* de 0% significa que a tensão ativa aplicada é zero volts durante o ciclo. Um *duty-cycle* de 50% corresponde a uma aplicação de tensão média igual à metade de V_A. Portanto, a **tensão média**, que é a tensão efetiva aplicada ao dispositivo, é obtida multiplicando V_A pelo *duty-cycle*, fornecendo um controle preciso sobre a potência fornecida ao dispositivo.

Existem duas formas principais de implementar um gerador de PWM usando temporizadores: alinhamento do valor 0 do contador na borda e no centro do sinal gerado. No modo PWM **alinhado na borda** (em inglês, *edge-aligned*), o contador conta de 0 até o valor máximo de contagem e, em seguida, leva um ciclo de *clock* para reiniciar em 0 (*overflow*). Se plotarmos os valores do contador, ele se parecerá com um sinal em dente de serra. No modo PWM **alinhado no centro** (em inglês, *center-aligned*), o contador conta de 0 até o valor máximo de contagem (*overflow*) e, em seguida, conta de volta até 0 (*underflow*). Se plotarmos os valores do contador, ele se parecerá com um sinal triangular. O sinal alterna de estado quando o valor do contador atinge o valor de comparação predefinido. Esses pontos correspondem ao "*Duty Cycle Match*" e às interseções da linha horizontal com o dente de serra ou sinal triangular nas figuras a seguir.

No modo de alinhamento por borda, o sinal de saída retorna à polaridade configurada (pulso ativo em nível alto ou em nível baixo) quando o contador atinge o valor de *overflow*. Já no modo de alinhamento central, o sinal de saída alterna seu estado lógico a cada comparação bem-sucedida com o valor de *match*, assumindo a polaridade configurada tanto na contagem crescente quanto na contagem decrescente.



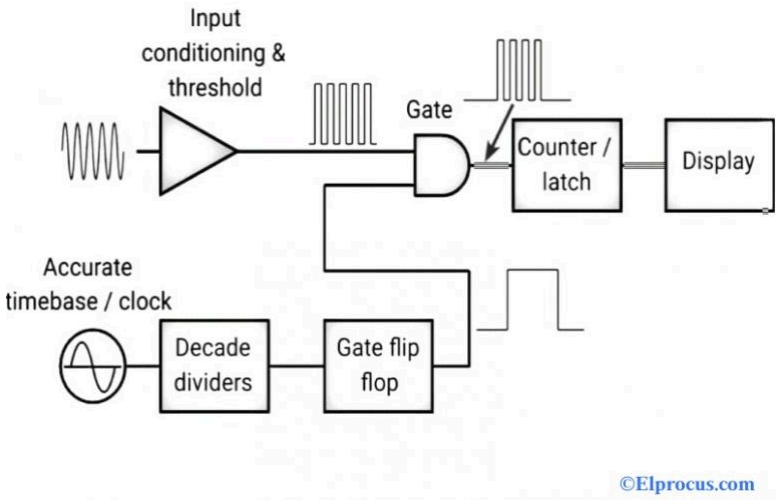
É importante ressaltar que as funções de OC, IC e PWM são implementadas em **canais** de temporizadores. Um módulo temporizador é definido por um único contador com seu *prescaler*, *postscaler* e registrador de módulo (*auto-reload*). Este módulo opcionalmente pode ter um ou mais canais. Cada canal pode ser individualmente configurado para realizar as funções de OC, IC ou PWM, tendo um registrador associado (normalmente denominado **registrador de pulso** ou ***capture/compare***) que determina o tempo transcorrido para o *compare* ou o tempo lido no *capture*, ou ainda a duração do tempo em nível alto/baixo do PWM.

CONTADORES DE EVENTOS EXTERNOS

Os temporizadores são baseados em contadores que registram pulsos de *clock* em intervalos fixos, permitindo medir períodos com precisão. Além de contar pulsos internos, esses contadores podem ser configurados para registrar eventos externos, conectando sua entrada de *clock* a um sinal digital. Nesse modo, o temporizador detecta transições do sinal aplicado a um pino específico, podendo operar em bordas de subida ou descida. Cada transição é registrada no contador, que pode ser incrementado ou decrementado conforme a configuração. O valor acumulado é armazenado em um registrador interno para processamento ou exibição. Essa flexibilidade permite contabilizar eventos externos de forma

acumulativa, configurar o sentido da contagem e definir quais bordas do sinal serão consideradas.

Para maior precisão, contadores podem incluir filtros que reduzem o impacto de ruídos e sinais de alta frequência. Além disso, podem operar em diferentes modos, gerando sinais de saída ou interrupções baseadas na contagem. A figura a seguir ilustra a geração de pulsos controlados ao condicionar um sinal de entrada por meio de uma porta AND, garantindo intervalos de contagem bem definidos.



Embora projetados para sinais assíncronos, os contadores de eventos externos possuem limitações baseadas na frequência do *clock* interno do temporizador. Cada evento externo é detectado em um ciclo de *clock*, o que significa que a frequência máxima de contagem é proporcional à frequência do *clock*. Se os eventos ocorrerem mais rápido do que o temporizador pode processar, ocorrerá perda de pulsos (“subcontagem”), levando a medições imprecisas. Para evitar esse problema, é essencial garantir que a frequência do sinal de entrada esteja dentro dos limites especificados para o temporizador, considerando a taxa de *clock* e as características do *hardware*.

Os contadores de eventos externos são componentes fundamentais em sistemas de temporização e controle, projetados para contar e registrar pulsos ou eventos provenientes de sinais externos com alta precisão. Esses contadores são amplamente utilizados em diversas aplicações, como a medição de frequência, onde o número de pulsos em um intervalo de tempo específico é contado para determinar a frequência de um sinal de entrada. São essenciais também em sistemas de controle de motores e robótica, onde *encoders* fornecem pulsos que são contados para calcular a posição angular ou a velocidade de rotação de um eixo. Além disso, os contadores de eventos externos são usados para monitorar a contagem de eventos, como o número de ciclos de uma máquina ou as ativações de um interruptor, e para gerar interrupções que permitem ao sistema responder rapidamente a mudanças externas.

LINEARIZAÇÃO DE CONTAGEM CÍCLICA

Já pararam para pensar que o tempo, como o percebemos, é uma linha reta que se estende infinitamente? Medimos intervalos, durações, progressões... tudo linear! Mas os temporizadores que vimos até agora contam o tempo de forma cíclica. Como podemos usar essa contagem cíclica para medir o tempo linear que nos interessa?

O **período do contador** é o intervalo de tempo necessário para que o contador complete um ciclo e reinicie. A **frequência do contador** é a taxa na qual o contador incrementa seu valor, ou seja, quantas vezes ele incrementa por segundo. Com essas informações, pode-se converter um valor do contador, digamos C_{T1}, em tempo real através da fórmula

$$t = C_{T1} \times Período_{clock} = \frac{C_{T1}}{f_{clock}},$$

onde f_{clock} é a frequência do relógio que alimenta o contador. No entanto, dado que o **contador é cíclico**, esse valor representa apenas a contagem atual dentro de um período do contador e precisa ser ajustado para refletir o tempo real. Em um contador cíclico, o tempo decorrido desde o início da contagem ou desde um evento de referência (C_{T1}) é calculado com base no período do contador. O período, por sua vez, é determinado pelo número total de contagens possíveis, armazenado no registrador de auto-recarga `TIMx_ARR`:

$$Período = \frac{TIMx\_ARR + 1}{f_{clock}}.$$

Para linearizar os valores da contagem cíclica, é necessário ajustar a contagem atual, considerando o número de ciclos completos do contador. Esse ajuste é realizado pela seguinte fórmula:

$$t = \frac{C_{T1} + N \times (TIMx\_ARR + 1)}{f_{clock}},$$

onde N representa o número de ciclos completos contabilizados. Para medirmos o tempo entre dois eventos, como em instantes T1 e T2, obtemos os valores de contagem C_{T1} e C_{T2}, respectivamente. A diferença de tempo entre essas leituras é calculada pelas seguintes equações:

T1 ≧ T2

$$t = \left(\frac{((TIMx\_ARR + 1) + C_{T2} - C_{T1})}{TIMx\_ARR + 1} + (N - 1)\right) \times Período$$

T1 < T2:

$$t = \left(\frac{(C_{T2} - C_{T1})}{TIMx\_ARR + 1} + N \right) \times Período$$

Através dessas abordagens, é possível converter a contagem cíclica em um intervalo de tempo linear contínuo, considerando a periodicidade e a frequência do contador.

UMA APLICAÇÃO IC/OC: CRONÔMETRO

Os modos de operação IC e OC dos temporizadores destacam-se como ferramentas indispensáveis em sistemas embarcados, permitindo medições e controle de alta precisão. O IC possibilita a captura de eventos externos ao registrar, com precisão, o instante em que eles ocorrem, sendo amplamente utilizado em medições de frequência, duração de pulsos e tempos de resposta em sistemas de controle. Por outro lado, o OC facilita o controle de eventos periódicos ou temporizados, ao acionar saídas em momentos predeterminados. Essas funcionalidades são essenciais para aplicações como controle industrial, automação residencial, dispositivos médicos e sistemas de comunicação, onde eficiência e confiabilidade são essenciais. Vamos apresentar em detalhes o projeto de um cronômetro utilizando ambos os modos de operação.

A função *Output Compare* permite gerar, por *hardware*, um evento de interrupção quando o valor do contador `TIMx_CNT` do temporizador TIMx se iguala ao valor no registrador `TIMx_CCRn` do canal n operando com a função *Output Commpare*. Isso aumenta a precisão e a velocidade da resposta do sistema em relação a uma referência pré-especificada. Em conjunto com a função *Input Capture*, ela simplifica o cômputo do intervalo de tempo entre os dois instantes capturados, T1 e T2, se ambas as funções compartilharem a mesma base de tempo.

Aplicando a linearização de contagem cíclica, o procedimento consiste em setar no registrador `TIMx_CCRn` do canal de função *Output Compare* o valor C_{T1} capturado pelo registrador `TIMx_CCRm` do canal de função *Input Capture* e contar a quantidade de ciclos de contagem do contador em relação a C_{T1}. Essa contagem pode ser implementada habilitando a interrupção do canal *Output Compare* para que ele gere um evento de interrupção cada vez que o contador `TIMx_CNT` passe por C_{T1}, realizando uma contagem de ciclos completos em relação a C_{T1}.

Com a contagem por *hardware* da quantidade *M* de ciclos, juntamente com as contagens C_{T1} no instante inicial T1 e C_{T2} no instante final T2, podemos estimar com precisão o intervalo de tempo t entre as duas contagens C_{T1} e C_{T2} através das expressões:

T1 ≧ T2:

$$t=\left(\frac{\left(\left(TIMx\_ARR+1\right)+C_{T2}-C_{T1}\right)}{TIMx\_ARR+1}+\left(M-1\right)\right)\times Período$$

T1 < T2

$$t=\left(\frac{\left(C_{T2}-C_{T1}\right)}{TIMx\_ARR+1}+M\right)\times Período$$

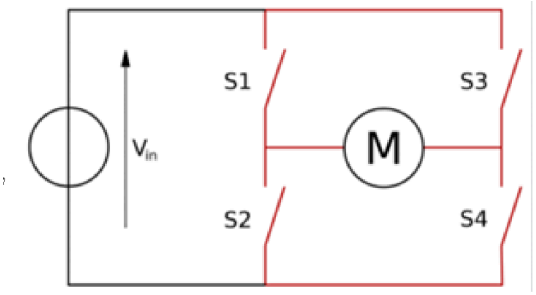
INTERFACES COM MUNDO FÍSICO

Em sistemas embarcados, a comunicação entre microcontroladores e dispositivos físicos ocorre através de circuitos de interface. Quando as condições operacionais dos dispositivos e microcontroladores são compatíveis, a interface é simplificada, como demonstrado na conexão de LEDs e botões em roteiros anteriores. No entanto, para dispositivos que exigem maior potência, como motores, ou controle sofisticado, como a detecção de rotação, são necessários circuitos de interface mais complexos.

Nesta seção, exploramos dois circuitos que exemplificam a interação entre o microcontrolador e o mundo físico: a interface com um motor (atuador) e a interface com um *encoder* (sensor). **Atuadores** convertem sinais elétricos em ações físicas, como movimento ou calor, enquanto **sensores** detectam e convertem grandezas físicas, como temperatura ou luz, em sinais elétricos.

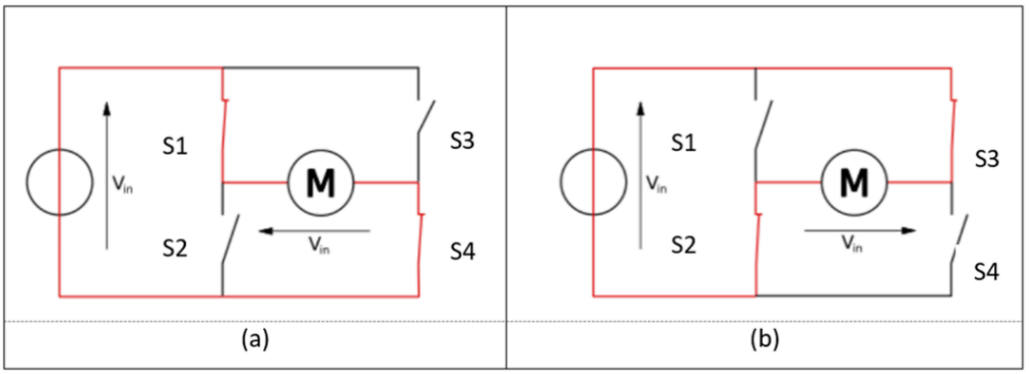
PONTE-H

Uma **ponte-H** é um circuito eletrônico que controla a polaridade de uma tensão aplicada a uma carga, sendo frequentemente usados para o controle de motores CC ou de motores de passo bipolares em robótica. As pontes-H estão disponíveis como circuitos integrados ou podem ser construídas a partir de componentes discretos. O termo *ponte-H* é derivado da representação gráfica típica de um circuito desse tipo, conforme ilustra a figura abaixo.



Uma ponte-H é construída com quatro interruptores (de estado sólido ou mecânico). Quando

os interruptores S1 e S4 (de acordo com a figura abaixo) estão fechados e S2 e S3 estão abertos, uma tensão positiva será aplicada no motor. Ao abrir os interruptores S1 e S4 e fechar os interruptores S2 e S3, essa tensão é revertida, permitindo a operação reversa do motor. Usando a nomenclatura acima, os interruptores S1 e S2 nunca devem ser fechados ao mesmo tempo, pois isso poderia causar um curto-circuito na fonte de tensão de entrada. A mesma restrição se aplica aos comutadores S3 e S4. Essa condição, conhecida como ***shoot-through***, pode ser evitada com o uso de um circuito gerador de *dead-time*.



O arranjo da ponte-H geralmente é usado para reverter a polaridade/direção do motor, mas também pode ser usado para "travar" o motor, onde o motor pára repentinamente, quando os terminais do motor estão em curto, ou para deixar o 'free run' do motor, pois o motor é efetivamente desconectado do circuito. A tabela abaixo resume a operação, com S1-S4 correspondentes à figura acima.

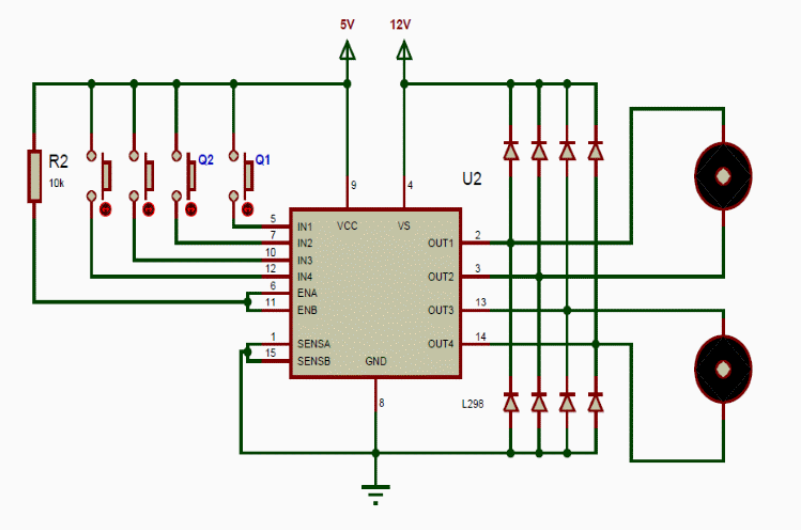
S1	S2	S3	S4	Resultado
1	0	0	1	O motor gira num sentido
0	1	1	0	O motor gira noutro sentido
0	0	0	0	Inércia, o motor gira livre até parar
1	0	0	0	
0	1	0	0	
0	0	1	0	
0	0	0	1	
0	1	0	1	O motor trava (freia)
1	0	1	0	
x	**x**	**1**	**1**	**Curto circuito na fonte**
1	**1**	**x**	**x**	

O **L298** é **um circuito integrado com duas pontes-H** cujas entradas de controle são compatíveis com níveis de lógica TTL. Suas duas pontes-H podem acionar cargas indutivas como relês, solenóides e motores de corrente contínua com valores máximos de tensão até 46 volts e de corrente até 2A em cada ponte. A figura a seguir ilustra a conexão de dois motores aos 4 sinais de alimentação de +5V/+12V através de L298. Os oito diodos são diodos de *flyback*, usados para proteger o CI contra picos de tensão indutiva. Os pinos de habilitação, ENA e ENB, estão sempre acionados através de um resistor, garantindo que a ponte H funcione continuamente; se for puxado para o terra, a ponte H será desativada,

independentemente da lógica de controle. Após conectar os pinos 5, 7, 10 e 12 ao terra usando resistores *pull-down* (um para cada pino), analisaremos como os botões Q1 e Q2, que servem como entradas de controle para a ponte H, afetam o fluxo de corrente entre os pinos OUT1 e OUT2. A tabela de controle lógico é a seguinte:

- Q1=HIGH, Q2=LOW: Corrente no sentido horário (S1 e S4 ou S2 e S3 fechadas).
- Q1=LOW, Q2=HIGH: Corrente no sentido anti-horário (o par de chaves complementares fechadas) .
- Q1=Q2: Parada rápida do motor (4 chaves abertas).

Assim, é possível controlar o sentido de rotação do motor utilizando o *chip* L298. Substituindo o controle manual dos botões Q1 e Q2 por sinais programáticos de um microcontrolador e a alimentação +5V/+12V por um sinal PWM, é possível obter controle automático do motor, permitindo ajustar a potência transferida e, consequentemente, a sua velocidade.



Para maiores informações, vide o *datasheet* do L298.

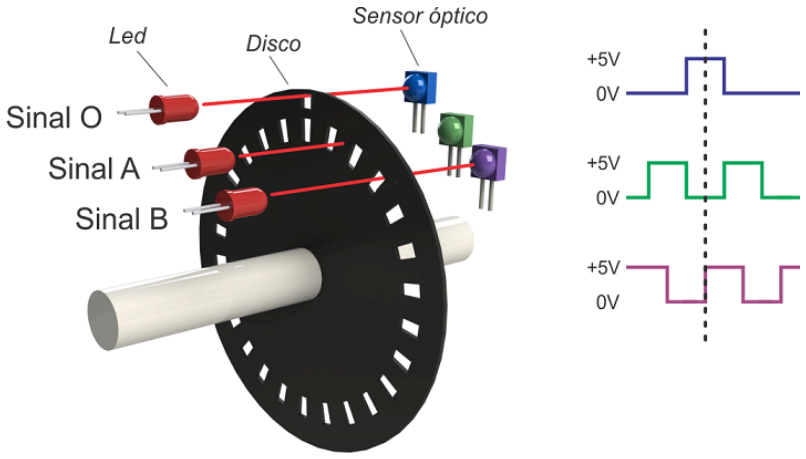
ENCODERS

Um ***encoder*** é um dispositivo que converte uma posição ou movimento mecânico em um sinal elétrico que pode ser lido e interpretado por um sistema eletrônico. *Encoders* são amplamente usados em sistemas de controle, automação industrial, robótica e em qualquer aplicação que necessite de medição precisa de posição ou velocidade. Os *encoders* podem ser classificados principalmente em duas categorias: *encoders* rotativos e *encoders* lineares.

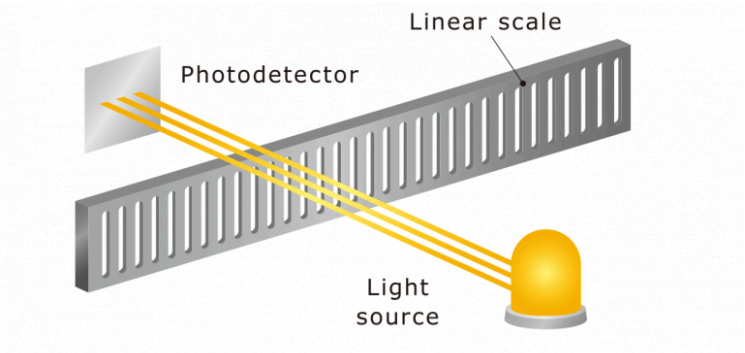
O *encoder* rotativo consiste em um disco giratório com marcas ou padrões codificados (geralmente faixas de luz e sombra ou pistas de metal condutoras e isolantes). À medida que

o disco gira, um sensor óptico (em *encoders* ópticos) ou magnético (em *encoders* magnéticos) detecta as marcas e gera sinais elétricos correspondentes. Estes sinais podem ser digitais (como um código binário) ou analógicos (como uma tensão variável). Alguns *encoders* fornecem sinais **incrementais**, que indicam a mudança na posição, e outros fornecem sinais **absolutos**, que indicam a posição exata.

Os *encoders* rotativos típicos usam **codificação por quadratura** para detectar o sentido de rotação. Esta codificação é baseada em dois sinais principais, conhecidos como canais A e B, que são gerados com um desfasamento de 90 graus entre si. O canal A produz uma onda quadrada que varia entre "1" e "0" conforme o *encoder* gira e o canal B produz também uma onda quadrada, mas com um desfasamento de 90 graus em relação ao canal A, conforme mostra a figura a seguir. O desfasamento entre esses dois sinais permite determinar a direção do movimento. Por exemplo, se o canal A está à frente do canal B, ou seja, canal A muda de estado antes de canal B, isso pode indicar que o *encoder* está girando num sentido (para a disposição mostrada na figura, sentido anti-horário). Se o canal B está à frente do canal A, isso pode indicar que o encoder está girando no outro sentido. O terceiro sensor geralmente é um **sensor de referência** ou **pulsador de índice**. Ele gera um pulso único por ciclo, fornecendo uma referência para a contagem de pulsos dos canais A e B.



O *encoder* linear tem um elemento de leitura e uma régua ou fita com marcas codificadas. À medida que o elemento de leitura se move ao longo da régua ou fita, ele detecta as marcas e gera um sinal elétrico correspondente. Para um *encoder* linear, a técnica usual para detectar o sentido de deslocamento é também baseada na **codificação por quadratura**, que é similar ao método usado em *encoders* rotativos. O encoder linear tipicamente possui dois canais de sinal (A e B) que são emitidos em fases deslocadas 90 graus entre si. A direção do deslocamento é determinada pela sequência dos sinais A e B. Se canal A muda de estado antes de canal B, o *encoder* está se movendo para um lado e se canal B muda de estado antes de canal A, o encoder está se movendo para outro lado.



Optical linear encoder (Transmissive type)

A integração de circuitos de interface de *encoders* em temporizadores avançados oferece vantagens significativas em termos de precisão, eficiência e facilidade de uso, reduzindo a necessidade de circuitos externos para condicionamento e processamento dos sinais do encoder. Embora a conexão direta via GPIO seja possível, ela apresenta limitações que podem comprometer o desempenho do sistema, especialmente em aplicações que exigem alta precisão e velocidade. A integração de temporizadores e *encoders* permite um controle de movimento preciso em sistemas que exigem alta resolução e velocidade. Em sistemas de controle de motores, o *encoder* gera pulsos proporcionais à rotação do eixo. O *Input Capture* do temporizador registra o momento exato desses pulsos e, ao contá-los em um intervalo de tempo definido, é viável calcular a velocidade do motor. Além disso, o temporizador pode gerar sinais sincronizados com os pulsos do *encoder*. Isso permite ajustes finos na posição do motor, garantindo um controle de movimento mais exato. Outra função importante é a detecção de falhas. O temporizador monitora o tempo de resposta do *encoder*, identificando possíveis problemas ou atrasos. Isso ajuda a calibrar o sistema e assegurar que tudo funcione perfeitamente. Essas funcionalidades são cruciais em robótica, controle de motores e automação industrial.

STM32H7A3

No Roteiro 5, apresentamos uma série de temporizadores com funções dedicadas integradas no microcontrolador STM32H7A3, incluindo interrupções periódicas, relógio, pulso único e *watchdog*. Nesta seção, detalhamos temporizadores TIM1/TIM8 e TIM2/TIM3/TIM4/TIM5 que se destacam por suas capacidades avançadas e flexibilidade, oferecendo maior escalabilidade e sendo ideais para sistemas que exigem precisão, alta frequência, sincronização, segurança e confiabilidade. Esses temprorizadores suportam modos PWM, possuem unidades de captura e comparação aprimoradas e oferecem maior integração com sinais externos, garantindo alta performance e versatilidade em diversos cenários de uso.

TIM2/TIM3/TIM4/TIM5

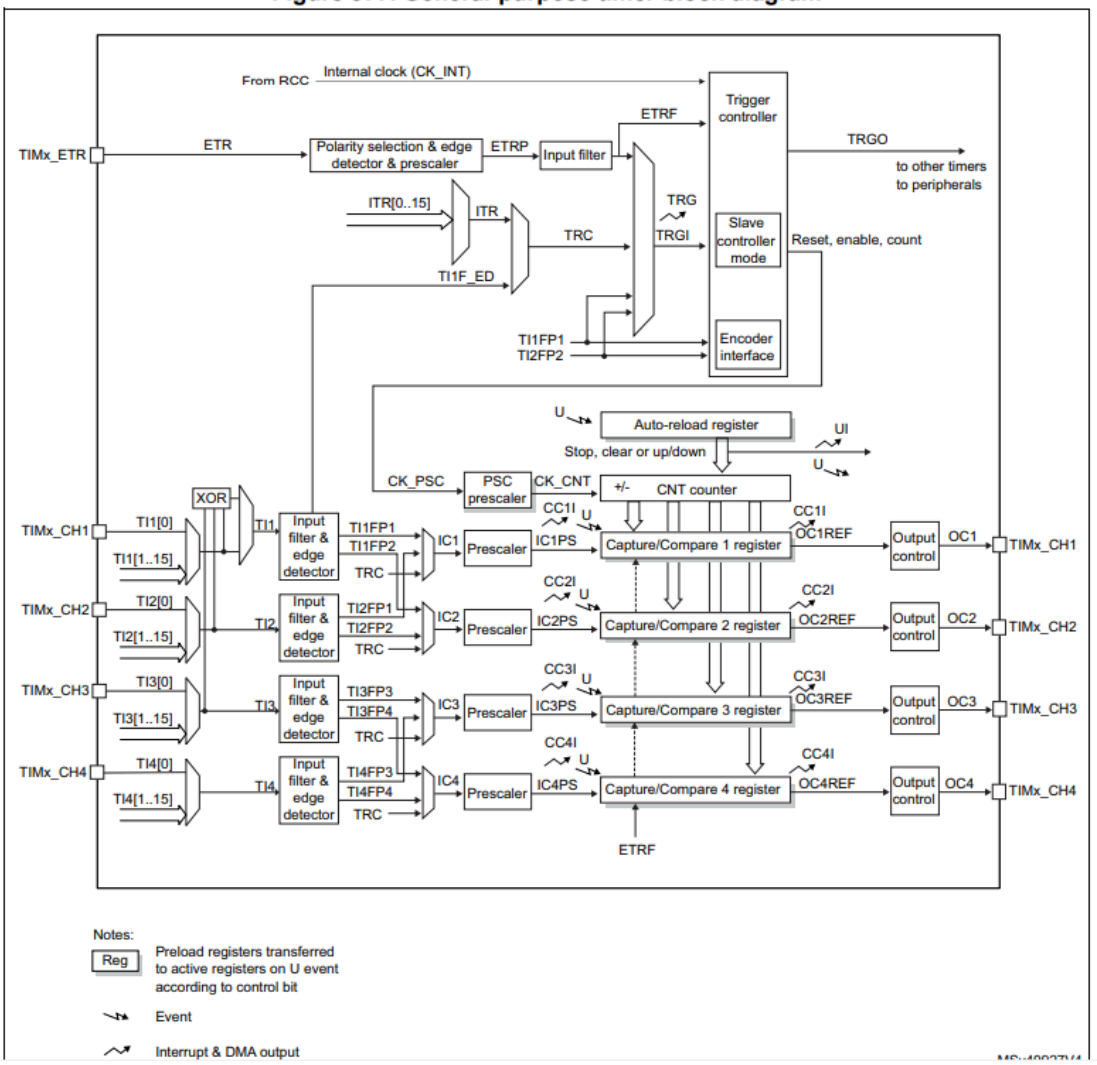
O bloco principal dos temporizadores TIM2/TIM3/TIM44/TIM5 é composto por um contador de 16 *bits* (TIM3 e TIM4) ou 32 *bits* (TIM2 e TIM5) e um registrador de *auto-reload* (registrador de módulo) associado. O contador pode operar em modo de contagem crescente, decrescente ou mista (crescente e decrescente). O relógio do contador pode ser ajustado por meio de um *prescaler*. Tanto o registrador de *auto-reload* quanto o registrador de *prescaler* podem ser lidos e escritos por *software*, mesmo quando o contador está em operação, por que eles são formados por dois registradores, um pré-carga e outro sombra. TIM2, TIM3, TIM4 e TIM5, como os temporizadores TIM6 e TIM7, estão conectados ao barramento APB1.

A unidade de base de tempo inclui:

- Registrador de Contador (TIMx_CNT),
- Registrador de *Prescaler* (TIMx_PSC),
- Registrador de *Auto-reload* (TIMx_ARR).

O registrador *auto-reload* é pré-carregado. Quando se escreve ou lê do registrador de *auto-reload*, o acesso é feito ao registrador pré-carga. O conteúdo deste registrador pré-carga é transferido para o registrador *shadow*, ou registrador ativo, de forma permanente ou a cada evento de atualização (UEV), dependendo do *bit* de habilitação de pré-carga de *auto-reload* `TIMx_CR1_ARPE` no registrador `TIMx_CR1`. O evento de atualização ocorre quando o contador atinge *overflow* (ou *underflow*, no caso de contagem regressiva), se o *bit* `TIMx_CR1_UDIS` no registrador `TIMx_CR1` for setado em 0. Esse evento também pode ser gerado por *software* ao setar o *bit* `TIMx_EGR_UG` (atualização dos registradores). Além disso, eventos de atualização podem ser gerados por um *trigger* interno ou externo através do controlador de modo escravo (em inglês, *slave mode*). O modo *slave* foi apresentado no Roteiro 5.

Figure 377. General-purpose timer block diagram



O contador é sincronizado pelo sinal de *clock* CK_CNT, proveniente do *prescaler*. A contagem é iniciada ao configurar o *bit* TIMx_CR1_CEN no registrador TIMx_CR1. No entanto, a ativação efetiva do contador ocorre com um atraso de um ciclo de *clock* após a configuração desse *bit*.

A configuração de valores máximos de contagem, MOD, como MOD-1 no registrador TIMx_ARR e o valor de divisão de frequência, *prescaler*, como *prescaler* -1 no TIMx_PSC é uma prática comum em STM32, assim como em muitos outros microcontroladores. Os contadores geralmente começam a contar a partir de zero. Ao configurar o valor máximo como MOD-1 ou *prescaler*-1, estamos efetivamente definindo um intervalo de contagem que inclui zero. Em termos de *hardware*, essa abordagem simplifica a implementação do contador.

As **funções básicas de *Input Capture*, *Output Compare* e *PWM*** apresentam configuração semelhante nos temporizadores TIM1/TIM8 e TIM2/TIM3/TIM4/TIM5. Contudo, TIM1 e TIM8 oferecem recursos adicionais, como o *bit* de habilitação de saída (`TIMx_BDTR`) e a configuração de sinais complementares, ausentes nos demais. Nos TIM2, TIM3, TIM4 e

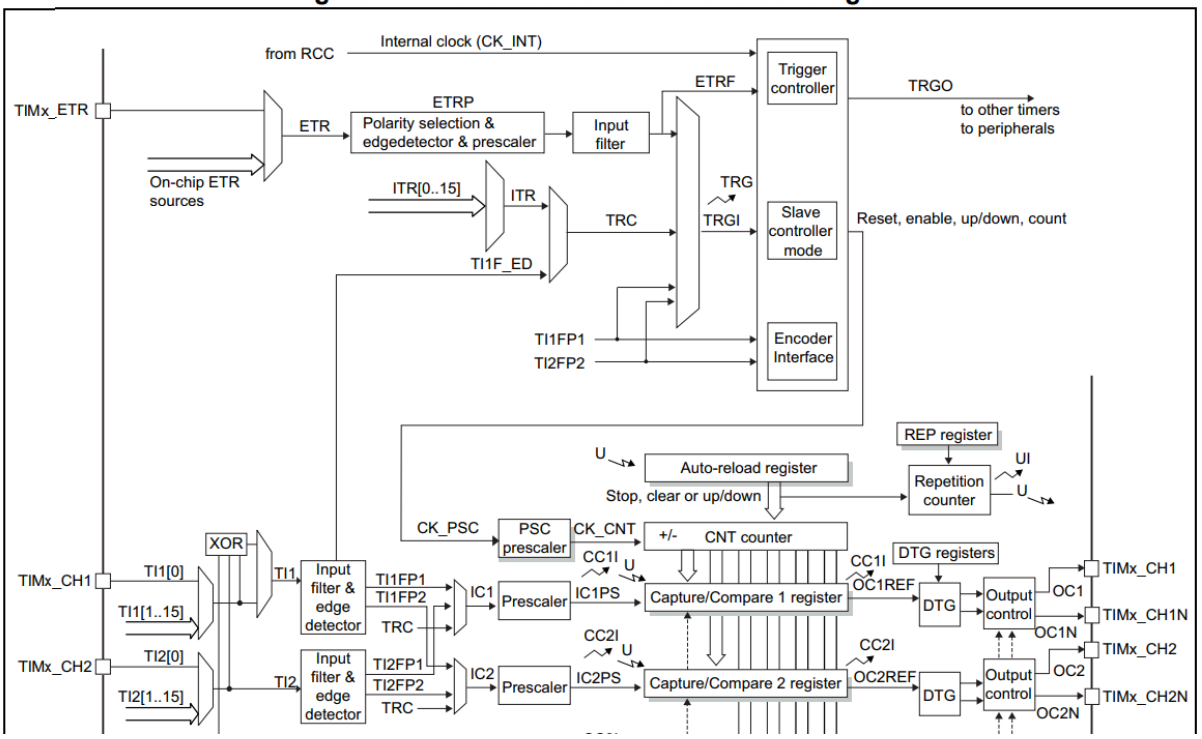
TIM5, a saída é sempre habilitada por padrão. Apesar das diferenças, detalharemos as funções básicas dos TIM1 e TIM8, que englobam as funcionalidades dos TIM2/TIM3/TIM4/TIM5. Assim, as características avançadas dos TIM1 e TIM8 são apresentadas como um superconjunto, destacando que as diferenças residem nos recursos não configuráveis dos últimos.

TIM1/TIM8

O componente principal dos temporizadores programáveis TIM1/TIM8 é um contador de 16 *bits*, TIMx_CNT ou CNT *counter*, associado a um registrador de recarga automática, TIMx_ARR. Este contador pode operar em modo de **contagem crescente**, **decrescente** ou **bidirecional**, configurável pelo campo TIMx_CR1_CMS[1:0] e o *bit* TIMx_CR1_DIR do registrador TIMx_CR1. O relógio do contador pode ser ajustado através de um *prescaler* TIMx_PSC, permitindo dividir a frequência do sinal CK_PSC num sinal de frequência menor CK_CNT para cronometrar TIMx_CNT. A frequência CK_CNT é igual a CK_PSC/(TIMx_PSC[15:0] + 1). O registrador de recarga automática e o registrador de *prescaler* são acessíveis para leitura e escrita por *software*, mesmo enquanto o contador está em operação.

O **registrador de recarga automática** (TIMx_ARR) é pré-carregado; ao escrever ou ler neste registrador, acessa-se de fato o **registrador pré-carga**. O conteúdo do registrador pré-carga então é transferido para o **registrador sombra**, ou registrador ativo, seja permanentemente ou a cada evento de atualização (UEV), conforme a configuração do *bit* de habilitação de recarga automática (ARPE) no registrador TIMx_CR1. O evento de atualização é gerado quando o contador atinge o valor de *overflow* (em modo de contagem crescente) ou *underflow* (em modo de contagem regressiva), se o *bit* UDIS no registrador TIMx_CR1 estiver definido como 0 e se o registrador TIMx_RCR atingir 0. O **registrador de repetição** (TIMx_RCR) define quantos *overflows* ou *underflows* devem ocorrer após o primeiro antes que um evento de atualização seja gerado; para um valor N em TIMx_RCR, o evento de atualização (UEV) ocorre a cada N+1 *overflows* ou *underflows*. Os detalhes sobre a geração de eventos de atualização variam conforme a configuração específica, incluindo o modo de contagem, os *bits* TIMx_CR1_UDIS e TIMx_CR1_URS, o uso do TIMx_RCR e eventos gerados por *software* ou pelo controlador de modo escravo.

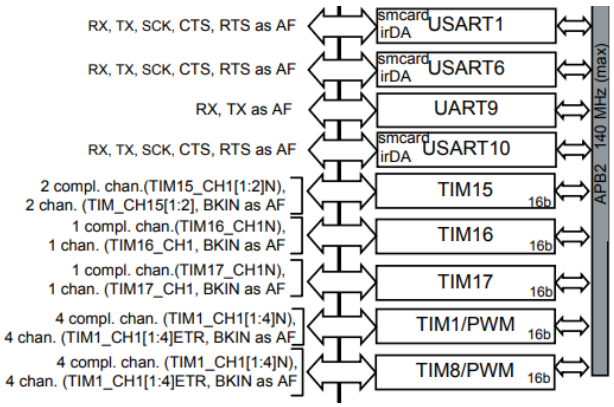
Figure 317. Advanced-control timer block diagram



O **contador** é sincronizado pela saída do ***prescaler***, CK_CNT, que é ativada apenas quando o bit de habilitação do contador (CEN) no registrador TIMx_CR1 está setado em 1. Vale ressaltar que o contador começa a contar um ciclo de *clock* após o *bit* CEN ser setado no registrador `TIMx_CR1`. O prescaler pode dividir a frequência do *clock* do contador por um fator que varia de 1 a 65536. Ele é implementado com um contador de 16 *bits*, controlado pelo registrador `TIMx_PSC` de 16 *bits*. Para permitir ajustes rápidos, o valor do *prescaler* é bufferizado. A nova configuração do *prescaler* é aplicada na próxima ocorrência de um evento de atualização (UEV)

Os sinais de relógio do contador podem ser fornecidos por uma das seguintes **fontes de relógio**, configurável pelo registrador `TIMx_SMCR`:

- **Relógio Interno (CK_INT)** (em inglês, *Internal Clock*): Se `TIMx_SMCR_SMS`=0b000, o sinal de *clock* interno é gerado pelo próprio microcontrolador através do barramento APB2 (mostado na Figura 1 do *Datasheet*). É útil para aplicações que não requerem fontes externas de *clock* e onde a precisão do *clock* interno é suficiente.



- **Modo de Relógio Externo 1 (Pino de Entrada Externo)** (em inglês, *External clock mode 1*): Nesse modo, o contador do temporizador recebe seu sinal de *clock* de um pino de entrada selecionado, derivado de um dos canais de captura/comparação do temporizador (TI1 ou TI2). Este pino pode ser configurado para fornecer um sinal de temporização proveniente de uma fonte externa, permitindo que o temporizador seja sincronizado com sinais gerados fora do microcontrolador.

- **Modo de Relógio Externo 2 (Entrada de Gatilho Externo ETR)** (em inglês, *External clock mode 2*): O sinal de *clock* é fornecido através de um pino de entrada de gatilho externo dedicado, conhecido como ETR (do inglês *External Trigger*). Esse modo é útil para sincronizar o temporizador com eventos externos específicos e é frequentemente usado em aplicações que necessitam de uma sincronização precisa com sinais externos.

- **Modo *Encoder*** (em inglês, *Encoder mode*): Neste modo, o temporizador é configurado para receber sinais de um *encoder*, que é um dispositivo que fornece informações sobre posição, velocidade ou rotação. O modo *encoder* utiliza sinais de quadratura de dois canais para contar e medir a rotação do *encoder*, permitindo uma medição precisa de movimento ou posição em sistemas de controle de motores e robótica.

O evento de *overflow* ou *underflow* é gerado quando o contador atinge seu valor máximo (no caso de contagem crescente) ou mínimo (no caso de contagem decrescente). Para habilitar as **interrupções** relacionadas a esses eventos UI (*Update Interrupt*), o *bit* de interrupção de atualização `TIMx_DIER_UIE` deve ser configurado no registrador `TIMx_DIER`. Além disso, o NVIC deve ser configurado para reconhecer e gerenciar essas interrupções. Isso inclui habilitar a linha de solicitação de interrupção específica para o TIM1 ou TIM8 nos registradores NVIC_ISERn e ajustar a prioridade da interrupção utilizando o registrador NVIC_IPRn. Conforme indicado na Tabela 123 do Manual de Referência, cada temporizador é alocado com quatro linhas IRQ, das quais três linhas de TIM8 são compartilhadas com outros temporizadores. Para o TIM1, a linha IRQ correspondente ao evento de atualização, `TIM1_UP`, é IRQ25. Para habilitar a interrupção, é necessário habilitar o *bit* (25&0x1F) do registrador NVIC_ISERn, n=25>>5, e setar a prioridade no *byte* (25&0b11) do registrador NVIC_IPRn, n=25>>2.

tim1_brk_it	31	24	TIM1_BRK	TIM1 break interrupt	0x0000 00A0
tim1_upd_it	32	25	TIM1_UP	TIM1 update interrupt	0x0000 00A4
tim1_trg_it	33	26	TIM1_TRG_COM	TIM1 trigger and commutation interrupts	0x0000 00A8
tim1_cc_it	34	27	TIM1_CC	TIM1 capture / compare interrupt	0x0000 00AC

tim8_brk_it tim12_gbl_it	50	43	TIM8_BRK_TIM12	TIM8 break and TIM12 global interrupts	0x0000 00EC
tim8_upd_it tim13_gbl_it	51	44	TIM8_UP_TIM13	TIM8 update and TIM13 global interrupts	0x0000 00F0
tim8_trg_it tim14_gbl_it	52	45	TIM8_TRG_COM _TIM14	TIM8 trigger /commutation and TIM14 global interrupts	0x0000 00F4
tim8_cc_it	53	46	TIM8_CC	TIM8 capture / compare interrupts	0x0000 00F8

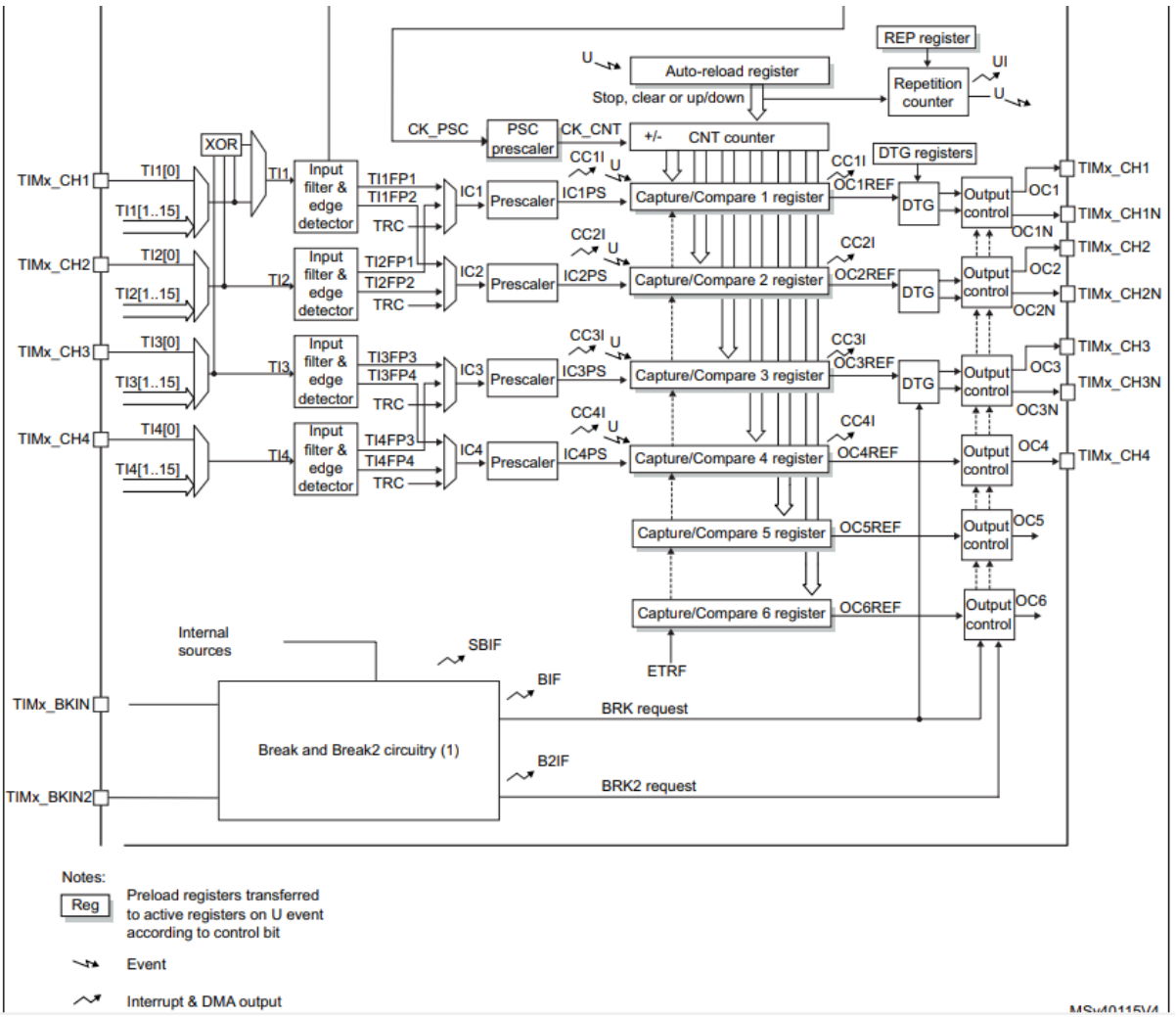
Os temporizadores TIM1 e TIM8 possuem seis canais de **captura/comparação** (CCn). Estes canais podem ser configurados individualmente para operar em diversos modos:

- Captura de Entrada: Permite medir a duração de sinais externos através dos sinais de entrada TI1[2] a TI4 para os canais n=1 a 4, respectivamente. O tempo entre as bordas dos sinais de entrada é registrado nos respectivos registradores `TIMx_CCRn`.
- Comparação de Saída: Possibilita gerar interrupções ou alterar o estado dos pinos de saída (canais n=1 a 4) quando o contador `TIMx_CNT` atinge um valor predefinido no registrador .
- Geração de PWM: Os canais (principalmente n=1 a 4) podem gerar sinais com largura de pulso controlada, definindo o ciclo de trabalho do sinal de saída. Isso é amplamente utilizado para controle de potência e velocidade.

Os canais 5 e 6 são destinados principalmente para uso interno, como na geração de formas de onda complexas ou para disparo de conversores analógico-digitais (ADCs). Embora todos os seis canais possam ser configurados para comparação de saída, apenas os canais 1 a 4 estão diretamente associados aos sinais de entrada externos TI1, TI2, TI3 e TI4 para a função de captura de entrada, que também podem ser configurados para a função de saída (comparação de saída ou PWM).

Além disso, TIM1 e TIM8 possuem um sistema de proteção avançado, com duas **entradas de parada** (em inglês, *break inputs*) independentes. Essas entradas desabilitam as saídas PWM do temporizador em resposta a eventos externos ou internos, assegurando a segurança do sistema. Essa funcionalidade é essencial para garantir a operação segura e confiável em sistemas de controle industrial, automação, conversores de potência e outras aplicações onde a resposta rápida a condições de falha é crucial para proteger o equipamento e o sistema como um todo.

[2] TI = *Timer Input*

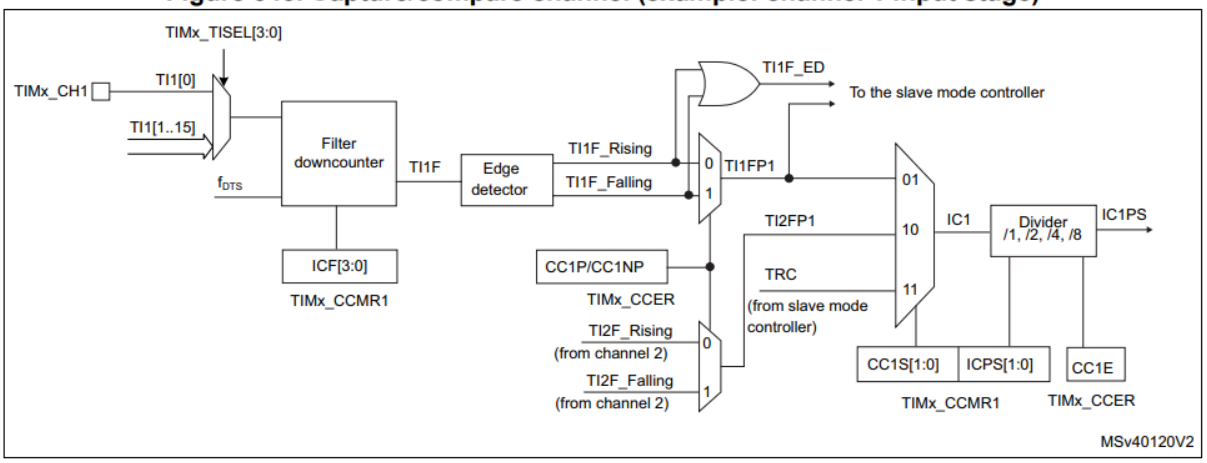


Um **canal de Captura/Comparação**, CCn, do TIM1 é estruturado com três componentes principais: um **registrador de captura/comparação** `TIMx_CCRn`, um **estágio de entrada para captura**, e um **estágio de saída**. O registrador `TIMx_CCRn` armazena o valor capturado ou de comparação. O estágio de entrada para captura é responsável por processar o sinal de entrada e inclui filtros digitais, multiplexação e um *prescaler*, exceto para os canais 5 e 6, que operam de maneira diferente. Por fim, o estágio de saída é composto por um comparador e o controle de saída, que geram o sinal final com base na comparação do valor do contador com o valor no registro de captura/comparação.

No **estágio de captura**, um sinal de entrada `TIn`, proveniente de um dos pinos de captura, é recebido pelo temporizador. Inicialmente, o sinal passa por um circuito de filtragem digital, configurável através dos *bits* `ICnF[3:0]` no registrador `TIMx_CCMRm` (onde m=(n−1)>>1), resultando no sinal filtrado `TInF`. Em seguida, um detector de borda analisa `TInF` para gerar sinais correspondentes às bordas de subida e descida, denominados `TIxFP1` e `TIxFP2`, respectivamente. A polaridade configurada pelo *bit* `CCnP` no

registrador TIMx_CCER define qual dessas bordas será considerada ativa. Após a filtragem e detecção de borda, um multiplexador, controlado pelos *bits* `CCnS` do registrador `TIMx_CCMRm`, seleciona a fonte do sinal de captura de entrada, denominado ICn. O sinal ICn pode ser opcionalmente processado por um prescaler, configurado pelos bits ICnPSC em `TIMx_CCMRm`, gerando o sinal `ICnPS`. Finalmente, se o *bit* de habilitação do canal, `CCnE`, no registrador TIMx_CCER estiver em '1', uma transição válida em `ICnPS` causa a captura do valor atual do contador TIMx_CNT no registrador TIMx_CCRn.

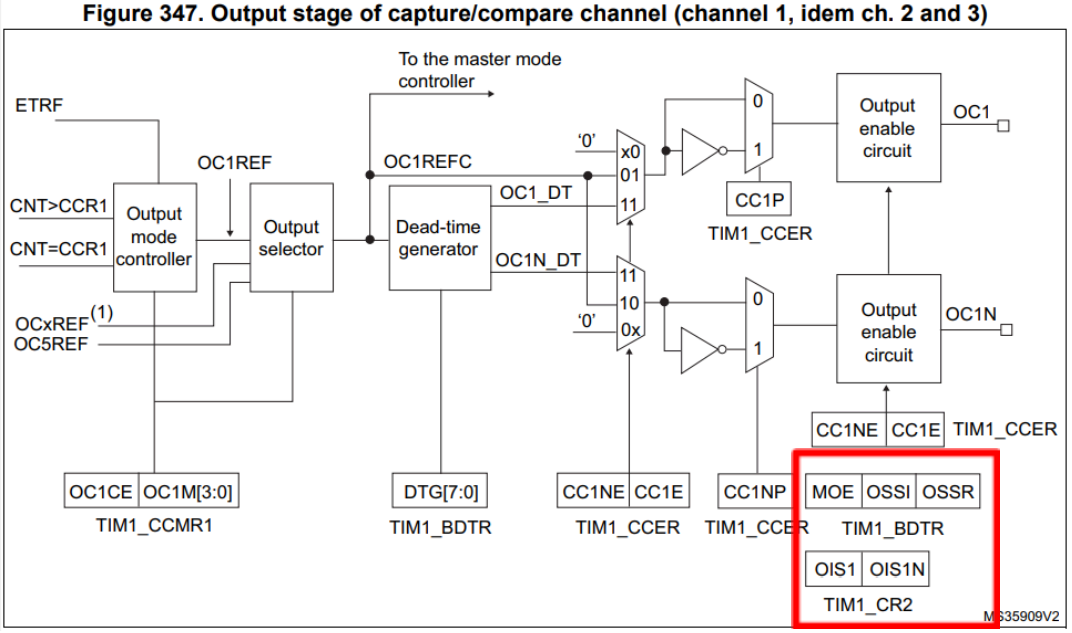
Figure 345. Capture/compare channel (example: channel 1 input stage)



No **estágio de comparação e saída**, o valor armazenado no registrador de captura/comparação `TIMx_CCRn` é continuamente comparado com o valor atual do contador TIMx_CNT. Quando ocorre uma correspondência (em inglês, *match*), o sinal de saída de referência `OCnREF` é gerado de acordo com o modo de comparação de saída configurado pelos *bits* `OCnM[3:0]` no registrador `TIMx_CCMRm` (alternativo). Este sinal `OCnREF` serve como base para a geração do sinal de saída principal `OCn` (disponível nos canais 1 a 4) e, para os canais 1 a 4, de seu sinal complementar `OCnN`. A ativação das saídas OCn e OCnN é controlada individualmente pelos *bits* `CCnE` e `CCnNE`, respectivamente, presentes no registrador `TIMx_CCER`. A polaridade dos sinais de saída finais `OCn` e `OCnN`, que define se o nível ativo é alto ou baixo, é configurada pelos *bits* `CCnP` e `CCnNP`, respectivamente, no registrador TIMx_CCER, influenciando o estado final dos pinos de saída.

Quando um canal (exceto canais 5 e 6 para captura) está configurado para captura de entrada, um evento de captura ocorre quando uma borda ativa é detectada no pino de entrada correspondente (TIx). Nesse momento, o valor atual do contador (TIMx_CNT) é capturado no registrador de captura/comparação correspondente (TIMx_CCRx). O flag de interrupção do canal (CCxIF no registrador TIMx_SR) é setado. Um flag de overcapture (CCxOF) também pode ser setado se ocorrerem duas capturas consecutivas sem que o flag CCxIF tenha sido limpo

Figure 347. Output stage of capture/compare channel (channel 1, idem ch. 2 and 3)



Adicionalmente aos temporizadores TIM2, TIM3, TIM4 e TIM5, os temporizadores TIM1 e TIM8 dispõem de circuitos de controle suplementares, conforme ilustrado na figura com um quadrado vermelho. O sinal de saída pode ser direcionado para um circuito gerador de *dead-time* (DTG), cuja função é inserir um intervalo de tempo programável entre as transições dos sinais complementares, prevenindo assim curtos-circuitos em aplicações como o controle de motores. O DTG é configurado através do campo de *bits* `DTG[7:0]` no registrador `TIMx_BDTR`. Para que as saídas OCn e OCnN sejam habilitadas e efetivamente emitidas, o *bit* `MOE` (do inglês *Main Output Enable*) no registrador TIMx_BDTR deve ser configurado como '1'. Para definir o estado dos sinais complementares quando o temporizador está inativo (por exemplo, quando MOE=0), os *bits* `OISx` (do inglês *Output Idle state* para OCx) e OISxN (do inglês *Output Idle state* para OCxN) no registrador `TIMx_CR2` permitem configurar o nível de saída desejado. Por exemplo, para garantir que `OC1` e `OC1N` estejam em nível baixo quando inativos, os *bits* `TIMx_CR2_OIS1` e `TIMx_CR2_OIS1N` devem ser configurados como '0'. Durante a operação normal (quando `MOE`=1), os *bits* OSSI (do inglês *Off-state selection for Idle mode*) e OSSR (do inglês *Off-state selection for Run mode*) no registrador TIMx_BDTR controlam o comportamento das saídas em estados inativo. Se `OSSI` e `OSSR` forem definidos como '1', respectivamente, os sinais `OCx` e `OCxN` serão forçados ao estado inativo definido pelos *bits* `OISx` e `OISxN` quando o temporizador estiver inativo (por exemplo, devido a um evento de *break* ou escrita de *software* para `MOE`=0 para OSSI) ou quando estiver ativo mas os sinais de referência `OCxREF` estiverem inativos para `OSSR`.

A configuração de um canal do temporizador TIM1 para as funções de **Captura de Entrada**, Comparação de Saída e Modulação por Largura de Pulso é centralizada no uso dos registradores `TIMx_CCMR1` e `TIMx_CCMR2`. Estes registradores são essenciais pois definem a funcionalidade de cada canal individualmente, de acordo com a configuração

específica dos seus *bits* para cada modo de operação. No modo de Captura de Entrada, o registrador TIMx_CCMRn (onde 'n' representa 1 ou 2, dependendo do canal) é configurado para capturar eventos nos sinais de entrada. A seleção do pino de entrada apropriado (TI1 ou TI2 para canais 1 e 2, TI3 ou TI4 para canais 3 e 4, respectivamente) é realizada através dos *bits* CCnS[1:0] nos registradores TIMx_CCMR1 e TIMx_CCMR2. Adicionalmente, estes registradores permitem configurar o filtro digital (ICnF[3:0]) e o *prescaler* de entrada (ICnPSC[1:0]) para o sinal de entrada capturada.

Para o modo de **Comparação de Saída**, a configuração alternativa do TIMx_CCMRx define o comportamento da saída com base na comparação entre o valor do contador (TIMx_CNT) e um valor programado no registrador de captura/comparação correspondente (TIMx_CCRn). O controle da saída é realizado pelos *bits* OCxM[3:0], que permitem programar diferentes ações no pino de saída quando a comparação é bem-sucedida. Estas ações incluem manter o nível atual (*frozen* - 0000), forçar o pino a um nível ativo (0001) ou inativo (0010), ou alternar seu estado (*toggle* - 0011) no momento da correspondência. No modo PWM, a configuração alternativa do TIMx_CCMRm ajusta o canal para gerar um sinal PWM. Os *bits* OCxM[3:0] são configurados para os modos PWM específicos (PWM modo 1 - 0110 ou PWM modo 2 - 0111), definindo o comportamento do sinal de saída (OCxREF) em relação à comparação entre o contador e o registrador TIMx_CCRn. No modo PWM 1 em contagem crescente, OCxREF é ativo enquanto TIMx_CNT < TIMx_CCRn e inativo caso contrário. No modo PWM 2 em contagem crescente, OCxREF é inativo enquanto TIMx_CNT < TIMx_CCRn e ativo caso contrário. O ciclo de trabalho do sinal PWM é determinado pela relação entre o valor em TIMx_CCRn e o valor de *auto-reload* (TIMx_ARR). É importante notar que para o correto funcionamento do PWM, o registrador pré-carga correspondente deve ser habilitado (*bit* OCxPE em TIMx_CCMRm) e, eventualmente, a pré-cargga do *auto-reload* (*bit* ARPE em TIMx_CR1). Antes de iniciar o contador, os registradores devem ser inicializados gerando um evento de atualização (*bit* TIMx_EGR_UG). Observe que os *bits* CCxS[1:0] nos registradores TIMx_CCMR1 e TIMx_CCMR2 só podem ser modificados quando o canal correspondente está desativado, ou seja, quando o *bit* CCxE (do inglês *Capture/Compare x output enable*) no registrador TIMx_CCER está definido como '0'. Esta precaução evita configurações inesperadas durante a operação ativa do canal.

Os temporizadores TIM1 e TIM8 geram interrupções através de seus canais de captura/comparação (CCn). Essas interrupções ocorrem em duas situações principais:

- Captura de Evento (*Input Capture*)**:** Quando um evento é capturado pelo canal.
- Comparação de Valores (*Output Compare*/PWM)**:** Quando o contador (TIMx_CNT) atinge o valor armazenado no registrador de captura/comparação (TIMx_CCRn) do canal.

Em ambas as situações, a *flag* de interrupção correspondente (ex: `CC1IF` para o canal 1) é ativada no registrador de estado (`TIMx_SR`). Para que a interrupção seja efetivamente enviada ao núcleo do microcontrolador, duas condições devem ser atendidas:

- A máscara de interrupção do canal (`TIMx_DIER_CCnIE`) deve estar habilitada.
- As linhas IRQ27 (TIM1) ou IRQ46 (TIM8) do NVIC, responsáveis pelas interrupções de captura/comparação, devem estar habilitadas.

Por exemplo, para habilitar a interrupção do canal 1 do TIM8, configure o *bit* `CC1IE` para '1' e habilite a IRQ46 no NVIC. O mesmo princípio se aplica aos demais canais (`CC2IE`, `CC3IE`, etc.). Além disso, as interrupções podem ser geradas por *software* ao escrever '1' no *bit* `TIMx_EGR_CCnG` do registrador de geração de eventos. Isso também ativa a *flag* `CCnIF` no `TIMx_SR`.

tim1_cc_it	34	27	TIM1_CC	TIM1 capture / compare interrupt	0x0000 00AC

tim8_cc_it	53	46	TIM8_CC	TIM8 capture / compare interrupts	0x0000 00F8

Modo *Output Compare*

Na função *Output Compare*, quando o valor do contador `TIMx_CNT` se iguala ao valor definido no registrador `TIMx_CCRn` do canal n, as seguintes ações são executadas, de acordo com o Manual de Referência:

- **Geração do Sinal de Saída**: O pino de saída correspondente é configurado com um valor programável, determinado pelo modo de comparação de saída (definido pelos *bits* `TIMx_CCMRm_OCnM`) e pela polaridade de saída (configurada pelo *bit* `TIMx_CCER_CCnP`). O pino pode ser configurado para manter seu nível atual (`OCnM`=0b0000), ser definido como ativo (`OCnM`=0b0001), ser definido como inativo (`OCnM`=0010), ou alternar seu estado (`OCnM`=0011) em resposta à correspondência.

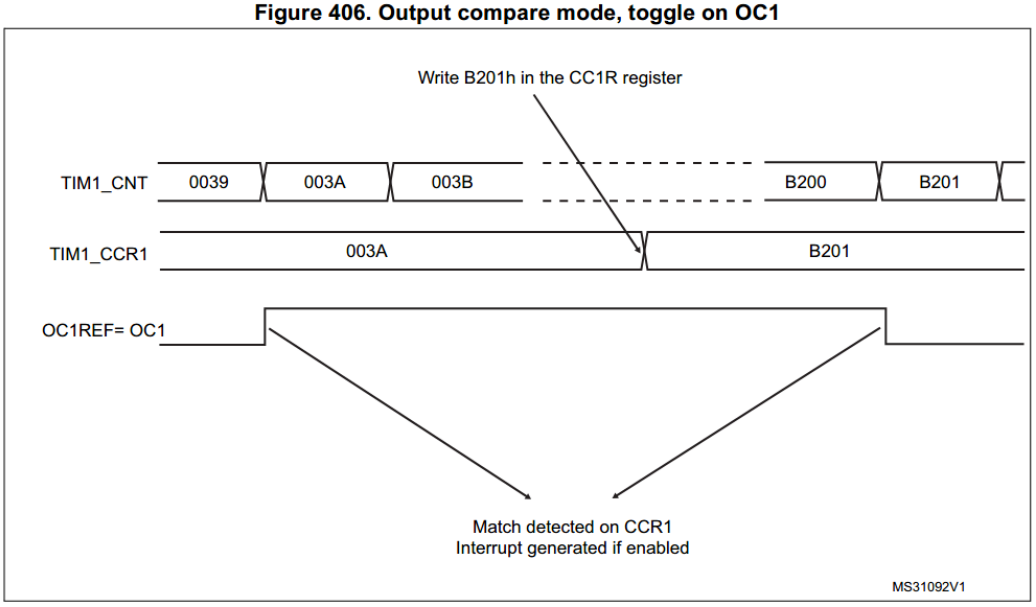
- **Ativação do Estado de Interrupção**: A *flag* é acionado no registrador de estado de interrupção (*bit* `TIMx_SR_CCnIF` no registrador `TIMx_SR`) quando ocorre uma correspondência entre `TIMx_CNT` e `TIMx_CCRn`.

- **Geração de Interrupção**: Se a máscara de interrupção correspondente estiver ativada (*bit* `TIMx_DIER_CCxIE`), uma interrupção é gerada.

- **Solicitação de DMA** (do inglês *Direct Memory Access*): Se o *bit* de habilitação correspondente estiver definido (*bit* `TIMx_DIER_CCnDE` em '1' e o *bit* `TIMx_CR2_CCDS` estiver configurado para seleção de solicitação DMA), uma solicitação DMA é gerada.

A figura a seguir, extraída do Manual de Referência, ilustra a forma de onda OC1REF gerada pela função *Output Compare* com `TIM1_CCER_CC1M` setado em 0b0001 no canal 1.

Figure 406. Output compare mode, toggle on OC1



Os registradores `TIMx_CCRn` podem ser configurados por pré-carga ou não, dependendo do estado do *bit* `TIMx_CCMRm_OCnPE`. Quando o *bit* `TIMx_CCMRm_OCnPE` está ativado, o registrador `TIMx_CCRn` é atualizado somente durante um evento de atualização (UEV). Se o *bit* estiver desativado, o registrador `TIMx_CCRn` é atualizado imediatamente. No modo de comparação de saída, o evento de atualização UEV não afeta a saída OCnREF nem OCn. A resolução de tempo é baseada na contagem do contador, e o modo de comparação de saída pode ser configurado para gerar um único pulso (no modo *One-Shot*). O *bit* de estado `TIMx_SR_CCnIF` pode ser limpo por *software*, escrevendo '0'.

Uma sutileza importante reside na diferença entre como os valores dos registradores `TIMx_ARR`, `TIMx_PSC` e `TIMx_CCRn` são programados, influenciada pela natureza da contagem (que geralmente começa em zero) e pela implementação do *hardware* de comparação. Os registradores `TIMx_ARR` e `TIMx_PSC` definem o limite superior da "contagem". Como a contagem inicia em zero, o valor N-1 é programado para obter N ciclos. O registrador `TIMx_CCRn`, por sua vez, compara diretamente com o valor atual do contador, exigindo a programação direta do valor desejado M para acionar o evento correspondente. Este evento ocorre na transição de M-1 para M, indicando a conclusão da contagem de M-1 ciclos.

Modo *Input Capture*

No modo de **captura de entrada**, os registradores de captura/comparação `TIMx_CCRn` são usados para armazenar o valor do contador, `TIMx_CNT`, após detectar uma transição ativa no sinal de entrada do canal ICn correspondente. Quando uma captura ocorre, o *bit* de estado `TIMx_SR_CnIF` (no registro `TIMx_SR`) é ativado, e uma interrupção ou solicitação DMA pode ser gerada se estiverem habilitadas. Se uma nova captura ocorrer enquanto o *bit* de estado `TIMx_SR_CnIF` ainda estiver ativo, o *bit* de estado de sobrecaptura (em inglês, *over-capture*) `TIMx_SR_CnOF` será ativado. O *bit* de estado `TIMx_SR_CnIF` pode ser limpo por *software*, escrevendo '0' ou lendo os dados capturados do registro `TIMx_CCRn`. O *bit* de estado `TIMx_SR_CnOF` é limpo escrevendo '0'.

O procedimento a seguir, traduzido do Manual de Referência, ilustra como se configura um canal para capturar o valor do contador em `TIMx_CCR1` quando a entrada TI1 apresenta uma transição ativa de subida:

1. **Selecione a Fonte de Entrada TI1**: Configure a fonte TI1 adequada (interna ou externa) usando os *bits* `TIMx_TILSEL_TI1SEL[3:0]` no registrador `TIMx_TISEL`.

2. **Configure a Entrada Ativa**: Defina `TIMx_CCR1` para estar vinculado à entrada TI1, configurando os *bits* `TIMx_CCMR1_CC1S` como 0b01 no registrador `TIMx_CCMR1`. Quando `TIMx_CCMR1_CC1S` for diferente de 0b00, o canal será configurado para entrada e o registrador `TIMx_CCR1` se tornará somente de leitura.

3. **Programe a Duração do Filtro de Entrada**: Ajuste o filtro de entrada para o sinal conectado ao temporizador, configurando os *bits* `TIMx_CCMRx_ICxF` no registrador. Por exemplo, se o sinal não for estável por pelo menos 5 ciclos de *clock*, programe um filtro mais longo. Para validar uma transição em TI1 quando 8 amostras consecutivas com o novo nível forem detectadas (amostradas na frequência f_{DTS}, que é proporcional à temperatura medida pelo sensor de temperatura digital integrado ao microcontrolador), configure os *bits* `TIMx_CCMR1_IC1F` como 0b0011 no registrador `TIMx_CCMR1`.

4. **Selecione a Borda da Transição Ativa**: Configure a borda ativa para TI1escrevendo '0' nos *bits* TIMx_CCER_CC1P e TIMx_CCER_CC1NP no registrador TIMx_CCER para capturas na borda de subida.

5. **Programe o *Prescaler* de Entrada**: Para capturar em cada transição válida, desative o *prescaler* escrevendo 0b00 nos *bits* `TIMx_CCMR1_IC1PS` no registrador `TIMx_CCMR1`.

6. **Habilite a Captura do Contador**: Ative a captura configurando o *bit* `TIMx_CCER_CC1E`.

7. **Configure Interrupção e Solicitação DMA**: Se necessário, habilite a interrupção configurando o *bit* `TIMx_DIER_CC1IE` no registrador `TIMx_DIER` e/ou a solicitação DMA configurando o *bit* `TIMx_DIER_CC1DE` no mesmo registrador.

Quando uma captura de entrada ocorre:

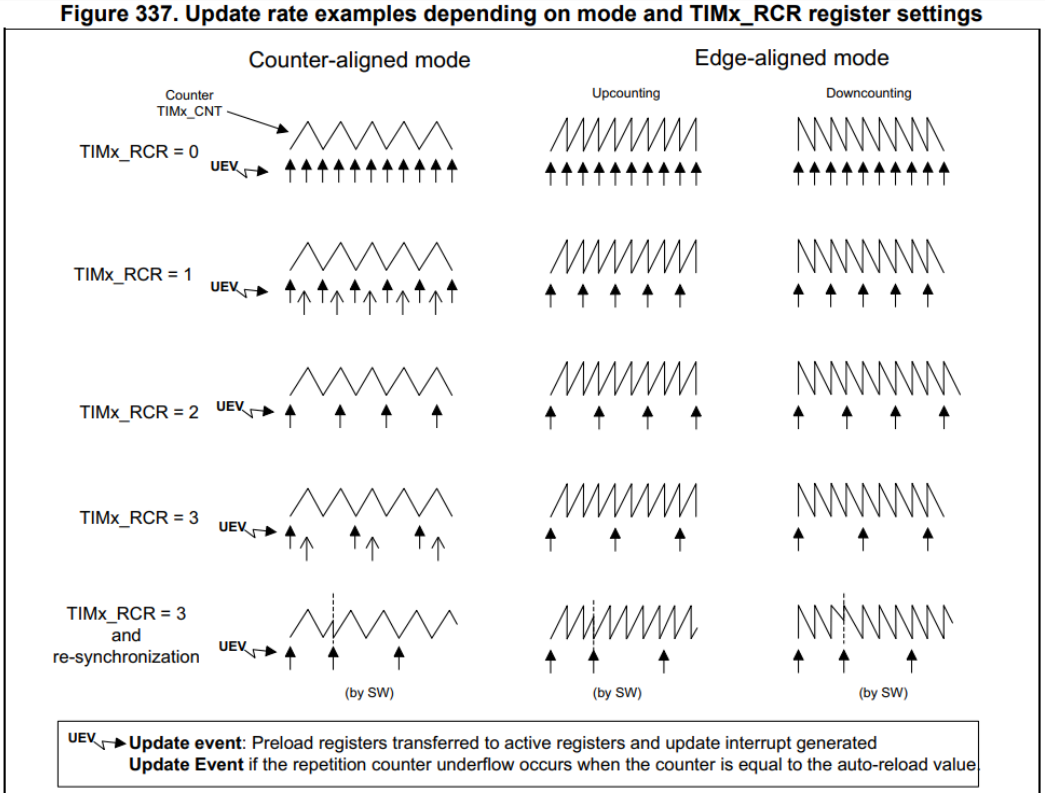
- O registrador `TIMx_CCR1` armazena o valor do contador no momento da transição ativa.
- O *bit* de estado de captura `TIMx_SR_CC1IF` é ativado. O *bit* de estado de sobrecaptura `TIMx_SR_CC1OF` também será ativado se houver pelo menos duas capturas consecutivas antes que o *bit* de estado `TIMx_SR_CC1IF` seja limpo.
- Uma interrupção é gerada com base na configuração do *bit* `TIMx_DIER_CC1IE`.
- Uma solicitação DMA é gerada com base na configuração do *bit* `TIMx_DIER_CC1DE`.

Para gerenciar **sobrecapturas**, recomenda-se ler os dados do contador antes de limpar o *bit* de estado de sobrecaptura para evitar perder qualquer sobrecaptura que possa ocorrer entre a leitura do *bit* de estado e a leitura dos dados.

Modo PWM

Os canais de PWM dos temporizadores TIM1 e TIM8 do STM32H7A3ZIT-Q oferecem funcionalidades versáteis para geração de sinais PWM com diferentes configurações. No modo PWM, os sinais de saída podem ser configurados para operar em dois modos principais: PWM *Mode* 1 e PWM *Mode* 2. No **PWM *Mode* 1**, o sinal é no nível alto quando o contador é menor ou igual ao valor predefinido no registrador de captura/comparação `TIMx_CCRn`, enquanto no **PWM *Mode* 2**, o sinal é no nível alto quando o contador é maior ou igual ao valor de comparação. Além disso, o temporizador pode ser configurado para modos de **alinhamento de borda** (em inglês, *edge-aligned*) ou **alinhamento central** (em inglês, *center-aligned*). No modo *edge-aligned*, o sinal PWM é gerado com uma única borda de início e término por ciclo de temporização, enquanto no modo *center-aligned*, o sinal PWM tem bordas de início e término centralizadas em torno de um ponto médio no ciclo. A figura extraída do Manual de Referência ilustra os diversos modos de operação do PWM e demonstra como as diferentes configurações do registrador de repetição `TIMx_RCR` influenciam a geração de eventos de atualização.

Figure 337. Update rate examples depending on mode and TIMx_RCR register settings



Os seguintes registradores são envolvidos na função PWM de um canal:

- TIMx_CCMRm (alternativo): Configura o modo de operação do canal. Os *bits* TIMx_CCMRm_OCnM definem o modo PWM (1 para PWM *Mode* 1, 2 para PWM *Mode* 2).
- TIMx_CCRn: Define o valor de comparação utilizado para gerar o sinal PWM. A largura do pulso PWM é determinada pela comparação entre o valor do contador e o valor no TIMx_CCRn.
- TIM_CR1: Habilita o temporizador e configura opções gerais, como o modo de contagem.
- TIMx_CR2: Configura a saída principal (TIMx_CR2_OISn) e a saída complementar (TIMx_CR2_OISnN) para o canal, controlando o estado do pino de saída quando inativo.
- TIMx_BDTR: Configura o controle de saída, incluindo o habilitação de saída principal (TIMx_BDTR_MOE), e seleção do estado de saída durante o modo inativo (TIMx_BDTR_OSSI e TIMx_BDTR_OSSR).

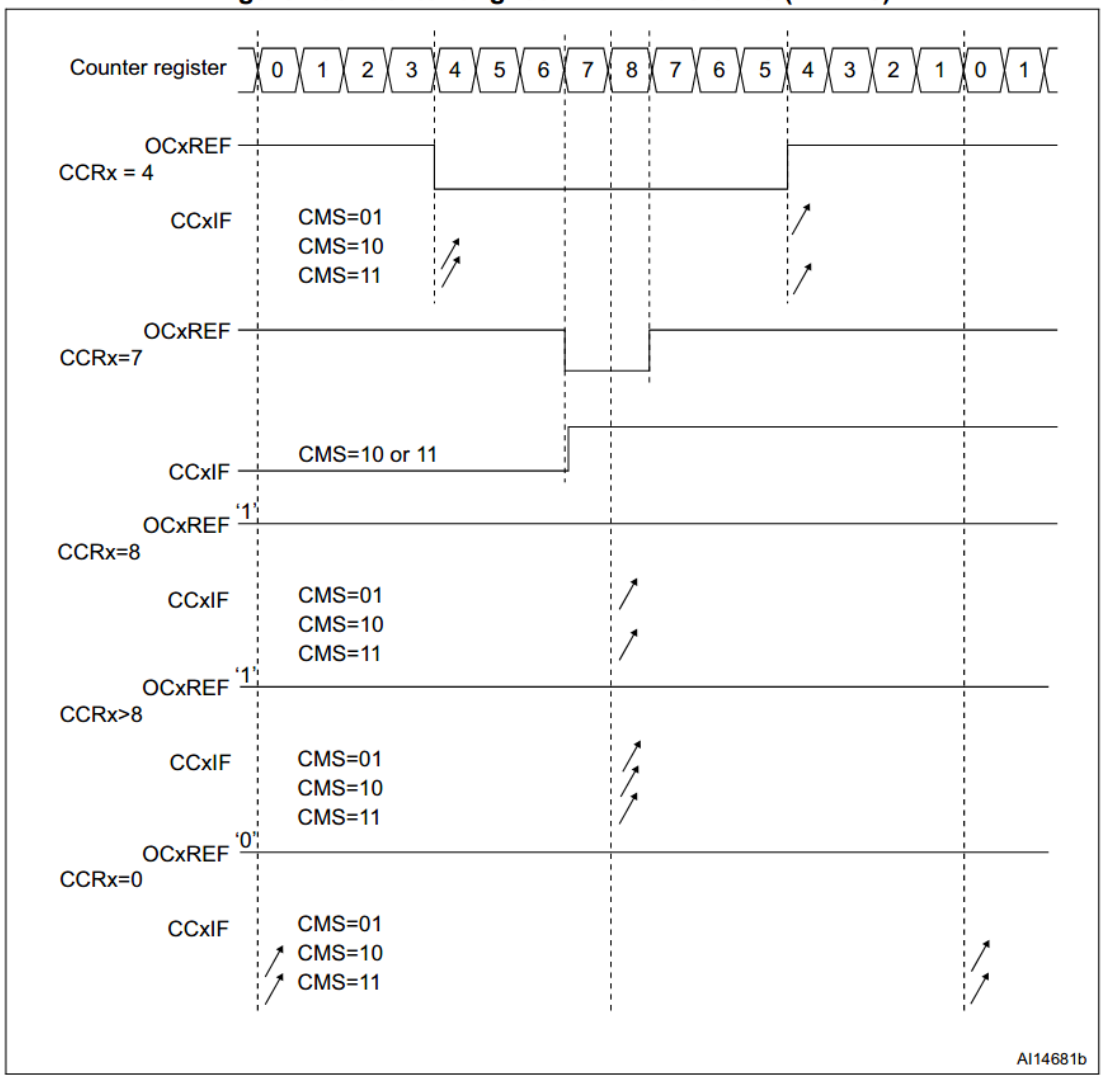
Para configurar a função PWM nos canais, são necessárias as seguintes etapas:

- **Selecione o modo PWM** desejado no TIMx_CCMRn usando os *bits* TIMx_CCMRm_OCnM.
- **Defina a largura do pulso PWM** no TIMx_CCRn.

- **Habilite e configure o temporizador** com os registradores `TIMx_CR1` (`TIMx_CR1_DIR` para direção de contagem; `TIMx_CR1_CMS` para tipo de alinhamento; `TIMx_CR1_UDIS` para atualização de evento; `TIMx_CR1_CEN` para hailitação do canal) e `TIMx_CR2`.
- **Configure o controle de saída** e o comportamento do pino com o registrador `TIMx_BDTR`.

A figura a seguir, extraída do Manual de Referência, ilustra as formas de onda geradas pela função PWM com alinhamento central, como também o instante em que o *bit* de estado `CCnIF` fique em '1', para diferentes valores configurados em `TIMx_CCRn`, mantendo fixo o valor no registrador de *auto-reload* `TIMx_ARR`.

Figure 408. Center-aligned PWM waveforms (ARR=8)

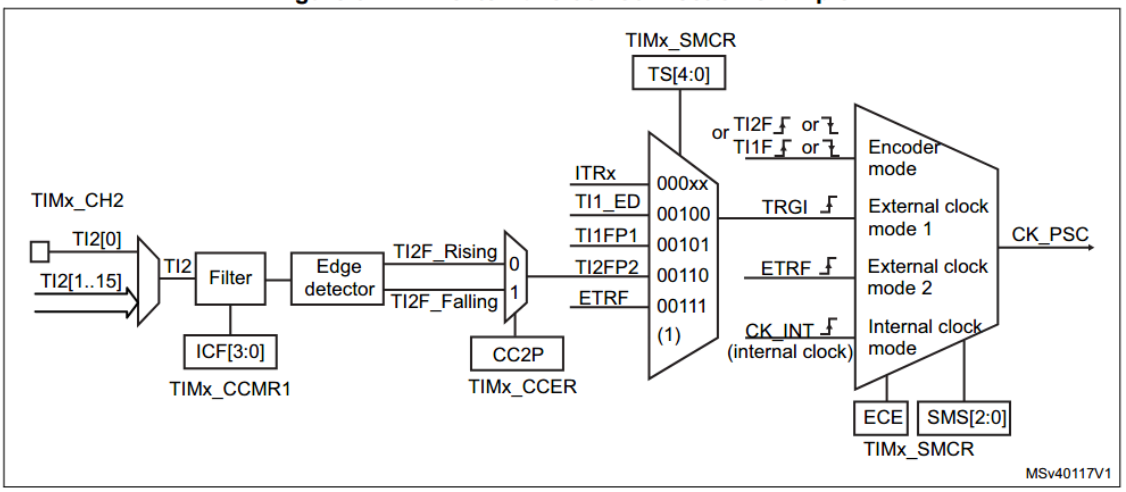


Modo de Relógio Externo

Os temporizadores TIM1 e TIM8 oferecem suporte a dois modos principais de relógio externo: *External Clock Mode 1* e *External Clock Mode 2*. Cada um desses modos é projetado para atender a diferentes necessidades de sincronização e contagem de eventos baseados em sinais externos. O *External Clock Mode 1* é útil quando se deseja sincronizar a contagem do temporizador com eventos periódicos ou assíncronos provenientes de um dispositivo externo. Já o *External Clock Mode 2* é adequado para situações em que o temporizador precisa operar como um contador síncrono impulsionado por um *clock* externo. Além disso, em ambos os modos, é possível configurar a polaridade do sinal de entrada, garantindo compatibilidade com diferentes fontes de *clock* externo. Dessa forma, a existência desses dois modos amplia a flexibilidade dos temporizadores TIM1 e TIM8, permitindo que eles se adaptem a uma variedade maior de aplicações que exigem sincronização precisa com sinais externos.

No modo *External Clock Mode* 1, selecionado ao configurar os *bits* `SMS`=111 no registrador TIMx_SMCR, o sinal de *clock* para o contador (`TIMx_CNT`) é derivado de um pino de entrada de um dos canais de captura/comparação (TIn). Especificamente, os pinos TI1 ou TI2 são associados aos canais 1 e 2. O sinal de entrada TIn passa por um detector de borda, onde as bordas detectadas atuam como pulsos de *clock* para o contador. A polaridade da borda ativa (subida ou descida) é definida pelos *bits* `CCnP` e `CCnNP` no registrador `TIMx_CCER`. Adicionalmente, um filtro de entrada digital (`ICnF`), configurado em `TIMx_CCMRm`, pode ser aplicado ao sinal TIn antes da detecção de borda. Essencialmente, este modo reutiliza a infraestrutura de entrada dos canais de captura/comparação para fornecer um *clock* externo ao contador. Graças à arquitetura dos temporizadores de TIM1/TIM8, que permite interconexões internas, o sinal de alguns módulos do STM32H7A3 pode ser roteado para os pinos TIn através dos *bits* `TI2SEL[3:0]` no registrador `TIMx_TISEL`.

Figure 341. TI2 external clock connection example

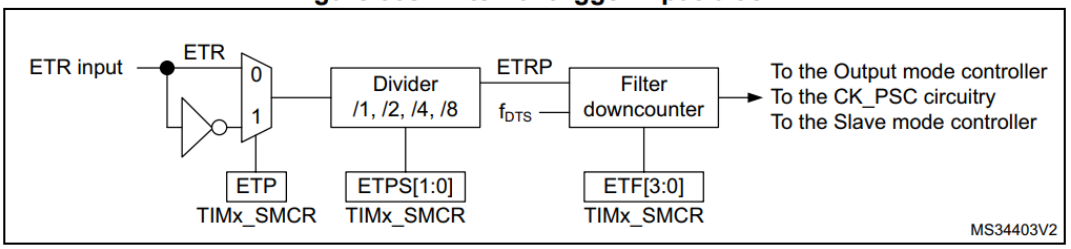


O modo *External Clock Mode 2*, ativado ao configurar o *bit* `TIMx_SMCR_ECE` para '1', utiliza o pino de *trigger* externo dedicado ETR (do inglês, *External Trigger*) para fornecer o sinal de relógio ao contador. O sinal no pino ETR passa por um circuito de condicionamento dedicado antes de alimentar o contador. Este circuito inclui:

- um filtro digital (ETF), configurado no registrador `TIMx_SMCR`,
- um *prescaler* (ETPS), que pode dividir a frequência do sinal ETR por 1, 2, 4 ou 8, configurado no registrador `TIMx_SMCR`. O sinal após o *prescaler* é chamado de ETRP.
- seleção da polaridade da borda ativa (ETP), também configurada no registrador `TIMx_SMCR`.

O sinal, após a seleção de polaridade, é chamado de ETRF, que serve como o *clock* do contador. Da mesma forma que os sinais gerados por alguns módulos do STM32H7A3, selecionáveis pelos *bits* `TIMx_TISEL_TI2SEL`, podem ser roteados para os canais de entrada dos temporizadores TIM1/TIM8 no *External Clock Mode 1*, a fonte do sinal ETRF pode ser o pino ETR ou outras fontes internas, selecionáveis através dos *bits* `ETRSEL[3:0]` nos registradores `TIM1_AF1` ou `TIM8_AF1` no *External Clock Mode 2*.

Figure 338. External trigger input block



O sinal de gatilho externo condicionado ETRF pode ser roteado para três blocos funcionais nos temporizadores TIM1/TIM8:

- ***CK_PSC circuitry* (Circuito do *clock* do *prescaler*)**: O sinal ETRF serve como a fonte de *clock* do contador (`TIMx_CNT`) quando o temporizador opera no *External Clock Mode 2* (ativado pelo *bit* `TIMx_SMCR_ECE`=1). Cada borda ativa detectada no sinal ETRF incrementa ou decrementa o contador, dependendo da configuração do modo de contagem do temporizador.
- ***Slave mode controller* (Controlador de modo escravo)**: O sinal condicionado ETRP (que leva ao ETRF) é também uma entrada para o *Slave mode controller*. Este bloco permite que o temporizador opere em modos escravos, sincronizando seu funcionamento com outros temporizadores ou eventos externos. Através do *Slave mode controller*, o sinal de *trigger* externo (derivado de ETRF) pode iniciar, parar, resetar ou clockear o contador do temporizador, dependendo do modo escravo configurado pelos *bits* `TIMx_SMCR_SMS`. No contexto específico do *External Clock Mode 2*, o *Slave mode controller* configura o temporizador para usar ETRF como sua fonte de *clock*.

- ***Output mode controller*** **(Controlador de modo de saída)**: Embora não esteja diretamente no caminho do sinal ETRF como *clock* para o contador no *External Clock Mode 2*, o sinal ETRF pode influenciar o comportamento do *Output mode controller* através do recurso de *Output Compare clear enable* (*bit* `TIMx_CCMR1_OCnCE`). Quando habilitado, um nível alto no sinal de entrada de *trigger* externo filtrado (ETRF) pode limpar o sinal de referência da saída (OCnREF) dos canais, afetando assim a saída nos modos *Output Compare* e PWM.

O Manual de Referência fornece um procedimento para configurar um contador progressivo para contar em resposta a uma borda de subida no pino TI2 no *External Clock Mode 1*:

1. Selecione a fonte apropriada para TI2 (interna ou externa) usando os *bits* `TI2SEL[3:0]` no registrador `TIMx_TISEL`.

2. Configure o canal 2 para detectar bordas de subida no pino TI2, escrevendo 0b01 em `TIMx_CCMR1_CC2S` do registrador `TIMx_CCMR1`.

3. Configure a duração do filtro de entrada escrevendo os *bits* `IC2F[3:0]` no registrador `TIMx_CCMR1` (se nenhum filtro for necessário, mantenha `TIMx_CCMR1_IC2F`=0b0000).

4. Selecione a polaridade da borda de subida escrevendo `TIMx_CCER_CC2P`=0 e `TIMx_CCER_CC2NP`=0 no registrador `TIMx_CCER`.

5. Configure o temporizador no modo de relógio externo 1, escrevendo `TIMx_SMCR_SMS`=0b111 no registrador `TIMx_SMCR`.

6. Selecione TI2 como a fonte de entrada de *trigger* escrevendo `TIMx_SMCR_TS`=0b00110 no registrador `TIMx_SMCR`.

7. Habilite o contador escrevendo `TIMx_CR1_CEN`=1 no registrador `TIMx_CR1`.

Para configurar o contador no*External Clock Mode 2*, em que o *clock* do contador é sincronizado com o sinal dedicado ETR, o Manual de Referência recomenda o seguinte procedimento:

1. Selecione a fonte apropriada para o ETR através dos *bits* `ETRSEL`[3:0] no registrador `TIM1_AF1` (para TIM1) ou TIM8_AF1 (para TIM8). Por padrão, o pino ETR é a fonte. Outras fontes podem incluir saídas de comparadores ou *Analog Watchdogs* de ADCs.
2. Configure o filtro digital no sinal ETR se necessário, escrevendo nos *bits* `ETF`[3:0] no registrador `TIMx_SMCR`. Se nenhum filtro for necessário, mantenha `ETF`[3:0] = 0000.

3. Configure o *prescaler* do gatilho externo através dos *bits* `ETPS`[1:0] no registrador `TIMx_SMCR`. Este *prescaler* pode dividir a frequência do sinal ETR por 1, 2, 4 ou 8. Por exemplo, para contar a cada duas bordas de subida em ETR, escreva `ETPS`[1:0] = 01.
4. Selecione a polaridade da borda ativa no pino ETR usando o *bit* `ETP` no registrador `TIMx_SMCR`. Escreva `ETP` = 0 para detecção de borda de subida e `ETP` = 1 para detecção de borda de descida.
5. Habilite o modo de relógio externo 2 (em inglês, *External Clock Mode 2*) escrevendo `ECE` = 1 no registrador `TIMx_SMCR`.
6. Habilite o contador escrevendo `CEN` = 1 no registrador `TIMx_CR1`.

Note que configurar o *bit* `TIMx_SMCR_ECE` como '1' faz com que o temporizador opere de maneira equivalente ao modo 1, apenas com o sinal de *trigger* externo ETR, conectado ao pino ETRF, que está mapeado no pino PE7.

STM32CubeMX

Uma das características mais marcantes do STM32CubeIDE é o STM32CubeMX, uma ferramenta gráfica que permite configurar os sinais de relógio dos módulos de forma intuitiva e visual. Com o STM32CubeIDE e o STM32CubeMX, a configuração dos sinais de relógio dos módulos STM32 se torna uma tarefa mais simples e eficiente, permitindo que os desenvolvedores se concentrem na implementação das funcionalidades principais de seus aplicativos.

O sistema de *clock* no STM32H7A3 começa com uma fonte de *clock* principal, como o oscilador de alta velocidade interno (HSI) ou o oscilador de cristal externo (HSE). A partir da fonte de *clock* principal, os sinais de *clock* são divididos por divisores de frequência para gerar *clocks* com frequências diferentes para os diferentes barramentos e periféricos. Os barramentos (como AHB, APB1 e APB2) e os periféricos recebem seus sinais de *clock* dos divisores de frequência, formando uma estrutura hierárquica onde os *clocks* de nível superior alimentam os *clocks* de nível inferior.

Através do STM32CubeMX, os desenvolvedores podem:

- **selecionar fontes de *clock*:** Escolher entre osciladores internos e externos, como HSI e HSE, para fornecer o *clock* principal do sistema.
- **configurar divisores de *clock*:** Ajustar os divisores de frequência para gerar sinais de *clock* com as frequências desejadas para os diferentes barramentos e periféricos.
- **visualizar a estrutura de *clock*:** A interface gráfica exibe a estrutura hierárquica dos *clocks*, facilitando a compreensão e a configuração os diferentes clocks e divisores de frequência de forma organizada e intuitiva.

- **gerar código de inicialização:** O STM32CubeMX gera automaticamente o código de inicialização do *clock*, economizando tempo e reduzindo a probabilidade de erros.