DISCIPLINA EA701

Introdução aos Sistemas Embarcados

ROTEIRO 5: Temporizadores e Camada de Abstração de *Hardware*

TIM6/TIM7, SYSTICK, RTC e *WATCHDOG*

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

FEEC / UNICAMP

Revisado em agosto de 2024

INTRODUÇÃO

A gestão do tempo em sistemas embarcados representa um elemento crítico que exerce impacto direto sobre a precisão, confiabilidade e eficiência desses sistemas. A habilidade de medir e controlar o tempo de maneira precisa, bem como coordenar atividades de forma eficaz, é um requisito fundamental em diversos setores, englobando automotivo, médico, aeroespacial, eletrônico e outros. Essa competência possibilita a sincronização de eventos, a geração de sinais periódicos e a medição precisa de intervalos de tempo. Na gestão do tempo de sistemas embarcados, existem duas tecnologias essenciais de temporização: os **relógios digitais**, ou temporizadores de *software*, e os **relógios analógicos**, ou temporizadores de *hardware*. Os relógios mecânicos são um exemplo clássico de temporizadores analógicos, onde a precisão é determinada pela qualidade das engrenagens e da construção do relógio. Em comparação com os relógios digitais, eles são geralmente menos precisos e requerem manutenção periódica para funcionar de maneira confiável. Com a ascensão da eletrônica digital, os relógios digitais, que utilizam circuitos de temporizadores digitais, tornaram-se mais comuns devido à sua precisão e facilidade de uso. A estabilidade e precisão dos temporizadores digitais são cruciais em sistemas microcontrolados, e

para garantir a precisão e estabilidade dos temporizadores digitais integrados em microcontroladores, esses temporizadores devem utilizar sinais de *clock* gerados no microcontrolador a partir do *clock* geral, fornecendo uma base sólida para a contagem de tempo e a execução precisa de operações temporizadas.

Os temporizadores podem variar em tipo e função, incluindo temporizadores de propósito geral, temporizadores avançados e temporizadores de *watchdog*. **Temporizadores de propósito geral** são usados para medir intervalos de tempo e gerar eventos periódicos, enquanto **temporizadores avançados**, que frequentemente possuem recursos como controle de PWM (do inglês *Pulse Width Modulation*), captura e comparação, são projetados para aplicações que exigem alta precisão e controle detalhado. Os temporizadores de *watchdog*, por outro lado, são críticos para garantir a confiabilidade do sistema, monitorando a operação do microcontrolador e reiniciando-o em caso de falha. Adicionalmente, o RTC (do inglês *Real-Time Clock*) é um temporizador especializado em manter a contagem precisa do tempo real, incluindo horas e datas, mesmo quando o microcontrolador está em modo de baixo consumo ou desconectado da fonte principal de energia.

Microcontroladores, como os da série STM32, frequentemente incorporam uma ampla gama de temporizadores digitais integrados para atender a diversas necessidades de aplicação. Essa variedade permite que o microcontrolador se adapte a diferentes requisitos, como controle preciso de motores, geração de sinais complexos, medição precisa de intervalos de tempo e gerenciamento de eventos. Além disso, temporizadores especializados, como os de *watchdog* e o RTC, garantem a confiabilidade do sistema e a manutenção contínua do tempo, mesmo em modos de baixo consumo ou quando o dispositivo está desligado. Essa diversidade proporciona aos desenvolvedores a flexibilidade necessária para escolher o temporizador mais adequado para cada projeto, otimizando a eficiência e a confiabilidade do sistema embarcado.

PROJETOS-EXEMPLO

Os temporizadores de microcontroladores são componentes essenciais para o controle preciso de sistemas embarcados, desempenhando um papel fundamental em uma ampla gama de aplicações, desde a medição de intervalos de tempo até a geração de sinais periódicos. Mas você já parou para pensar sobre como esses temporizadores funcionam de fato? Você tem alguma ideia do princípio básico que faz com que eles desempenhem esse papel tão? E como você imagina a sua programação? Será que é tão simples quanto parece ou envolve camadas mais complexas de controle e precisão? Nesta seção, vamos explorar juntos a programação de diferentes classes de temporizadores integrados no microcontrolador STM32H7A3. Vamos entender como esses temporizadores podem ser configurados e utilizados em projetos práticos, em aplicações variadas, como controle de motores, geração de sinais PWM, monitoramento de eventos e muito mais.

Projeto com *timers* usando CMSIS

Vamos criar um projeto com um *timer* gerando uma interrupção periódica para fazer o LED verde piscar a uma frequência fixa de 1Hz.

1. Crie um projeto com o nome "Timer_CMSIS", com suporte ao CMSIS, como foi feito em roteiros anteriores (adicionando as pastas com os arquivos de suporte `stm32h7a3xxq.h` e `core_cm7.h` e colocando o "include" do arquivo `stm32h7a3xxq.h` no "main.c").

2. Vamos inicialmente adicionar a configuração de PB0 como saída para acionar o LED verde. No início da função "main", adicione o seguinte código:

```
int main(void)
{
    //Inicializa GPIOB, pino 0 (PB0)
      RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk; // GPIOB clock enable
      // 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
      for (;;);
}
```

3. Agora precisamos configurar um dos *timers* disponíveis para contar 500ms e reiniciar. O reinício do *timer* irá realizar uma interrupção, cujo tratamento é a inversão do estado do LED. Para esta aplicação, os *Timers* mais simples são suficientes. O Manual de Referência apresenta os vários tipos de *Timers* de propósito geral nos capítulos 43 a 45, e os mais básicos no capítulo 46, que são o TIM6 e o TIM7. Vamos usar o TIM6 para gerar a interrupção periódica. O *timer* usa a frequência de *clock* TIMxCLK de seu barramento para realizar a contagem de tempo.

A figura 1 do *Datasheet* do microcontrolador mostra que o TIM6 está ligado ao barramento APB1 através de 16 linhas de *bits*. Ao criar um projeto vazio ("Empty") no STM32CubeIDE, a fonte de sinal de relógio padrão selecionada é o HSI, pois o campo `RCC_CFGR_SW` são resetados em 0b000. Nesse processo, o campo HSIDIV no registrador RCC_CR, como no campo RCC_CDCFGR1_CDDPRE1, é configurado com o valor 1. Portanto, após um *reset* ou quando o microcontrolador é energizado, isso resulta na geração de um *clock* geral com frequência de 64 MHz. O barramento APB1 recebe o sinal da frequência 64MHz, pois o campo RCC_CDCFGR1_HPRE de HSIDIV é configurado como 1, e assim a frequência de entrada do *timer* é 64MHz.

4. Como o *timer* é de 16 *bits*, ele pode contar até, no máximo, 65536 pulsos (de 0 a 65535). Com um *clock* de 64MHz, o tempo máximo de contagem seria de 1024µs. Para ampliar o tempo de contagem para 500ms, usaremos o *prescaler* PSC do *timer*, também de 16 *bits*. Podemos dividir o *clock* por 64000 no *prescaler*, fazendo com que o contador principal receba um *clock* de 1kHz. Assim, cada unidade do *timer* corresponde a 1ms, e podemos configurar o *timer* para contar 500 pulsos.

IMPORTANTE: Tanto o contador do *prescaler* como o do *timer* em si são síncronos, ou seja, ao atingir o valor de contagem definido eles apenas serão zerados no próximo pulso de *clock*. Assim, para contar *n* pulsos, o contador deve contar de 0 até *n* - 1. Por isso, os valores definidos devem ser

os calculados no parágrafo anterior subtraídos de 1. Ou seja, precisamos colocar o valor de 63999 no módulo de contagem do *prescaler* e 499 no módulo de contagem do contador principal.

5. É necessário ativar o *timer*, da mesma forma que ativamos o GPIOB. O TIM6 tem seu *clock gating* ativado pelo *bit* 4 do registrador RCC_APB1LENR.

8.7.43 RCC APB1 clock register (RCC_APB1LENR)

Address offset: 0x148

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
UART8EN	UART7EN	DAC1EN	Res.	CECEN	Res.	Res.	Res.	I2C3EN	I2C2EN	I2C1EN	UART5EN	UART4EN	USART3EN	USART2EN	SPDIFRXEN
rw	rw	rw		rw				rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3EN	SPI2EN	Res.	Res.	Res.	Res.	LPTIM1EN	TIM14EN	TIM13EN	TIM12EN	TIM7EN	TIM6EN	TIM5EN	TIM4EN	TIM3EN	TIM2EN
rw	rw					rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Para isso, teremos que adicionar a seguinte linha de código se usar as macros da interface CMSIS:

```
RCC->APB1LENR |= RCC_APB1LENR_TIM6EN;
```

6. Agora vamos configurar os parâmetros do *timer*. O valor de divisão do *prescaler* é definido no registrador TIM6_PSC e o valor máximo de contagem é definido no registrador de *auto-reload* TIM6_ARR. Ambos os registradores estão mapeados no espaço de endereçamento do processador e são acessados através da struct `TIM_TypeDef` da interface CMSIS definida no arquivo `stm32h7a3xxq.h`.

```
1399 typedef struct
1400 {
1401   __IO uint32_t CR1;         /*!< TIM control register 1,                 Address offset: 0x00 */
1402   __IO uint32_t CR2;         /*!< TIM control register 2,                 Address offset: 0x04 */
1403   __IO uint32_t SMCR;        /*!< TIM slave mode control register,        Address offset: 0x08 */
1404   __IO uint32_t DIER;        /*!< TIM DMA/interrupt enable register,      Address offset: 0x0C */
1405   __IO uint32_t SR;          /*!< TIM status register,                    Address offset: 0x10 */
1406   __IO uint32_t EGR;         /*!< TIM event generation register,          Address offset: 0x14 */
1407   __IO uint32_t CCMR1;       /*!< TIM capture/compare mode register 1,    Address offset: 0x18 */
1408   __IO uint32_t CCMR2;       /*!< TIM capture/compare mode register 2,    Address offset: 0x1C */
1409   __IO uint32_t CCER;        /*!< TIM capture/compare enable register,    Address offset: 0x20 */
1410   __IO uint32_t CNT;         /*!< TIM counter register,                   Address offset: 0x24 */
1411   __IO uint32_t PSC;         /*!< TIM prescaler,                          Address offset: 0x28 */
1412   __IO uint32_t ARR;         /*!< TIM auto-reload register,               Address offset: 0x2C */
1413   __IO uint32_t RCR;         /*!< TIM repetition counter register,        Address offset: 0x30 */
1414   __IO uint32_t CCR1;        /*!< TIM capture/compare register 1,         Address offset: 0x34 */
1415   __IO uint32_t CCR2;        /*!< TIM capture/compare register 2,         Address offset: 0x38 */
1416   __IO uint32_t CCR3;        /*!< TIM capture/compare register 3,         Address offset: 0x3C */
1417   __IO uint32_t CCR4;        /*!< TIM capture/compare register 4,         Address offset: 0x40 */
1418   __IO uint32_t BDTR;        /*!< TIM break and dead-time register,       Address offset: 0x44 */
1419   __IO uint32_t DCR;         /*!< TIM DMA control register,               Address offset: 0x48 */
1420   __IO uint32_t DMAR;        /*!< TIM DMA address for full transfer,      Address offset: 0x4C */
1421   uint32_t      RESERVED1;   /*!< Reserved, 0x50                                               */
1422   __IO uint32_t CCMR3;       /*!< TIM capture/compare mode register 3,    Address offset: 0x54 */
1423   __IO uint32_t CCR5;        /*!< TIM capture/compare register5,          Address offset: 0x58 */
1424   __IO uint32_t CCR6;        /*!< TIM capture/compare register6,          Address offset: 0x5C */
1425   __IO uint32_t AF1;         /*!< TIM alternate function option register 1, Address offset: 0x60 */
1426   __IO uint32_t AF2;         /*!< TIM alternate function option register 2, Address offset: 0x64 */
1427   __IO uint32_t TISEL;       /*!< TIM Input Selection register,           Address offset: 0x68 */
1428 } TIM_TypeDef;
```

Lembre-se de que os valores a serem escritos nos registradores são os valores calculados menos 1:

```
TIM6->PSC = 64000 - 1;
TIM6->ARR = 500 - 1;
```

7. Vamos ainda iniciar com o contador em zero. Para isso, basta escrever o valor diretamente no registrador que guarda a contagem atual:

```
TIM6->CNT = 0;
```

8. Precisamos agora desmascarar a interrupção, colocando "1" no *bit* 0 do registrador TIM6_DIER, e configurar a prioridade (vamos usar o valor 1) e habilitar a interrupção no NVIC, como foi feito no Roteiro 3.

```
TIM6->DIER |= TIM_DIER_UIE;
```

Para determinar o vetor de interrupção associado ao `TIM6`, consultamos a Tabela 123 do Manual de Referência, onde encontramos que o número do vetor é 54. A interface CMSIS define o tipo `IRQn_Type` como um tipo enumerado que inclui a constante `TIM6_DAC_IRQn`, com o valor 54, conforme definido no arquivo `stm32h7a3xxq.h`. As funções CMSIS para configurar o nível de prioridade em "1" no *byte* 2 (54&~0xFFFFFFFC) do registrador de prioridade NVIC_IPRn, onde n = 54>>2 = 13 e para habilitar a interrupção da linha IRQn são, respectivamente `NVIC_SetPriority` e `NVIC_EnableIRQ`, sendo essas funções definidas no arquivo `core_cm7.h`.

```
NVIC_SetPriority(TIM6_DAC_IRQn, 1);
NVIC_EnableIRQ(TIM6_DAC_IRQn);
```

9. Até aqui, configuramos os parâmetros do contador e a sua interrupção. No código, ainda é necessário iniciar a contagem, habilitando o contador a receber os pulsos de *clock*. Isto é feito no *bit* 0 do registrador de controle (TIM6_CR1):

```
TIM6->CR1 |= TIM_CR1_CEN;
```

10. Para finalizar, é necessário definir a ISR com o código que irá inverter o estado do LED. No arquivo `Startup/startup_stm32h7a3zitxq.s`, é definida a função de nome `TIM6_DAC_IRQHandler` como ISR da interrupção do TIM6.

```
199   .word TIM6_DAC1_IRQHandler                /* TIM6 global interrupt
200   .word TIM7_IRQHandler                     /* TIM7 global interrupt
```

Acima da função *main*, escreva a seguinte função:

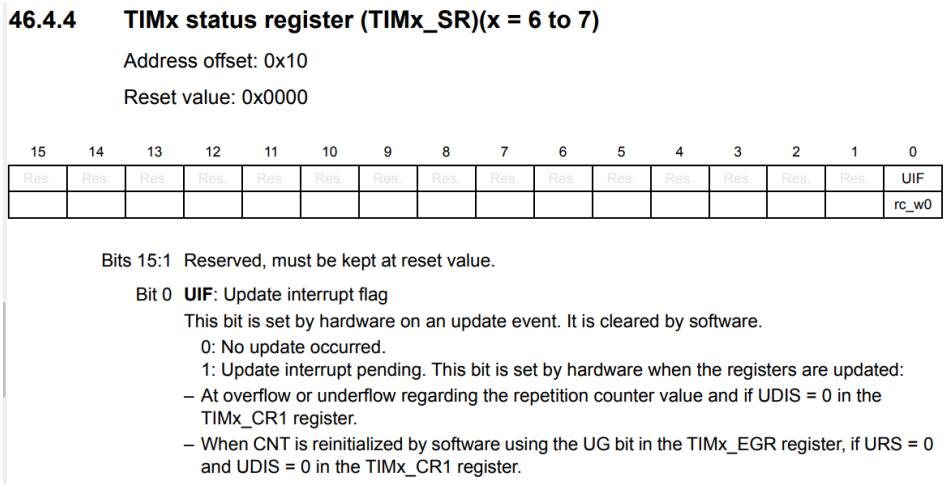
```
void TIM6_DAC_IRQHandler(void) {
      static uint8_t status = 0;

    if (TIM6->SR & TIM_SR_UIF) { // Testa a flag de atualizacao
        // Limpa a flag de atualizacao escrevendo ZERO
        TIM6->SR &= ~TIM_SR_UIF;

        if(status) { // LED ligado
            GPIOB->BSRR = GPIO_BSRR_BR0; // Desliga LED
            status = 0;
        } else { // LED desligado
            GPIOB->BSRR = GPIO_BSRR_BS0; // Liga LED
            status = 1;
        }
```

```
    }
}
```

Verifique o uso do registrador de status TIM6_SR para confirmar se o *bit* de *flag* de interrupção está setado antes de tratar o evento. Conforme descrito no Manual de Referência, esse *bit* é automaticamente setado pelo *hardware* quando o contador atinge o valor configurado no TIM6_ARR e o contador TIM6_CNT é resetado. O *bit* deve ser limpo utilizando a técnica de "*read/clear write0* (rc_w0)", ou seja, escrevendo um valor que tem o *bit* de *flag* específico definido como "0" e todos os outros *bits* em "1", para remover a solicitação de interrupção.



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

Address offset: 0x10

Reset value: 0x0000

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

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

Bit 0 **UIF**: Update interrupt flag

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

0: No update occurred.

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

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

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

Note que, neste caso, há apenas um evento de interrupção ativo (o periférico DAC1 está inativo) associado à linha IRQ54, conforme ilustrado no diagrama de blocos na Figura 488 do Manual de Referência. Como é único o evento que pode gerar uma solicitação na linha IRQ54, não é necessário identificar a fonte do evento de interrupção que gerou a solicitação. Porém, é uma boa prática de programação testar o *flag* sempre, pois em uma eventual modificação do projeto, outras fontes de interrupção que usam o mesmo vetor podem ser adicionadas.

Assim, o código completo do arquivo "main.c" fica:

```
#include <stdint.h>
#include <stm32h7a3xxq.h>

void TIM6_DAC_IRQHandler(void) {
      static uint8_t status = 0;

    if (TIM6->SR & TIM_SR_UIF) { // Testa a flag de atualizacao
        // Limpa a flag de atualizacao escrevendo ZERO
        TIM6->SR &= ~TIM_SR_UIF;

        if(status) { // LED ligado
            GPIOB->BSRR = GPIO_BSRR_BR0; // Desliga LED
            status = 0;
        } else { // LED desligado
            GPIOB->BSRR = GPIO_BSRR_BS0; // Liga LED
            status = 1;
        }
    }
}

int main(void)
{
```

```
    //Inicializa GPIOB, pino 0 (PB0)
    RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk; // GPIOB clock enable
    // 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

    //Inicializa o TIM6
    RCC->APB1LENR |= RCC_APB1LENR_TIM6EN; // clock gating
    TIM6->PSC = 64000 - 1; // prescaler
    TIM6->ARR = 500 - 1; // Auto-reload, modulo de contagem
    TIM6->CNT = 0; // Zerando o conrador

    // Configura interrupcao
    TIM6->DIER |= TIM_DIER_UIE; // Mascara de interrupcao de atualizacao
    NVIC_SetPriority(TIM6_DAC_IRQn, 1); // Prioridade 1
    NVIC_EnableIRQ(TIM6_DAC_IRQn); // Habilita interrupcao

    // Inicia a contagem de tempo
    TIM6->CR1 |= TIM_CR1_CEN;

    /* Loop forever */
    for(;;);
}
```

11. Faça o “Build” e o “Debug” e veja o funcionamento do programa.

12. Vamos analisar e desmembrar o comportamento do módulo para compreender detalhadamente suas operações e características. Em primeiro lugar, reinicie o programa com “*Terminate and Relaunch*” (ícone quadrado vermelho e triângulo verde). Coloque um *breakpoint* na linha da instrução “for (;;);”. Verifique se os valores dos seguintes registradores na aba SFRs estão compatíveis com a configuração programada.

Registrador	Endereço mapeado	Função do registrador	Valor
SCB_AIRCR			
NVIC_ICER1			
NVIC_IPR13			
TIM6_CR1			
TIM6_DIER			
TIM6_PSC			
TIM6_ARR			

Faça ações combinadas de “Pause” e “Resume” para ver a variação do conteúdo de TIM6_CNT e responder se o temporizador TIM6 é de contagem progressiva ou regressiva e se as interrupções ocorrem em *overflow* ou *underflow*.

13. Vamos verificar se o fluxo de controle é deslocado periodicamente para a ISR `TIM6_DAC_IRQHandler`. Desloque o *breakpoint* para dentro de `TIM6_DAC_IRQHandler` e continue (“`Resume`”) a execução. Ao pausar o fluxo dentro da ISR, execute a ISR, passo a passo, para monitorar o valor do *bit* de *flag* de interrupção `TIM6_SR_UIF`. Qual instrução é responsável por zerá-lo? É uma operação de mascaramento ou de atribuição direta? Remova o *breakpoint* e continue (“`Resume`”) o fluxo de execução. Reinicie (“Terminate and Relaunch”) a execução.

14. Alternadamente, continue (Resume) e pause ("Pause") a execução. Em cada pausa, ajuste o valor do *prescaler* em `TIM6_PSR`. Observe como a frequência de piscada do LED muda quando

você aumenta o valor do prescaler. E como ela se comporta quando o valor do *prescaler* é reduzido? Essa variação está de acordo com suas expectativas?

Projeto com o *SysTick*

Vamos configurar o *Systick* para gerar uma interrupção periódica a cada 1ms, e usar esta interrupção para implementar uma função *Delay*, que espera um tempo determinado em milissegundos para continuar a execução do programa. Crie o projeto "Delay_1ms", seguindo o mesmo procedimento utilizado no projeto anterior.

1. O *timer* ***SysTick*** tem 4 registradores. Um deles, o `SYST_CALIB`, é opcional, e guarda um valor de calibração fina para o temporizador; ele não será usado neste exemplo. Primeiro precisamos definir o valor máximo de contagem do *Systick* para a contagem de 1ms. A frequência inicial dos *clocks* AHB é de 64MHz. Vamos usar o *prescaler* RCC_CDCFGR1_CDCPRE dividindo esta frequência por 8, assim teremos 8MHz no contador. Para contar 1ms, serão 8000 pulsos. Assim, devemos carregar no registrador `SYST_RVR` (*Reload Value*) o valor 8000-1, pois o *Systick* também possui seu *reset* síncrono.

2. O segundo registrador que vamos configurar é o do valor corrente do contador. O registrador `SYST_CVR` permite ler o valor atual, bem como modificar o valor ao ser escrito. Vamos carregar zero neste registrador.

3. Por fim, precisamos configurar o registrador `SYST_CSR` (*SysTick Control and Status*). Este registrador usa apenas 3 *bits* para controle e um *bit* para indicar o status. Os *bits* de controle habilitam o *clock* no contador, permitindo que ele realize a contagem (*bit* 0 em "1"), habilitam a interrupção (*bit* 1 em "1") e definem se a fonte do *clock* é externa(*bit* 2 em "0") ou usa a mesma fonte do *clock* do processador (*bit* 2 em "1"). O *bit* 16 é o *flag* que, quando em "1", indica que o contador chegou a zero, sendo apagado automaticamente quando o registrador é lido. O endereço da sua ISR está localizado na 16ª posição da Tabela de Vetores de Interrupção, o que significa que o seu número do vetor é 15. **Obs:** Como não há uma linha IRQn específica associada ao SysTick, não é necessária nenhuma configuração no NVIC. A prioridade é pré-definida e a interrupção é permanentemente habilitada, podendo apenas ser mascarada.

Table 123. NVIC(1)

Signal	Priority	NVIC position	Acronym	Description	Address offset
-	-	-	-	Reserved	0x0000 0000
-	-3	-	Reset	Reset	0x0000 0004
-	-2	-	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
-	-1	-	HardFault	All classes of fault	0x0000 000C
-	0	-	MemManage	Memory management	0x0000 0010
-	1	-	BusFault	Prefetch fault, memory access fault	0x0000 0014
-	2	-	UsageFault	Undefined instruction or illegal state	0x0000 0018
-	-	-	-	Reserved	0x0000 001C-0x0000 002B
-	3	-	SVCall	System service call via SWI instruction	0x0000 002C
-	4	-	DebugMonitor	Debug monitor	0x0000 0030
-	-	-	-	Reserved	0x0000 0034
-	5	-	PendSV	Pendable request for system service	0x0000 0038
-	6	-	SysTick	System tick timer	0x0000 003C
wwdg_it	7	0	WWDG	Window Watchdog interrupt	0x0000 0040

4. Crie um projeto com suporte a CMSIS, similar ao anterior e inclua os caminhos dos arquivos-cabeçalho `stm32h7a3xxq.h` e `core_cm7.h`. Nomeie o projeto como “Delay_1ms” e abra o arquivo “main.c”. Antes da função “main()”, adicione a seguinte função para realizar a inicialização do *Systick*:

```
void SysTick_Init(void) {
    // Clock de sistema: 64 MHz
    // SysTick usa o clock dividido por 8: 64 MHz / 8 = 8 MHz
    // Precisamos de uma interrupção a cada 1ms, então:
    // Contador = 8 MHz / 1000 = 8000 ticks

    uint32_t reload_value = 8000 - 1; // Subtraímos 1 porque o contador vai de 0 a
reload_value

    // Configurando o SysTick Reload Value
    SysTick->LOAD = reload_value;
    // Resetando o valor atual do contador
    SysTick->VAL = 0;
     // Configurando uso do clock da CPU
     SysTick->CTRL |= SysTick_CTRL_CLKSOURCE_Msk;

    // Configurando o SysTick para usar o clock do processador
    SysTick->CTRL |= SysTick_CTRL_CLKSOURCE_Msk;

    // Iniciar o contador
    SysTick->CTRL = SysTick_CTRL_ENABLE_Msk;  // Habilita o SysTick
}
```

Observe que utilizamos o tipo de dado `SysTick_Type` e as macros da interface CMSIS, definido no arquivo-cabeçalho `core_cm7.h`, para acessar os registradores do módulo SysTick.

```
/**
  \brief  Structure type to access the System Timer (SysTick).
 */
typedef struct
{
  __IOM uint32_t CTRL;                   /*!< Offset: 0x000 (R/W)  SysTick Control and Status Register */
  __IOM uint32_t LOAD;                   /*!< Offset: 0x004 (R/W)  SysTick Reload Value Register */
  __IOM uint32_t VAL;                    /*!< Offset: 0x008 (R/W)  SysTick Current Value Register */
  __IM  uint32_t CALIB;                  /*!< Offset: 0x00C (R/ )  SysTick Calibration Register */
} SysTick_Type;
```

5. Precisamos ter uma variável que guarda o número de milissegundos restantes na contagem. Esta variável será decrementada pela ISR do *Systick*. Assim, deve ser uma variável global. Logo após as declarações "#include", declare a variável:

```
uint32_t count; // ms restantes
```

6. Agora vamos implementar a ISR. Na tabela de vetores de interrupção montada no arquivo `startup_stm32h7a3zitxq.s`, o nome da função é definido como "SysTick_Handler".

```
  .section .isr_vector,"a",%progbits
  .type g_pfnVectors, %object

g_pfnVectors:
  .word _estack
  .word Reset_Handler
  .word NMI_Handler
  .word HardFault_Handler
  .word MemManage_Handler
  .word BusFault_Handler
  .word UsageFault_Handler
  .word 0
  .word 0
  .word 0
  .word 0
  .word SVC_Handler
  .word DebugMon_Handler
  .word 0
  .word PendSV_Handler
  .word SysTick_Handler
  .word WWDG_IRQHandler                   /* Window Watchdog interrupt                  */
  .word PVD_PVM_IRQHandler                /* PVD through EXTI line                      */
  .word RTC_TAMP_STAMP_CSS_LSE_IRQHandler /* RTC tamper, timestamp                      */
  .word RTC_WKUP_IRQHandler               /* RTC Wakeup interrupt                       */
```

Assim, entre a declaração da variável global e a função de inicialização do *Systick*, escreva a ISR:

```
void SysTick_Handler(void) {
	count--;
}
```

Com isso, a cada milissegundo o valor de "count" será decrementado de 1.

7. Resta ainda a definição da função "Delay". A função deve carregar a variável "count" com o número de milissegundos da espera, ativar a interrupção do *SysTick* para contagem, aguardar o valor chegar a zero, e então desativar *SysTick*. Assim, depois da função de inicialização do *Systick*, escreva a função de "Delay":

```
void Delay(uint32_t ms) {
	count = ms; // carrega o valor em ms
	SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;
	SysTick->VAL = 0;
	while(count); // Loop enquanto "count" e diferente de zero
	SysTick->CTRL &= ~SysTick_CTRL_TICKINT_Msk;
}
```

8. Agora vamos implementar um pisca usando a função "Delay". Na função "main.c" vamos configurar a fonte do *clock* do *SysTick* para que seja a mesma do processador, configurar o divisor

de frequência em 8, configurar o pino de saída do LED verde como nos exemplos anteriores, inicializar o *Systick* com a interrupção desativda e alternar o LED entre os estados ligado e desligado usando o "Delay" para espaçar os instantes de alternância. A função "main()" fica assim:

```
int main(void)
{
	// Configurar o divisor em 8
	RCC->CDCFGR1 &= ~RCC_CDCFGR1_CDCPRE_DIV512_Msk;
	RCC->CDCFGR1 |= RCC_CDCFGR1_CDCPRE_DIV8;

	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk;
	GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk);
	GPIOB->MODER |= GPIO_MODER_MODE0_0;
	GPIOB->OTYPER &= ~(GPIO_OTYPER_OT0_Msk);

	SysTick_Init();

	for(;;) {
		GPIOB->BSRR = GPIO_BSRR_BS0; // Liga LED
		Delay(500);
		GPIOB->BSRR = GPIO_BSRR_BR0; // Desliga LED
		Delay(500);
	}
}
```

9. Faça o "Build" e o "Debug".

10. Coloque um *breakpoint* na linha "for (;;)" e continue ("Resume") a execução. Na pausa, verifique se os valores dos seguintes registradores na aba SFRs estão compatíveis com a configuração programada.

Registrador	Endereço mapeado	Função do registrador	Valor
RCC_CR			
RCC_CDCFGR1			
SysTick_STCSR			
SysTick_STRVR			

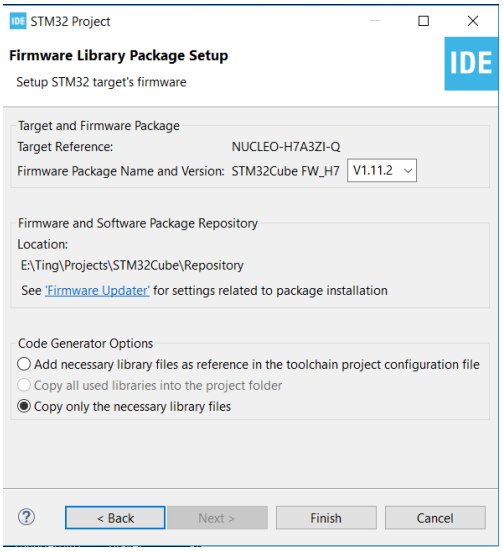
11. Usando combinadamente "Pause" e "Resume", verifique se o SysTick é um temporizador de contagem progressiva ou regressiva e se a sua interrupção ocorre em *overflow* ou *underflow*. Explique o funcionamento do sistema implementado.

12. Você pode alternar o estado do LED verde diretamente na ISR, semelhante ao que foi feito no projeto anterior. Valide essa abordagem implementando-a e verificando seu funcionamento.

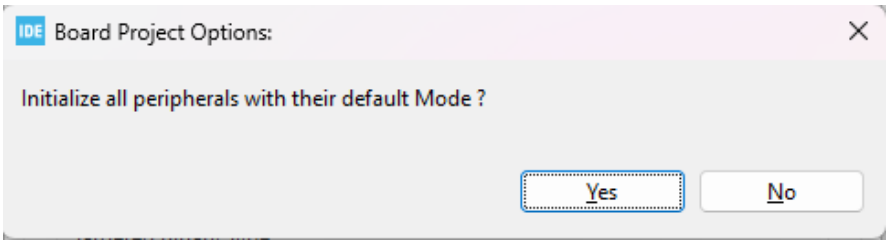
Projeto com um *timer* usando o STM32CubeMX e a HAL

Vamos trabalhar os *timers* e a GPIO usando o STM32CubeIDE e a HAL no lugar do CMSIS:

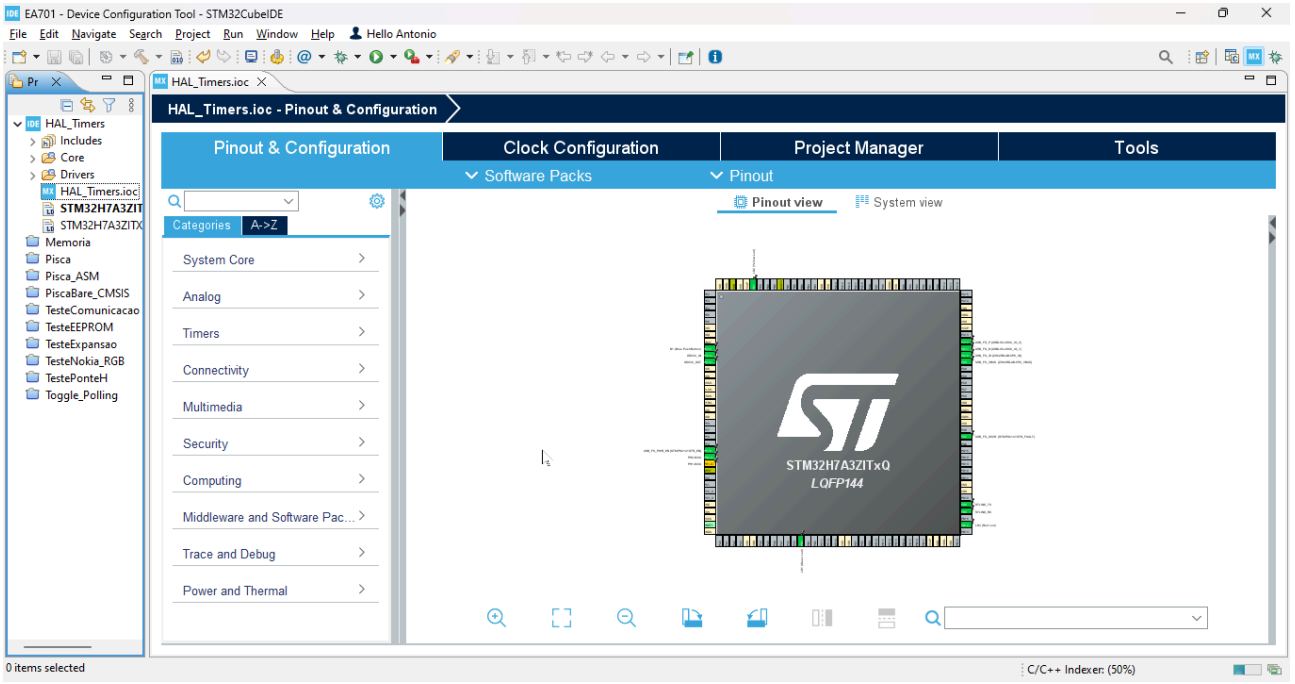
1. Crie um novo projeto da mesma forma que foi feito com os anteriores, exceto que na opção "Targeted Project Type" vamos manter o padrão "STM32Cube" para que o Cube seja incluído. Dê o nome de "HAL_Timers" a este projeto. Na sequência, se for clicado o botão "Next" abre-se uma janela para a configuração do pacote de biblioteca de *firmware*. Deixe as **opções padrão** e clique em *Finish.* Na janela anterior, pode-se clicar em "Finish" em vez de "Next" e omitir esta etapa.



Deixe as opções padrão e clique em *Finish.* Como estamos trabalhando com uma placa que dispõe de alguns periféricos ligados ao microcontrolador, aparecerá uma janela perguntando se deseja iniciar os periféricos na configuração "default". Clique em "Yes" para que o código gerado pelo STM32CubeIDE já inclua a inicialização dos periféricos da placa. Com isso, teremos já inicializados os três LEDs da placa como saídas digitais, e o botão azul como entrada digital.

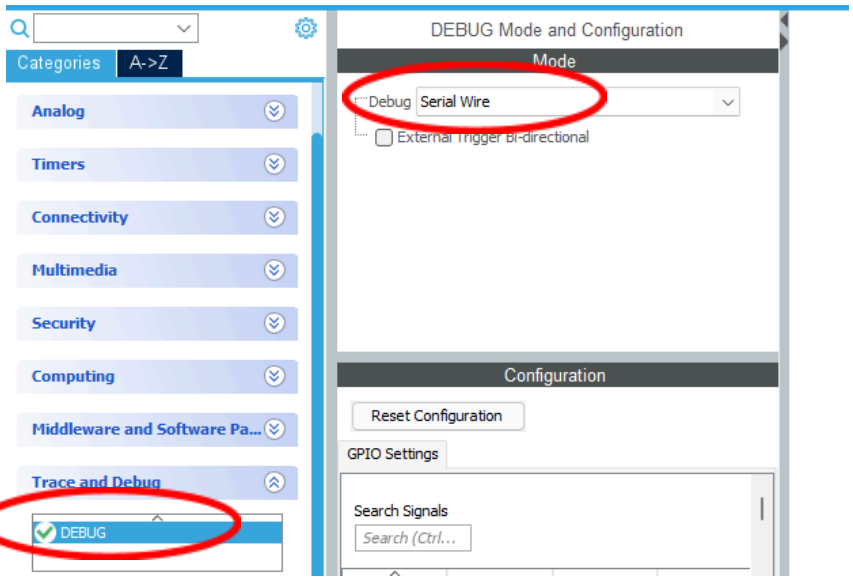


Após a instalação dos pacotes de suporte, o IDE entra na perspectiva de Inicialização, com o arquivo `HAL_Timers.ioc` aberto no modo gráfico numa aba (este é o plugin ***CUBE*** do IDE). A aba mostra graficamente o arquivo de configuração `HAL_Timers.ioc`, que guarda todas as configurações de periféricos para que o módulo STM32CubeMX gere o código de inicialização dos mesmos. Dominando a tela, aparece uma representação física do microcontrolador, com controles de zoom. À esquerda, há um painel com uma lista dos diversos periféricos do microcontrolador, bem como eventuais algoritmos de cálculo e recursos de *Middleware* (sistemas de arquivos, dispositivos USB, etc).



2. Dê um *zoom* na imagem do circuito integrado (clicar no ícone Maximize no canto direito superior) para que ele preencha a tela. Pode-se também clicar com o *mouse* sobre ele e arrastar para ajustar a posição. Vamos usar como periféricos os LEDs da placa e o *Push-Button* de usuário (azul).

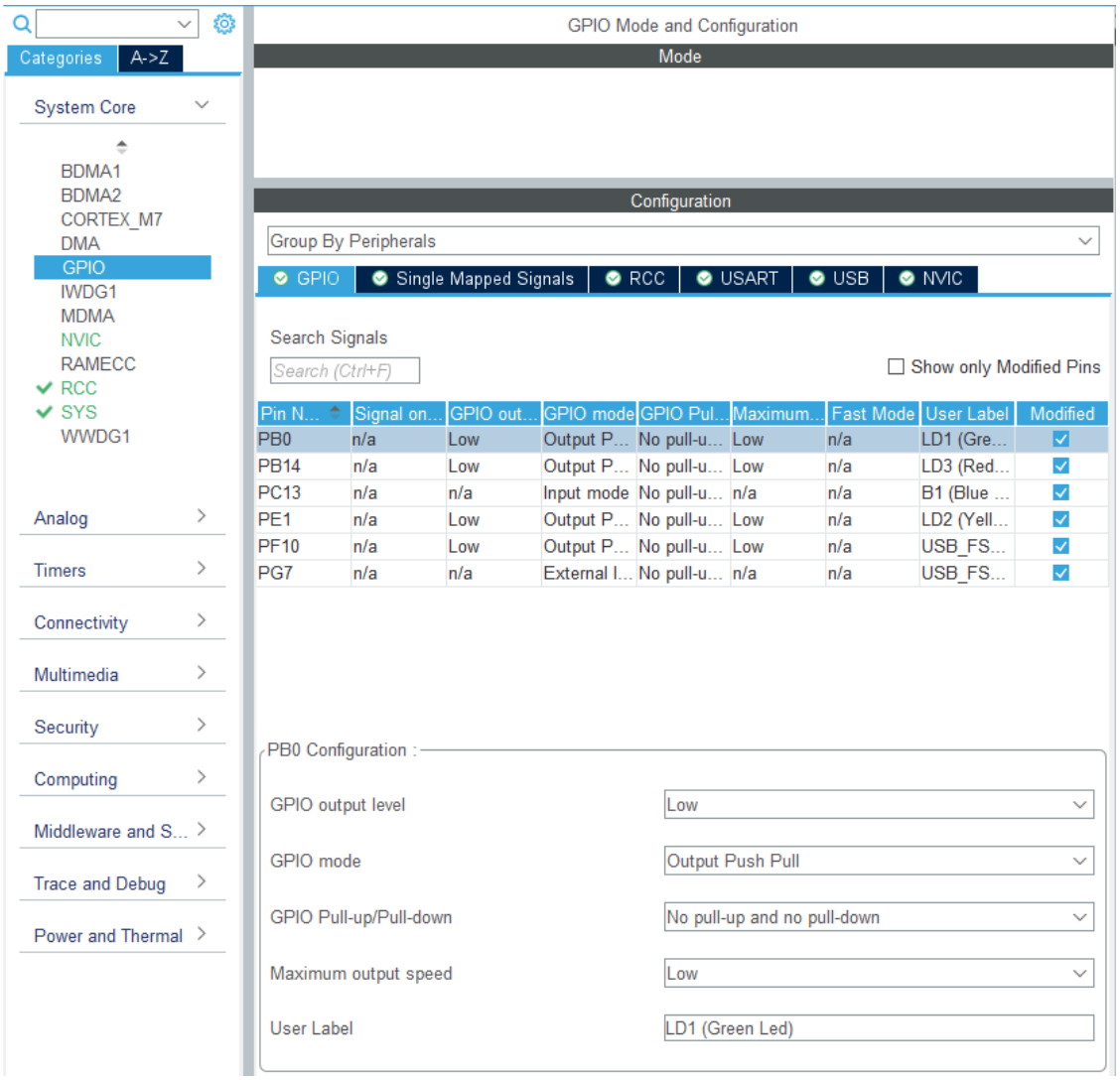
Vamos revisar os itens e configurar alguns deles. Clique sobre o item "*Trace and Debug*" e verá o item "*DEBUG*" abrir. Clique sobre o item e um novo painel irá abrir entre a lista e a representação física do controlador. Precisamos ativar o modo de depuração do microcontrolador, e para isso selecione "*Serial Wire*" selecione na lista do item "*Debug"*, Curiosamente, o *Debug* não é ativado por padrão ao se criar um projeto no STM32CubeIDE. **Deve-se lembrar de sempre ativar o *debug* nos próximos projetos.**



Assim, o modo SWD de *debug* é ativado (o modo usado pelo depurador ST-LINK). Pode-se ver que dois dos pinos da imagem "*Pinout view*" do microcontrolador (PA13 e PA14) agora aparecem em verde e com identificações nas funções selecionadas. Além disso, os dois pinos são inseridos na lista

de "*Configuration*". Estes pinos são ligados ao ST-LINK presente na placa NUCLEO. Cada periférico ativado através do STM32CubeMX tem seus pinos correspondentes em verde no editor gráfico.

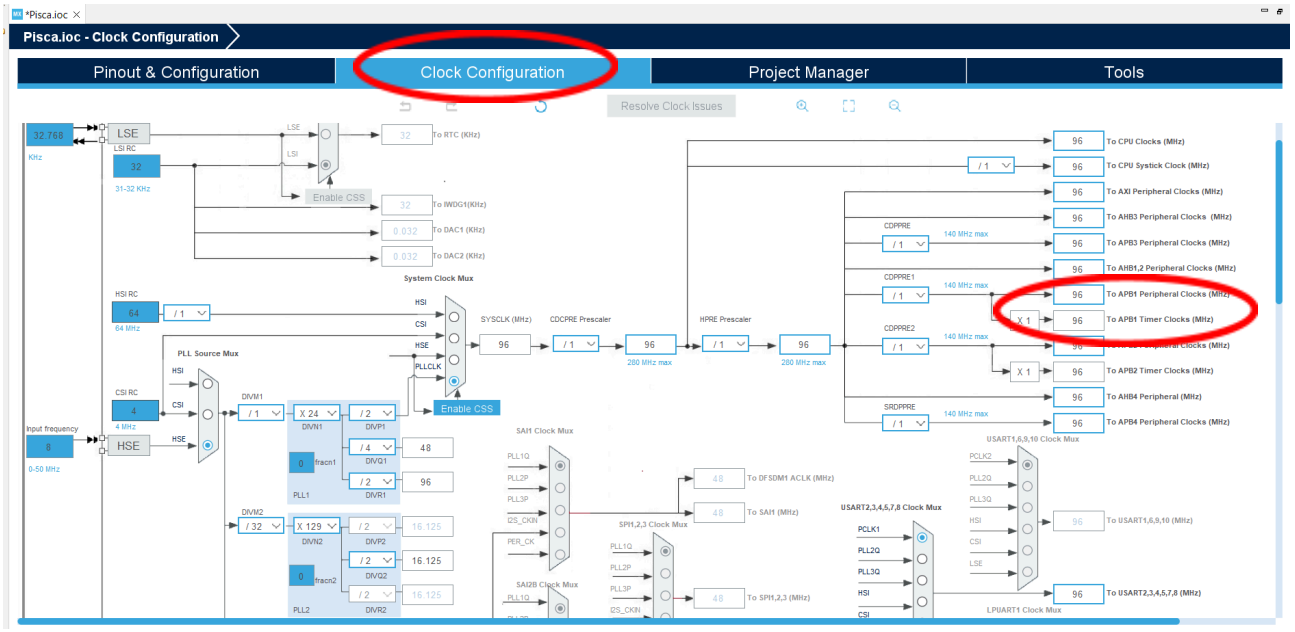
3. Agora vamos revisar os pinos de saída digital para controlar os 3 LEDs existentes na placa. Estes LEDs são ligados nos pinos PB0 (verde), PE1 (amarelo) e PB14 (vermelho), acendendo quando o pino correspondente está em nível lógico alto. Procure o pino PB0 na imagem (no canto inferior direito deste painel há um campo de busca. Basta digitar o nome do pino e ele pisca na imagem). Pode-se ver que este pino já foi selecionado e está com um identificador (LD1). No painel esquerdo, selecione o grupo "System Core" e a opção "GPIO". No painel do meio, aparecerão todos os pinos já configurados como GPIO na placa, o que inclui os três LEDs de usuário e o *Push-Button*. Clicando sobre cada pino de GPIO configurado neste painel, pode-se ver se ele está configurado como entrada ou saída, o nível lógico inicial (*Low*), bem como outros parâmetros de configuração, além de um campo para definir o "User Label" (ou identificador atribuído pelo usuário). Os identificadores LD1, LD2 e LD3 foram atribuídos aos três LEDs e o identificador B1 foi atribuído ao *Push-Button*. Estes identificadores são mnemônicos que podem ser usados na definição dos parâmetros das funções HAL. **Quando as funções forem usadas, veremos o uso dos identificadores**.



4. O sistema de *clock* já é configurado para que o microcontrolador receba *clock* externo (no grupo "System Core", na opção "RCC", pode-se ver que o "*High Speed Clock*" está ajustado para "*BYPASS Clock Source*", ou seja, *clock* externo). A MCU que funciona como ST-LINK na placa NUCLEO possui um cristal de 8MHz para gerar seu *clock* básico de alta precisão, e seu *firmware* configura seu pino PA8 para se conectar ao gerador, exteriorizando o *clock*. Este pino é ligado ao pino PH0 (OSC_IN) do STM32H7A3, que, na configuração *default* do *Cube*, passa a ter seu *clock* recebido de fonte externa através deste pino.

Mais acima no painel, há uma aba chamada "Clock Configuration". Clicando nela, a imagem muda para uma visão hierárquica dos sinais de relógio suportados pelo microcontrolador, destacados em azul, para atender os mais diversos módulos integrados mostrados no lado direito, podendo-se definir uma frequência distinta para cada barramento interno. À esquerda, pode-se ver as duas frequências de entrada disponíveis no nosso microcontrolador, o bloco do cristal de 8MHz ao lado de HSE (*High Speed External*) e a fonte de 32.768 kHz. Além disso, os multiplexadores MUX foram selecionados para que o sinal de 8MHz passe pelo PLL e tenha a frequência multiplicada para 96MHz, e esta frequência seja usada pelo sistema.

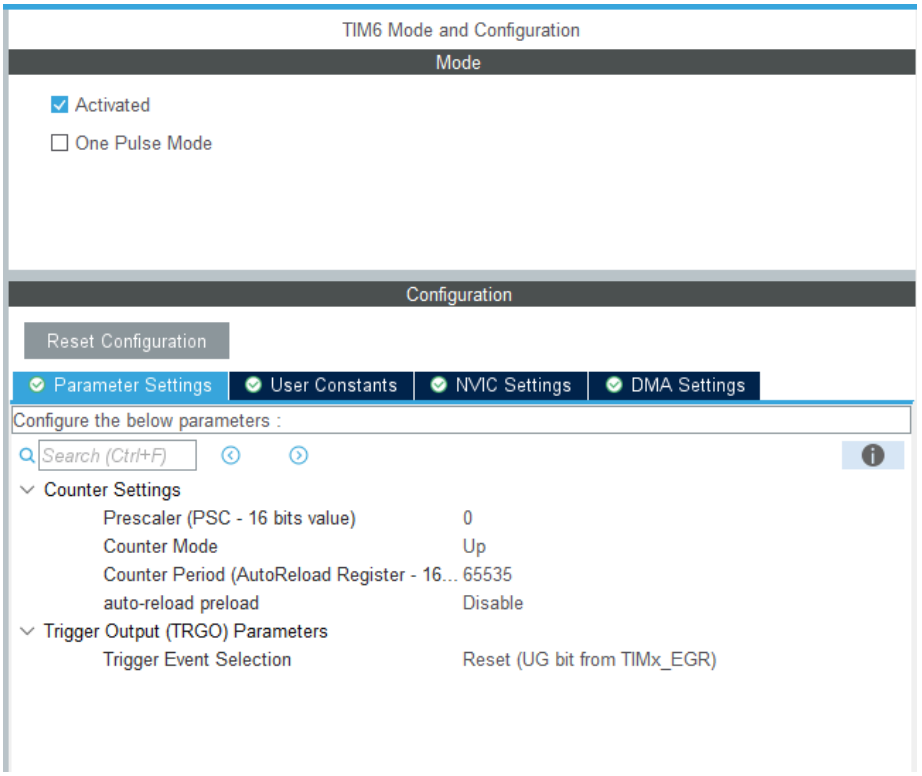
Para definir a duração do período de interrupção, é necessário saber o *clock* que o *timer* utilizado recebe. No item 4, vimos que todos os barramentos recebem um *clock* de 96MHz. É possível ter *clocks* diferentes nos vários barramentos, e neste caso vemos a frequência presente no quadro "To APB1 Timer Clocks" (96MHz) que nos interessa. **Note bem esta frequência!**



5. Precisamos agora configurar um *timer* para gerar uma interrupção periódica, que definirá o intervalo de tempo entre a troca do estado do LED. Voltando à aba "Pinout & Configuration", expanda a categoria "Timers". Vamos usar novamente o TIM6 para gerar a interrupção periódica. Clique na palavra "TIM6" para que as opções deste *timer* apareçam.

6. Vamos ativar o TIM6. No painel central, parte superior (*Mode*), temos apenas uma caixa de seleção denominada "Activated". Os *timers* mais complexos permitem selecionar uma de várias fontes de *clock*, mas neste *timer* simples, só podemos usar a frequência do *clock* do APB1. Marque a caixa "Activated" e o sub-painel "Configuration" na parte inferior do painel central ficará ativa.

Arraste a barra entre os dois painéis para dar mais espaço ao painel inferior. Nesta aba de editor gráfico de arquivos de configuração de periféricos (em inglês *Input/Output Configuration*, sigla ioc) utilizados pelo STM32CubeMX, todas as divisões entre painéis são ajustáveis. O STM32CubeMX (a partir daqui designado ***Cube***) é uma ferramenta dedicada à configuração dos microcontroladores STM32 integrada no ambiente de desenvolvimento STM32CubeIDE.

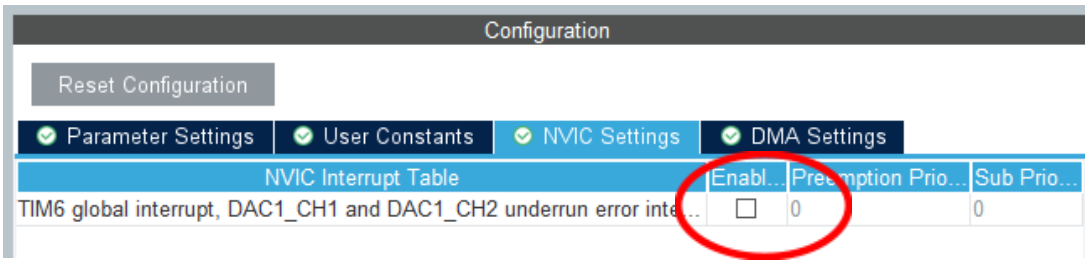


7. Vamos primeiro configurar os parâmetros gerais. Temos um *timer* que trabalha a 96MHz (T = 10,4166ns) e precisamos contar um tempo de cerca de 500ms (f = 0,5Hz). Dividindo 96 x 10^6 por 0,5, teremos um valor de 192 x 10^6. Como o *timer* é de 16 *bits,* ele só pode contar até 65535. Precisamos reduzir a frequência de *clock* na entrada do *timer*, e para isso usamos o *prescaler*, que divide a frequência de *clock* por um valor definido, também de 16 *bits*. Se configuramos o *prescaler* para o valor 48000, a frequência de contagem no *timer* será de (96MHz / 48000) 2kHz. Para contar 500ms, serão 1000 pulsos de contagem (2 pulsos para dada milissegundo). Agora podemos configurar os registradores de *prescaler* e de módulo de contagem.

IMPORTANTE: Tanto o contador do *prescaler* como o do *timer* em si possuem o *reset* síncrono, ou seja, ao atingir o valor de contagem definido eles serão zerados apenas no próximo pulso de *clock*. Assim, para contar *n* pulsos, o contador deve contar de 0 até *n* - 1. Por isso, os valores definidos devem ser os calculados no parágrafo anterior subtraídos de 1. Clique no número “0” ao lado do campo “Prescaler” e digite “47999” ou ainda “48000-1”. Faça o mesmo no campo “Counter Period” e digite “999” ou ainda “1000-1”. Assim, o contador do *timer* irá contar de 0 a 999 com uma frequência de 2kHz e será reiniciado. Este reinício gera um sinal interno que coloca um *bit* do registrador de estado do *timer* em “1” e, se habilitado, gera uma interrupção.

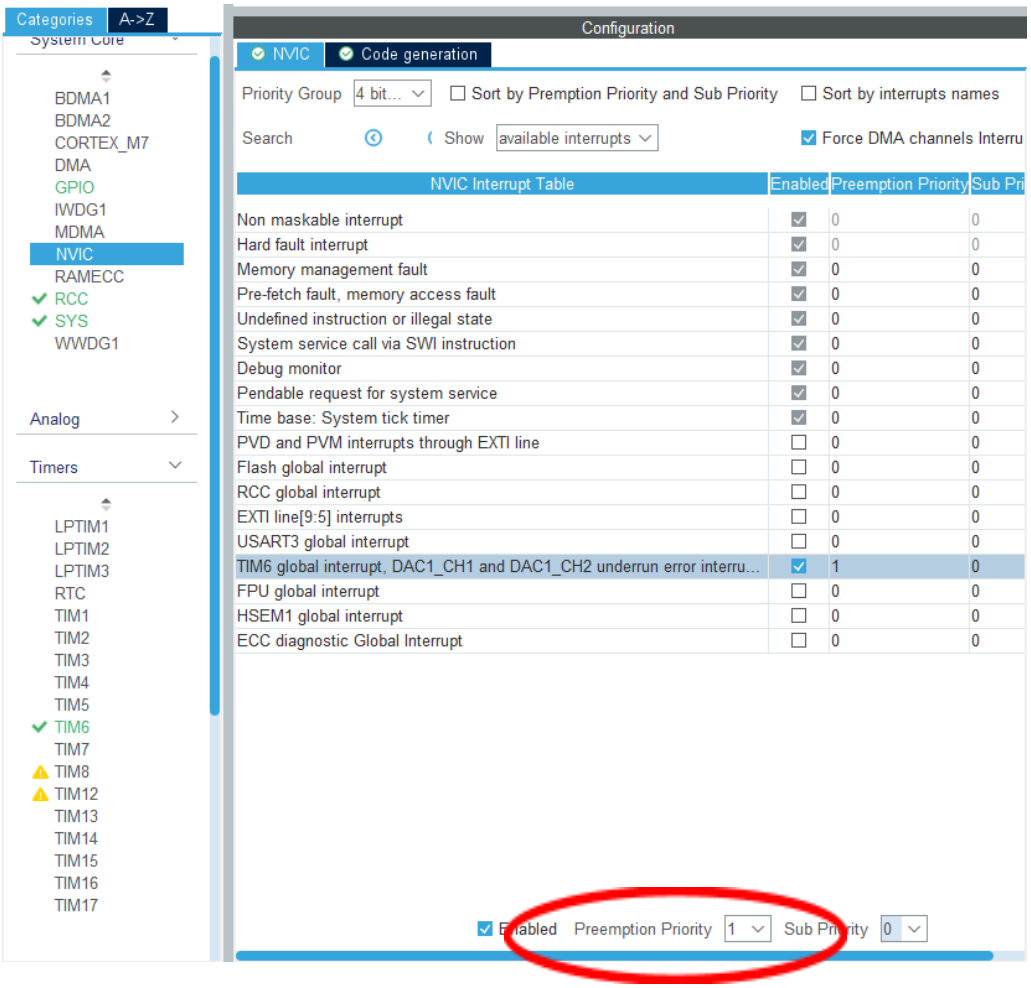
8. Vamos configurar a interrupção. No mesmo painel central inferior, há um conjunto de abas, sendo que a aba selecionada é a “Parameter Settings”. Clique na aba “NVIC Settings” para configurar a

interrupção do *timer*. Veja que o vetor de interrupção é compartilhado pela interrupção global (elicitada por reinício de contagem ou ainda por erros) e por interrupções dos DAC, que estão desativados neste projeto. Assim, a interrupção será usada exclusivamente por TIM6. Clique na caixa de seleção da coluna "Enable" para ativar a interrupção.



9. Agora, no painel à esquerda, dentro do grupo "System Core" selecione o módulo "NVIC". Veja que aparecem os vetores de interrupção ativados ou estão pré-reservados para ativação, indicando periféricos que foram ativados mas cuja interrupção ainda não foram ativada; Note que na linha da interrupção de TIM6, a caixa da coluna "Enabled" está selecionada. Vamos apenas ajustar a prioridade da interrupção. Na parte inferior do painel existe uma caixa de seleção nomeada "Preemption Priority". Selecione o valor "1" para a mesma.

Geralmente reservamos as prioridades mais altas (nível "0") para as exceções do *Core*. Assim, vamos ajustar nossa interrupção para um nível abaixo das exceções. Selecionar adequadamente o nível de prioridade de cada interrupção em um sistema com múltiplas interrupções é fundamental para o bom funcionamento do sistema.

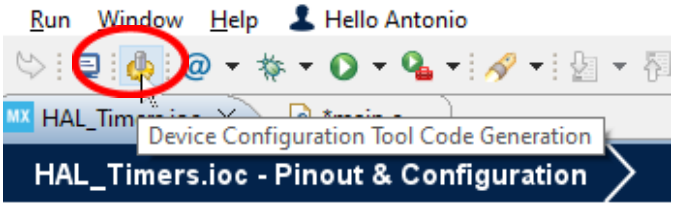


10. Abra a aba “Code Generation” para certificar se é habilitada a geração de códigos para a ISR correspondente a “TIM6 global interrupt, DAC1_CH1 and DAC1_CH2, underrun error interrupts”.

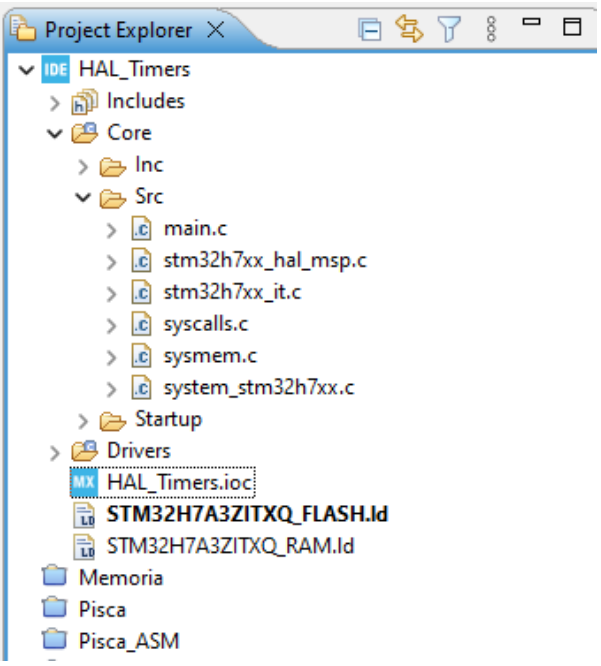
11. Agora temos a configuração completa para o projeto. Ao salvar o arquivo “HAL_Timers.ioc” por meio de “File” > “Save” (ou no ícone padrão de “Salvar” ou ainda usando Ctrl+S no teclado), é perguntado se deseja gerar o código. Basta clicar em “Yes”, e o código será gerado. Se quiser, ative “*Remember my decision*” para que esta solicitação ocorra apenas no primeiro uso.

Na primeira vez que isto é feito, aparece na sequência uma caixa de diálogo perguntando se deseja mudar a perspectiva. Selecione a caixa para que essa opção seja lembrada e clique no botão “Yes”.

Se o projeto foi salvo mas o código não foi gerado (nenhum protótipo de função privada foi inserido), pode-se gerar o código clicando no ícone com uma engrenagem amarela (*Device Configuration Tool Code Generation*).



12. À esquerda podemos ver arquivos inseridos na estrutura de arquivos do projeto e os demais projetos do *workspace* na aba “Project Explorer”. O ideal é manter apenas um projeto aberto de cada vez. Para fechar um projeto, basta clicar sobre ele com o botão direito e escolher a opção “Close Project”. Para abrir um projeto fechado, basta dar um duplo-clique sobre ele. Nos projetos usando o STM32CubeIDE, a pasta “Src” que contém o arquivo “main.c” está dentro de outra pasta nomeada “Core”. Ao passar para a perspectiva de programação, o arquivo “main.c” é aberto no editor.



13. Dentro do arquivo “main.c”, podemos ver a árvore de estrutura à esquerda (*outline*, que destaca os principais elementos do arquivo), e linhas de comentários marcando BEGIN e END de várias seções. Qualquer código escrito entre as linhas “BEGIN <rótulo>” e “END <rótulo>” da seção <rótulo> será mantido, mesmo que haja modificações na perspectiva de Inicialização (e consequente re-geração de código). Portanto, lembre-se de escrever seu código sempre em regiões do arquivo entre um BEGIN e um END da mesma seção, senão, caso faça modificações na perspectiva do STM32CubeIDE e mande gerar novamente o código de configuração, seu código será apagado.

As principais áreas para código de usuário em um projeto gerado pelo STM32CubeIDE são (na ordem em que aparecem no main.c):

Header: Logo no início, para comentários gerais

Includes: Para declarações “#include”

PTD: Para a criação de *typedefs* (tipos de dado criados pelo usuário a partir de tipos padrão)

PD: Para declarações de “#define”

PM: Para declarações de macros

PV: Para declarações de variáveis globais

PFP: Para declarações de protótipos de funções

0: Também pode ser usada para variáveis globais

1: Para declarações de variáveis locais

Init: Para inicializações realizadas após a inicialização das funções HAL mas antes das inicializações de *clock* e de periféricos

SysInit: Para inicializações realizadas após a inicialização do *clock* mas antes das inicializações de periféricos

2: Código principal que não é executado em *loop*

3: Código principal executado em *loop* infinito

4: Funções auxiliares criadas pelo desenvolvedor

```
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */
/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
/* USER CODE BEGIN PFP */
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
/* USER CODE END 0 */

```

Além disso, o STM32CubeIDE inclui cinco protótipos de funções específicas do projeto, geradas automaticamente com base nas configurações realizadas no arquivo "HAL_Timers.ioc". A definição dessas cinco funções está localizada após a definição da função "main" no arquivo "main.c" (seção "Private function prototypes").

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART3_UART_Init(void);
static void MX_USB_OTG_HS_USB_Init(void);
static void MX_TIM6_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */
```

14. Neste exemplo, vamos fazer com que a cada momento um dos três LEDs da placa esteja aceso, em sequência, alternando a cada 500ms (interrupção periódica do *timer*). Para isto, vamos usar as funções da biblioteca HAL. Há uma função que permite escrever o valor lógico de um pino GPIO de saída. Veja que dentro da função main(), o gerador de código já incluiu as funções de inicialização dos periféricos. Ou seja, tudo aquilo que antes era feito mudando *bits* em registradores agora será feito através de funções HAL. Lembre-se de que cada uma destas funções implementa um código que manipula os registradores da mesma forma como fizemos nos roteiros anteriores. Ou seja, o código que criamos nos roteiros anteriores aqui foi gerado automaticamente e encapsulado em funções.

15. Para facilitar o entendimento do código principal, vamos criar uma função que recebe um número de 1 a 3 e acende o LED correspondente, apagando os demais. Procure a seção `/* USER CODE BEGIN PFP */` e entre esta linha e seu END, digite o **protótipo** da função:

```
void LedOn(uint8_t);
```

O **protótipo** da função especifica a sintaxe da mesma, ou seja, seu nome, os dados que recebe e o que ela retorna. Os protótipos devem ser sempre colocados antes da função main(), ou ainda em arquivos ".h" incluídos no projeto, para que o compilador saiba se a sintaxe escrita pelo programador está correta.

16. Logo abaixo da linha `/* USER CODE BEGIN 4 */`, escreva a função:

```
void LedOn(uint8_t led) {
	if(led == 1) {
		HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET);
		HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
		HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
	}
	if(led == 2) {
		HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_RESET);
		HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
		HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
	}
	if(led == 3) {
		HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_RESET);
		HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
		HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);
	}
}
```

Cada condição acende um LED e apaga os outros dois. Note que a função *HAL_GPIO_WritePin* recebe como parâmetros a porta onde o pino está, o número do pino e o nível lógico. Neste caso, o gerador de código criou no arquivo de cabeçalho "main.h" macros para estes parâmetros: *LD1_GPIO_Port* e *LD1_Pin* (e as mesmas para LD2 e LD3). As macros para o estado lógico, GPIO_PIN_SET e GPIO_PIN_RESET, são definidas no arquivo "stm32h7xx_hal_gpio.h" que está na árvore de alto nível de abstração (HAL) dos *drivers* do STM32Hxx.

A seção de usuário 4 é normalmente usada para escrever as funções de usuário definidas dentro do "main.c".

17. Agora vamos escrever o programa principal, que simplesmente ficará em um *loop* infinito esperando a interrupção de *timer* e depois mudando o LED aceso. Para isso, precisamos de uma variável que indique qual LED está aceso, e precisamos dar um valor inicial à mesma, bem como acender um LED inicial, antes de entrar no *loop*. Na seção `/* USER CODE BEGIN 1 */` no começo da função main(), declare a variável:

```
uint8_t led = 1;
```

Agora temos a variável de estado de LED já com o valor inicial. Podemos acender o LED apenas após a inicialização do ecossistema HAL e configuração do sistema de sinais de relógio e dos periféricos. Na seção `/* USER CODE BEGIN 2 */` digite a linha:

```
LedOn(led);
```

para acender o LED 1.

18. O STM32CubeIDE configura o *timer*, porém não habilita a contagem. Isto deve ser feito no código do usuário. Assim, ainda na seção de usuário 2, logo após a linha que chama a função "LedOn", digite:

```
HAL_TIM_Base_Start_IT(&htim6);
```

Veja que as funções da HAL sempre iniciam com "HAL_", seguida de um mnemônico para o tipo de módulo (no caso, "TIM", de "TIMER"), e a função em si. Aqui vamos iniciar a contagem no contador "Base" ("Base_Start") habilitando o uso das interrupções ("_IT"). Note o parâmetro "&htim6". Aqui estamos passando o endereço de um *device pointer*, que é uma struct `TIM_HandleTypeDef` definida em "STM32H7xx_HAL_Driver/stm32h7xx_hal_tim.h". Esta struct contém informações sobre a configuração de um periférico, no caso o *device* TIM6. O STM32CubeIDE cria os *device pointers* e as funções de inicialização dos periféricos usando os mesmos. Os *device pointers* permitem que o programador use a mesma função HAL para vários periféricos do mesmo tipo, reaproveitando a função e passando à mesma o *device pointer* do periférico correspondente naquela chamada.

19. Precisamos ainda implementar a ISR do *timer*. Segundo a tabela de vetores de interrupção implementada "Startup/startup_stm32h7a3zitxq.s", a ISR para tratar eventos de interrupção no TIM6 (IRQ=54) é `TIM6_DAC_IRQHandler`. Na HAL, o tratamento de interrupções é simplificado. A ISR delega o gerenciamento de interrupções à HAL, que identifica a função *callback* previamente registrada. O desenvolvedor apenas precisa substituir o *callback* padrão pela sua função personalizada, sem se preocupar com os detalhes das fontes de interrupção (como IRQn) ou parâmetros envolvidos. No Manual do Usuário da HAL, capítulo 88 (*HAL TIM Generic Driver*), localize o *callback* que melhor se adapta ao tratamento de interrupções periódicas para periféricos TIMx. O *callback* apropriado é "void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)". Copie e cole o protótipo no arquivo "main.c", logo abaixo da função "LedOn" que criamos, dentro da área "User Code" 4. Adicione um "{" ao lado da declaração e dê um "Enter". O editor irá adicionar o "}" que encerra a função. Em seguida, escreva o código dentro da função criada, para que a função resultante fique conforme o que segue abaixo.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {

	led++;
	if(led > 3) {
		led = 1;
	}
	LedOn(led);
}
```

Note que não é necessário declarar o protótipo desta função, pois seu protótipo já está inserido em "STM32H7xx_HAL_Driver/Inc/stm32h7xx_hal_tim.h". Este arquivo-cabeçalho é incluído com a diretiva #include no arquivo "main.h".

20. A área de código 3 é destinada a funções que são executadas repetidamente em um loop infinito. Note que, em vez de usar "for(; ;){ }", é utilizado "while (1) { }". Ambos os comandos são equivalentes em termos de funcionalidade. Neste exemplo, após a configuração geral (até a seção de

usuário 2), o programa simplesmente entra em um *loop* infinito e deixa que a função de *callback* realize a mudança dos LEDs a cada interrupção de *timer*. Assim, não mexa em nada nesta seção

21. Dê um *Build* e note o erro: a variável "led" não está declarada na função. Isso acontece porque esta variável foi declarada dentro do escopo da função main() e ela é usada na função "**void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)**". Para resolver o problema, remova a linha `uint8_t led = 1;` de dentro da função main() e a coloque logo abaixo da linha `/* USER CODE BEGIN PV */`, que é a área para variáveis globais.

22. Dê novamente o *Build*, transfira o código executável para o microcontrolador no modo *Debug*.

23. Coloque um *breakpoint* na linha da instrução "for (;;);". Continue ("Resume") a execução. Na pausa, verifique se os valores dos seguintes registradores na aba SFRs estão compatíveis com a configuração programada e compare-os com os valores setados no primeiro projeto, em qua apenas a interface CMSIS foi utilizada.

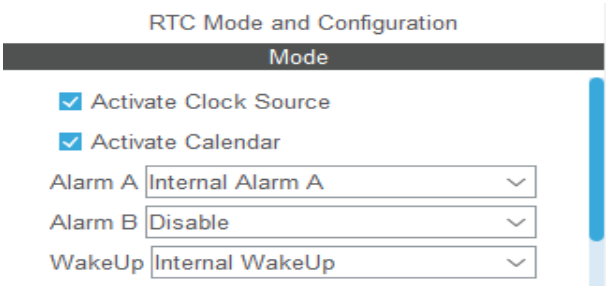
Registrador	Endereço mapeado	Função do registrador	Valor
SCB_AIRCR			
NVIC_ICER1			
NVIC_IPR13			
TIM6_CR1			
TIM6_DIER			
TIM6_PSC			
TIM6_ARR			

23. Como você pode mudar o intervalo entre troca de LEDs de 500ms para 300ms, através do registrador de *autoreload*? Pause o fluxo de execução e modifique o registrador através da aba dos SFRs. Volte a executar o programa e veja o resultado.

Projeto com RTC usando STM32CubeMx e HAL

1. Vamos programar o RTC usando o STM32CubeMX e as funções HAL para que ele conte o tempo, gere um evento de alarme no dia e horário definidos, e dispare uma interrupção a cada segundo. Crie um projeto do tipo STM32Cube, denominado "RTC". Ative o DEBUG via SWD ("Serial Wire"), conforme o item 2 do projeto anterior.

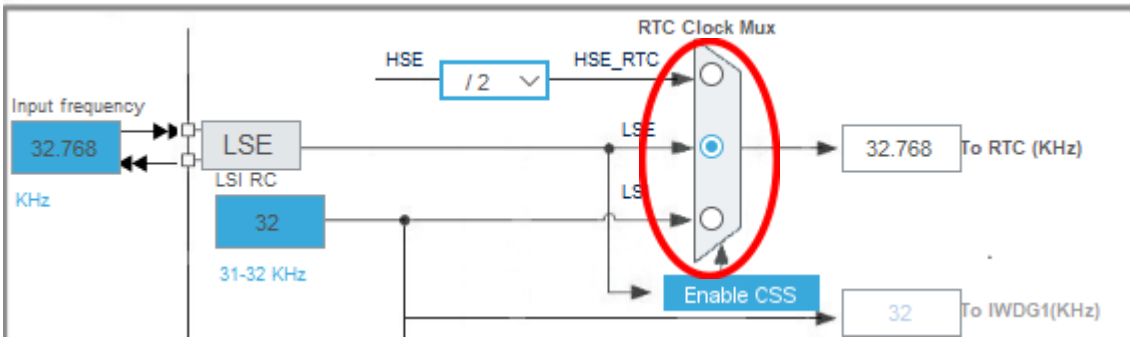
2. Expanda a seção "Timers" novamente e selecione RTC. No painel central, teremos as opções do RTC. Na parte superior ("Mode"), selecione as caixas "Activate Clock Source" (ativa o módulo) e "Activate Calendar" (ativa as funções de calendário e alarme). No item "Alarm A", selecione a opção "Internal Alarm A" (gera interrupção interna no alarme A), e no item "WakeUp", selecione a opção "Internal WakeUp" (gera interrupção periódica a partir do RTC).



3. Ainda no painel central, abaixo de “Mode”, temos o painel “Configuration”. Selecione “Parameter Settings”. Vamos manter inalteradas as opções de “General” (notando que vamos apresentar as horas no formato de 24 horas). No grupo “Calendar Time”, escolha um horário para colocar em “Hours”, “Minutes” e “Seconds”, bem como uma data para colocar em “Week Day”, “Month”, “Date” (dia do mês) e “Year” (valores 0 a 99 para anos 2000 a 2099). No grupo “Alarm A”, escolha um horário para “Hours”, “Minutes” e “Seconds” próximo ao horário ajustado no Calendar (por exemplo, se no calendar temos 10:11:00, podemos colocar 10:12:00 para o alarme ser acionado um minuto após o início da execução do programa).

4. Ainda no grupo “Alarm A”, existem as máscaras que permitem que alguns valores sejam ignorados na comparação do alarme. Os itens a serem ignorados devem ter suas máscaras em “Enable”. Nós vamos usar horas, minutos e segundos para o alarme (se colocarmos a máscara nos segundos, quando as horas e minutos do alarme coincidirem, o RTC irá gerar uma interrupção por segundo até que o minuto mude; por isso vamos manter os segundos para que apenas uma interrupção seja gerada). Defina a máscara “Alarm Mask Date Week Day” como “Enable” para garantir que o alarme seja acionado no dia ou no dia da semana configurado, e mantenha as outras opções como “Disable”. Além disso, selecione “Day” na configuração “Alarm Date Week Day Set”.

4. Clique na aba “Clock Configuration” para ver a “árvore” de clock. Na parte superior, há um bloco denominado LSE, que usa o cristal de 32.768kHz para gerar o clock de precisão de 1Hz. Mais à direita deste bloco, há um seletor de fonte, e a seleção deve ser modificada da entrada inferior para a do meio:



6. Volte à aba de “Parameter Settings”. No grupo “WakeUp”, selecione o “Wake Up Clock” para “RTCCLK/16”. Isto permite que o contador de “WakeUp” use o mesmo *clock* de 32.768Hz usado para a contagem dos segundos, dividido por 16, o que resulta em 2048. No campo “Wake Up Counter”, coloque o valor “2047” para gerar a interrupção a cada 1 segundo (Este é também um contador de *reset* síncrono).

7. No painel “Configuration”, mude a aba para “NVIC Settings” e ative as caixas de seleção para ativar a interrupção de alarmes e a interrupção de “wakeup”. No painel da esquerda, no grupo

“System Core” selecione “NVIC” e, da mesma forma que no exercício anterior, mude a prioridade da linha “RTC Alarms (...)” para 2 e a da linha “RTC wake-up (…)” para 1.

8. Salve o projeto e gere o código de inicialização. A perspectiva do IDE muda para a de programação. Vamos criar variáveis para ler o RTC (hora e data), e uma variável de contagem simples para facilitar a colocação do “breakpoint”. Abaixo da linha `/* USER CODE BEGIN 1 */`, digite:

```
RTC_TimeTypeDef myTime = {0};
RTC_DateTypeDef myDate = {0};
```

A HAL definiu as estruturas “RTC_TimeTypeDef” e “RTC_DateTypeDef” em “Drivers/STM32Hxx_HAL_Driver/Inc/stm32h7xx_hal_rtc.h” para gravar ou ler hora e data.

9. Note que logo antes da linha `/* USER CODE BEGIN 2 */`, temos a função “MX_RTC_Init()”, que inicializa o RTC com a data e hora configurados, bem como o alarme. Entretanto, a função “MX_RTC_Init()” não consegue reescrever os parâmetros do RTC a não ser que seja realizada uma “deinicialização” (desativação e desabilitação) explícita. Assim, vamos realizar esta etapa de “deinicialização”, seguida de uma reinicialização e da definição dos parâmetros, para que, caso a placa NUCLEO já tenha executado um programa com RTC, o atual programa consiga reescrever os parâmetros do RTC. Abaixo da linha `/* USER CODE BEGIN 2 */`, digite:

```
HAL_RTC_DeInit(&hrtc);
HAL_RTC_Init(&hrtc);
MX_RTC_Init();
```

10. Assim, só precisamos ler periodicamente os valores do RTC. Dentro do *loop* infinito, abaixo da linha `/* USER CODE BEGIN 3 */`, digite:

```
HAL_RTC_GetTime(&hrtc, &myTime, RTC_FORMAT_BIN);
HAL_RTC_GetDate(&hrtc, &myDate, RTC_FORMAT_BIN);
```

Com estas linhas, vamos ler a hora e a data, além de incrementar o contador.

11. Vamos implementar a função de *callback* que é chamada pela ISR do alarme. No projeto, abra a pasta “Core” e depois a pasta “Startup”. Dê um duplo-clique no arquivo “startup_stm32h7a3zitxq.s” e no painel “Outline” à direita para buscar os nomes das ISRs que atendem as solicitações IRQ=41 e IRQ=3. Elas são, respectivamente, `RTC_Alarm_IRQHandler` e `RTC_WKUP_IRQHandler`. Vá para a pasta “Core/Src” e abra o arquivo “stm32h7xx_it.c” no painel “Outline” à direita. Procure no painel as duas ISRs.

```
/**
  * @brief This function handles RTC wake-up interrupt through EXTI line.
  */
void RTC_WKUP_IRQHandler(void)
{
  /* USER CODE BEGIN RTC_WKUP_IRQn 0 */

  /* USER CODE END RTC_WKUP_IRQn 0 */
  HAL_RTCEx_WakeUpTimerIRQHandler(&hrtc);
  /* USER CODE BEGIN RTC_WKUP_IRQn 1 */

  /* USER CODE END RTC_WKUP_IRQn 1 */
}

/**
  * @brief This function handles RTC alarms (A and B) interrupt through EXTI line.
  */
void RTC_Alarm_IRQHandler(void)
{
  /* USER CODE BEGIN RTC_Alarm_IRQn 0 */

  /* USER CODE END RTC_Alarm_IRQn 0 */
  HAL_RTC_AlarmIRQHandler(&hrtc);
  /* USER CODE BEGIN RTC_Alarm_IRQn 1 */

  /* USER CODE END RTC_Alarm_IRQn 1 */
}
```

Como vimos no projeto anterior, as duas ISRs chamam as respectivas funções do HAL, além de permitir que o desenvolvedor insira as suas instruções. Abra a pasta “Drivers”, depois a pasta “STM32H7xx_HAL_Driver” e a pasta “Src”. Dê um duplo-clique no arquivo “stm32h7xx_hal_rtc.c” para abrí-lo no painel “Outline” à direita. Procure o protótipo da função `HAL_RTC_AlarmIRQHandler` que determinará a chamada do *callback* registrado `HAL_RTC_AlarmAEventCallback`. Copie a linha da definiçãoda função sem o modificador “__weak”. Voltando ao arquivo “main.c”, logo abaixo da linha `/* USER CODE BEGIN 4 */`, digite:

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc){
	HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET);
}
```

Com isso, quando ocorrer a interrupção de alarme, o LED verde será aceso, sobrepondo a função com o mesmo nome definida em “STM32H7xx_HAL_Driver/Src/stm32h7xx_hal_rtc.c” com qualificativo `weak`.

12. Para a função de *callback* periódica (*wakeup*), a definição da função está no arquivo-fonte das funções **estendidas** do RTC, ou seja, em “stm32h7xx_hal_rtc_ex.c”, na mesma pasta, Abra este arquivo e no painel “Outline” clique em `HAL_RTCEx_WakeUpTimerEventCallback`. Faça como no item anterior, e abaixo da função escrita no item anterior (e ainda dentro da seção USER CODE 4), escreva a função:

```
void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef * hrtc) {
	static uint8_t ledon = 0;

	if(ledon) {
		HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
		ledon = 0;
	} else {
		HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
		ledon = 1;
	}
```

```
}
```
que sobreporá a função com o mesmo nome definida em "STM32H7xx_HAL_Driver/Src/stm32h7xx_hal_rtc_ex.c" qualificada como `weak`.

13. Vamos agora otimizar a eficiência do código. Em vez de ler o RTC a cada iteração do *loop*, faremos isso periodicamente a cada segundo. Para isso, siga os passos abaixo:

- declarar uma variável global na seção de variáveis privadas PV para controlar a leitura do RTC:
```
/* USER CODE BEGIN PV */
uint8_t status = 1;
/* USER CODE END PV */
```
- adicionar uma condição de leitura no *loop* infinito para verificar quando a leitura do RTC deve ser realizada:
```
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */
/* USER CODE BEGIN 3 */
  if (status) {
       HAL_RTC_GetTime(&hrtc, &myTime, RTC_FORMAT_BIN);
       HAL_RTC_GetDate(&hrtc, &myDate, RTC_FORMAT_BIN);
       status = 0;
  }
 }
/* USER CODE END 3 */
```
- atualizar `status` no *callback* a cada segundo:
```
void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef * hrtc) {
static uint8_t ledon = 0;
if(ledon) {
       HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
       ledon = 0;
} else {
       HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
       ledon = 1;
}
status = 1;
}
```

13. Faça o "Build", transfira o arquivo executável no modo "Debug".

14. Coloque um *breakpoint* na linha da instrução "while (1)". Continue ("Resume") a execução. Na pausa, verifique se os valores dos seguintes registradores na aba SFRs estão compatíveis com a configuração programada.

Registrador	Endereço mapeado	Função do registrador	Valor
SCB_AIRCR			
NVIC_ICER0			
NVIC_ICER1			
NVIC_IPR0			
NVIC_IPR10			
RCC_CR			
RCC_CFGR			
RCC_BDCR			
RCC_CSR			
RTC_PRER			
RTC_CR			

14. Desloque o *breakpoint* para "if (status) {". Veja na aba "Variables" os elementos das *structs* "myTime" e "myDate". Alternando entre "Resume" e "Continue", você poderá observar a variação dos valores em ambas as structs. Veja que a contagem dos segundos avança mesmo com o programa interrompido.

15. Remova o *breakpoint* e dê um "Resume". Use um relógio para verificar que o LED amarelo pisca com a frequência de 0,5Hz (1s aceso e 1s apagado), confirmando a interrupção periódica de 1s. Espere o relógio avançar até o instante do alarme e o LED verde acenderá, pois a ISR de alarme foi executada.

16. Pause e defina um *breakpoint* dentro da ISR `RTC_WKUP_IRQHandler` no arquivo `Core\Src\stm32h7xx_it.c`. Continue ("Resume") a execução. Ao pausar a execução e utilizar a função "Step Into" para rastrear a `HAL_RTCEx_WakeUpTimerIRQHandler()`, você encontrará as instruções responsáveis por alternar o estado do LED amarelo. Note que, ao contrário dos roteiros anteriores, há processamento adicional de chamadas de *callback* antes que as instruções de interesse sejam executadas diretamente. Esse processamento extra pode introduzir latência no sistema.

17. Vamos agora ver uma abordagem para reduzir essa latência, combinando o uso de HAL com CMSIS. Abra `Core\Src\stm32h7xx_it.c` e insira instruções na seção <ISR 0> em `RTC_WKUP_IRQHandler`, conforme a seguir:

```
void RTC_WKUP_IRQHandler(void)
{
 /* USER CODE BEGIN RTC_WKUP_IRQn 0 */
       static uint8_t ledon = 0;
       if(ledon) {
              GPIOE->BSRR = GPIO_BSRR_BR1;
              ledon = 0;
       } else {
               GPIOE->BSRR = GPIO_BSRR_BS1;
              ledon = 1;
       }
       status = 1;
 /* USER CODE END RTC_WKUP_IRQn 0 */
 HAL_RTCEx_WakeUpTimerIRQHandler(&hrtc);
```

```
 /* USER CODE BEGIN RTC_WKUP_IRQn 1 */
 /* USER CODE END RTC_WKUP_IRQn 1 */
}
```

e na ISR `RTC_Alarm_IRQHandler`:

```
void RTC_Alarm_IRQHandler(void)
{
 /* USER CODE BEGIN RTC_Alarm_IRQn 0 */
 GPIOB->BSRR = GPIO_BSRR_BS0;
 /* USER CODE END RTC_Alarm_IRQn 0 */
 HAL_RTC_AlarmIRQHandler(&hrtc);
 /* USER CODE BEGIN RTC_Alarm_IRQn 1 */
 /* USER CODE END RTC_Alarm_IRQn 1 */
}
```

Além disso, inclua a declaração da variável global na seção EV como variável externa do arquivo `Core\Src\stm32h7xx_it.c`, uma vez ela já está declarada no arquivo `main.c`:

```
59 /* USER CODE BEGIN EV */
60 extern uint8_t status;  /* nao eh uma boa pratica !!!! */
61 /* USER CODE END EV */
```

Volte para `main.c` e remova as definições dos *callbacks*.

```
353 /* USER CODE BEGIN 4 */
354 //void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc){
355 //  HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET);
356 //}
357 //
358 //void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef * hrtc) {
359 //  static uint8_t ledon = 0;
360 //
361 //  if(ledon) {
362 //      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
363 //      ledon = 0;
364 //  } else {
365 //      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
366 //      ledon = 1;
367 //  }
368 //  status = 1;
369 //}
370
371 /* USER CODE END 4 */
```

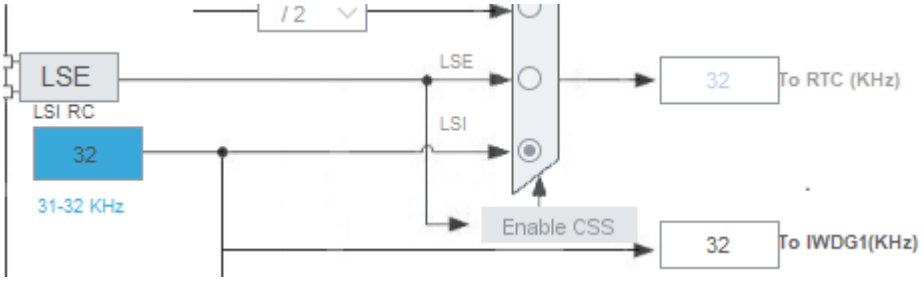
18. Faça o “Build”, transfira o arquivo executável no modo “Debug”. Em seguida, repita o procedimento anterior, definindo um *breakpoint* em `RTC_WKUP_IRQHandler` para verificar a quantidade de níveis de chamadas nesta versão até chegar às instruções propriamente ditas. Com base nas observações feitas até agora, qual estratégia você adotaria para otimizar o uso das técnicas CMSIS e HAL?

Projeto usando *Watchdog*

1. Vamos criar um projeto implementando um *watchdog*, que irá recebed o *feed* manualmente, ao se apertar o botão azul da placa. O sistema é iniciado e espera 5 segundos para acender o LED verde. O programa então entra em *loop* infinito lendo o estado do botão azul e permanece assim até um

reset. Caso o botão azul não seja pressionado em intervalos menores que 5 segundos, o sistema passará por *reset* e o LED verde apagará, reiniciando o ciclo. O LED amarelo acende enquanto o botão azul estiver pressionado, para que o usuário confirme que o botão foi realmente rpessionado. . Crie um projeto com o STM32CubeMX, como os dois projetos anteriores, com o nome "Watchdog". Ative o *Debug*, como visto anteriormente.

2. Clique na aba de "Clock Configuration" para ver a árvore de *clock*. Note que na parte superior há um bloco chamado "LSI RC", com a caixa em azul e o valor 32 dentro. Note que este é o oscilador interno independente usado pelo *watchdog* independente (IWDG) desta MCU.



3. Voltando à aba "Pinout & Configuration", vamos preparar o *watchdog*. No grupo "System Core", selecione o módulo "IWDG1". No painel "Mode", selecione a caixa "Activated", e no painel "Configuration", selecione o valor "64" para "IWDG counter clock prescaler". Assim, teremos os 32kHz divididos por 64, resultando em 500Hz.

4. Queremos que o *watchdog* espere até 5 segundos pelo *feed*. Assim, precisamos contar até 2500. No campo "IWDG down-counter reload value", escreva o valor "2500". O valor de "IWDG window value" deve ser mantido em 4095, que é o valor máximo (o *watchdog* conta em 12 *bits*), para que a função de *window* seja desativada. Salve e gere o código.

5. No arquivo "main.c", note que antes da linha /* USER CODE BEGIN 2 */, temos a função "MX_IWDG1_Init()" . Ou seja, o *watchdog* é iniciado antes mesmo do código escrito pelo programador. Queremos uma espera de cerca de 5 segundos antes de acender o LED para que um *reset* do programa seja claramente identificado. Vamos criar uma variável local para um *loop*. Após a linha /* USER CODE BEGIN 1 */, escreva:

```
uint8_t i;
```

6. Após a linha /* USER CODE BEGIN 2 */, escreva:

```
for(i = 0; i < 5; i++) { // 5 refreshes a cada 1s para aguardar e acender o LED
        HAL_Delay(1000);
        HAL_IWDG_Refresh(&hiwdg1);
  }
  HAL_GPIO_WritePin(LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET); // Acende o LED
```

Assim, vamos gerar 5 esperas de 1 segundo, com um *feed* ao final de cada espera, para evitar o *reset*. Ao final da espera total, o LED verde é aceso.

7. No *loop* infinito, vamos realizar um *polling* do botão azul, que irá realizar o *feed* quando for pressionado. Abaixo da linha /* USER CODE BEGIN 3 */, escreva:

```
while(!(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin))); // espera apertar o botao
HAL_IWDG_Refresh(&hiwdg1); // FEED THE DOG!
HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
while(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin)); // espera soltar o botao
```

```
HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
```

Assim, a cada pressão do botão, ocorrerá um *feed* (O LED amarelo irá piscar para confirmar o *feed*). Se o botão não for apertado periodicamente, o *watchdog* irá reiniciar o programa, apagando o LED e esperando 5 segundos para acendê-lo.

8. Realize o "Build" e o "Debug" e execute o programa. Após o LED acender, aperte o botão em intervalos menores que 5 segundos. Depois deixe o tempo passar sem apertar o botão e note que, após 5 segundos do último acionamento, o LED irá apagar e o programa irá reiniciar, comprovando o funcionamento do *watchdog*.

9. Continue ("Resume") a execução. Após o LED acender, aperte o botão em intervalos menores que 5 segundos. Depois deixe o tempo passar sem apertar o botão e note que, após 5 segundos do último acionamento, o LED irá apagar e o programa irá reiniciar, comprovando o funcionamento do *watchdog*.

TEMPORIZADORES (*TIMERS*)

Os **temporizadores digitais** estão intimamente relacionados aos circuitos contadores. A operação básica de um temporizador envolve a contagem de um número *N* de ciclos de relógio com uma frequência conhecida *f*, o que nos permite calcular o correspondente intervalo de tempo *t* por meio de:

$$t = N \times (1 / f)$$

Se for substituída a fonte de sinais de relógio por uma fonte de eventos externos, o mesmo circuito do temporizador pode ser adaptado para contar eventos provenientes de fontes externas.

Todo temporizador possui um valor inicial (geralmente zero) e um valor limite, geralmente chamado **valor de referência** (REF). Esses parâmetros definem o intervalo de tempo que o temporizador irá monitorar ou controlar. A diferença, em módulo, entre esses valores estabelece a duração do intervalo de tempo. Isso é usualmente regulado por meio de um **comparador**. O comparador compara o valor atual do temporizador com o REF, desencadeando ações específicas quando ocorre uma correspondência. Outra nomenclatura usada para o valor de referência é o **módulo** (**MOD**) **de contagem**. Entende-se como o módulo de contagem o número máximo que um contador pode atingir antes de ser reiniciado, ou o intervalo completo de uma contagem cíclica.

Muitos temporizadores incluem divisores de frequência, conhecidos como **pré-escala** (em inglês, ***prescaler***), que têm a finalidade de reduzir a frequência dos tiques de relógio (em inglês, *clock ticks*) provenientes da fonte. Além disso, alguns temporizadores possuem um circuito integrado que divide a frequência dos *overflows*/*underflow* do contador, tornando mais espaçados os momentos em que *overflows*/*underflows* ocorrem. Esse circuito é denominado **pós-escala** (em inglês, ***postscaler***). Levando em consideração esses divisores, pode-se definir o período de um temporizador como o intervalo de tempo T necessário para que ele complete uma contagem até que ocorra um evento de *overflow*:

$$\mathbf{T = MOD \times (}\boldsymbol{prs}\mathbf{ / f) \times }\boldsymbol{pos}$$

sendo *prs* e *pos* os valores dos divisores de frequência *prescaler* e *postscaler*, respectivamente. A expressão nos mostra que é possível controlar o período de um temporizador por 4 parâmetros: MOD, prs, pos e f. Mais especificamente, fixados f, prs e pos, pode-se controlar a periodicidade de interrupções T de um temporizador através da configuração de REF:

$$\mathbf{MOD = T \times (f / (}\boldsymbol{prs} \times \boldsymbol{pos}\mathbf{))}$$

Temporizadores podem funcionar em um dos modos de contagem: contagem **regressiva** (em inglês, *downcounter***)**, onde o tempo diminui a partir do valor limite superior MOD até o limite inferior, tipicamente zero, ou contagem **progressiva** (em inglês, *upcounter*), onde o tempo aumenta a partir de zero até MOD. Quando uma contagem progressiva atinge o valor de referência (MOD) e reinicia do zero, geralmente esse evento é chamado de ***overflow***. Em contraste, em uma contagem regressiva, quando a contagem atinge zero e reinicia em MOD, esse evento é frequentemente referido como ***underflow***. Portanto, os termos *underflow* e *overflow* são utilizados para descrever situações em que o contador ultrapassa o limite inferior ou superior, respectivamente, e retorna ao início do intervalo. Outra abordagem, conhecida como **bidirecional**, possibilita a configuração do contador para realizar contagens tanto progressivas quanto regressivas, alternando entre esses modos conforme necessário. Nesse caso, o mesmo contador pode ser empregado para realizar contagens ascendentes (progressivas) ou descendentes (regressivas), dependendo da configuração do sistema ou das condições específicas da aplicação.

Assim, um temporizador básico normalmente é controlado por um conjunto de registradores:
Registrador de Controle: Permite a configuração do modo de operação, ativação e desativação do temporizador, forma de contagem e a definição de ações em eventos específicos.
Registrador de Comparação: Usado para definir valores de comparação que, quando atingidos pelo contador, acionam eventos, como interrupções.
Registrador de Estado: O conteúdo do registrador de estado é frequentemente atualizado automaticamente pelo circuito do temporizador em resposta aos eventos que ocorrem durante a operação, como *underflow/overflow*, comparações, disparos externos etc.
Registrador de Contagem: Quando lido apresenta o valor atual no contador; se escrito, carrega o valor no contador.
Registrador de Módulo (*Auto-reload*): Contém o valor máximo a ser contado, quando então ocorre o *reset* do contador.
Registradores de *prescaler*/*postcaler*: Determinam os fatores de divisão da frequência no *prescaler* e no *postscaler*.

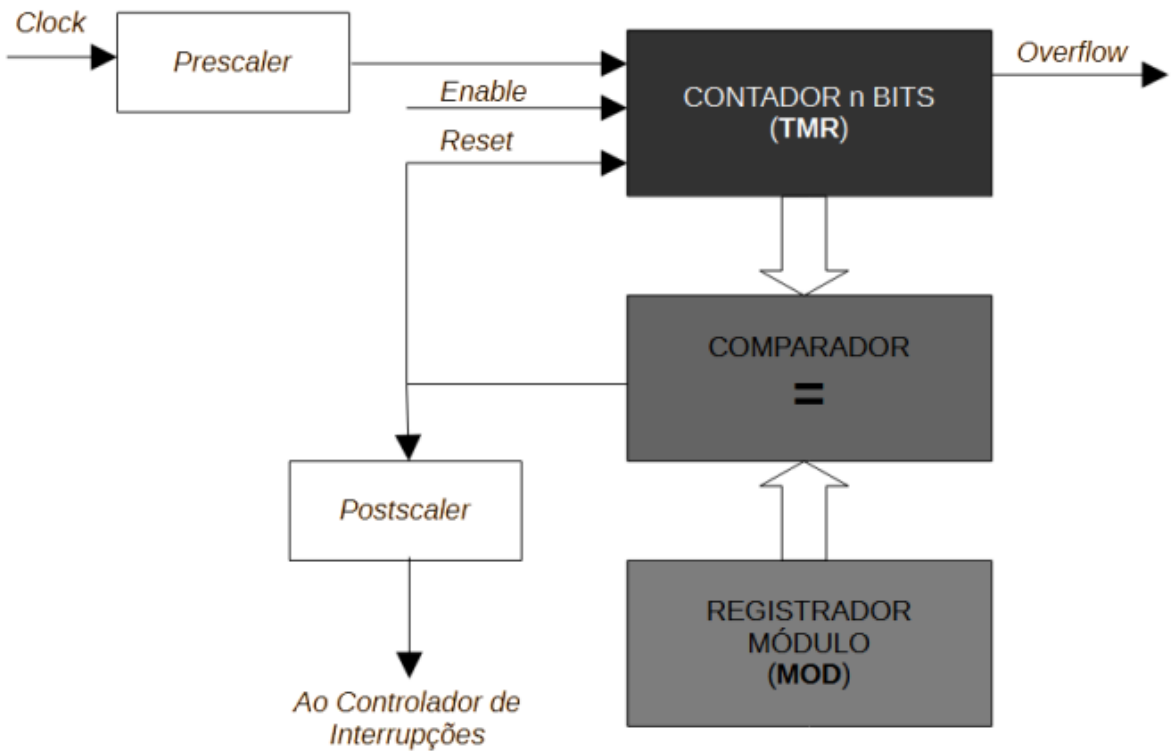


Diagrama de blocos de um temporizador básico

A **precisão** e a **resolução** do tempo controlado pelo temporizador são influenciadas pelo número de *bits* que compõem o contador. Quanto maior o número de *bits*, maior é a quantidade de valores distintos que o temporizador pode representar. Isso resulta em duas possíveis melhorias: uma contagem mais fina para uma mesma duração do intervalo de tempo ou a capacidade de aumentar a duração do intervalo de tempo mantendo uma resolução adequada. Essa característica é de importância fundamental em aplicações onde a precisão temporal é crucial, como em sistemas de controle, medições de tempo e sincronização de eventos em sistemas microcontrolados.

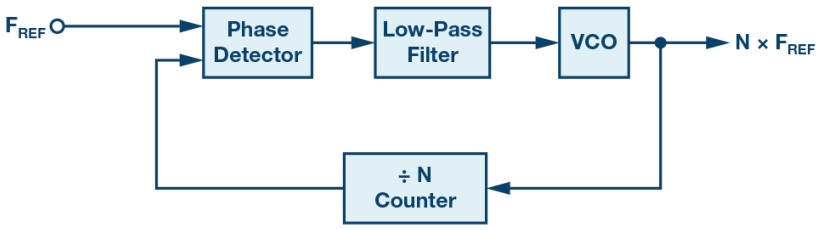
Os eventos de *underflow* e *overflow* em um temporizador podem ser síncronos ou assíncronos em relação aos sinais de relógio do sistema. Se eles estão diretamente sincronizados com os sinais de relógio do sistema, diz-se que são **síncronos**. Isso significa que esses eventos ocorrem em momentos específicos em relação ao ciclo de relógio do sistema. Geralmente, essa sincronização é utilizada em sistemas onde a temporização precisa ser controlada e coordenada de maneira rigorosa. Se os eventos de *underflow* e *overflow* ocorrem independentemente do relógio do sistema, eles são considerados **assíncronos**. Isso oferece mais flexibilidade em situações em que a temporização não precisa ser rigidamente vinculada ao ciclo de relógio principal. Dentro do espectro de temporizadores assíncronos, nos quais os sinais de relógio operam independentemente do sistema principal, é possível ainda diferenciar entre eventos síncronos e assíncronos com base em seus próprios sinais de relógio internos. Quando os eventos do temporizador, como uma escrita no contador, são executados em momentos específicos e bem definidos em relação aos sinais de relógio internos, são classificados como eventos síncronos. Por outro lado, eventos assíncronos ocorrem independentemente dos sinais de relógio internos do temporizador.

FONTES DE SINAIS DE RELÓGIO PARA PERIFÉRICOS

As fontes de relógio são fundamentais no funcionamento dos temporizadores em circuitos eletrônicos, sendo essenciais para a operação sincronizada desses dispositivos. Sem uma fonte de relógio adequada, que fornece o sinal de temporização necessário, os temporizadores não conseguiriam realizar suas funções de contagem e controle de tempo. Microcontroladores modernos incorporam uma ampla gama de periféricos, cada um com necessidades específicas de temporização para funcionar de maneira eficaz. A diversidade desses periféricos exige uma variedade de sinais de relógio, cada um adaptado para otimizar o desempenho do módulo correspondente. Cada periférico, seja um temporizador, ADC (Conversor Analógico-Digital), UART (do inglês *Universal Asynchronous Receiver-Transmitter*) ou PWM (do inglês *Pulse Width Modulation*), pode ter requisitos de frequência e precisão diferentes. Para atender a essas demandas, o microcontrolador deve ser capaz de gerar e distribuir diversos sinais de relógio.

Os sinais de relógio podem ser provenientes de diferentes fontes internas ou externas ao microcontrolador. Internamente, o microcontrolador pode ter osciladores de alta precisão, como o oscilador de cristal ou o oscilador RC (Resistor-Capacitor), que geram sinais de relógio básicos. Esses sinais básicos são frequentemente utilizados como referência para a geração de sinais de relógio adicionais através de divisores de frequência ou PLLs (do inglês *Phase-Locked Loops*), que ajustam a frequência do sinal para atender às necessidades específicas de cada periférico.

Mencionamos no Roteiro 1 que o PLL é um sistema de controle de frequência que gera sinais de relógio com alta precisão e estabilidade, ajustando a frequência de um oscilador com base em um sinal de referência. O circuito é composto por um comparador de fase (ou detector de fase) que mede a diferença entre a fase do sinal de referência e a fase do sinal realimentado (o sinal do VCO dividido). O sinal de erro resultante é filtrado por um filtro de *loop* (normalmente um filtro passa-baixo) para suavizar e eliminar ruídos. O VCO (do inglês *Voltage Controlled Oscillator*, oscilador controlado por voltagem) gera um sinal cuja frequência é ajustada com base no sinal de controle proveniente do filtro de *loop*. Divisores de frequência são usados para dividir a frequência do sinal do VCO e compará-la com a frequência do sinal de referência. O PLL trabalha de forma a sincronizar e estabilizar a frequência gerada, ajustando-a para ser uma frequência múltipla ou submúltipla da frequência do sinal de referência.



TIMERS: MODOS DE OPERAÇÃO

Entre os diversos modos de operação dos *timers*, o modo *slave* e o modo *gated* são bastante comuns e têm características distintas. No **modo *slave***, o *timer* opera sincronizado com um *clock* externo, em vez de usar seu próprio relógio interno. Esse modo é ideal quando é necessário que o *timer* funcione em sincronismo com outro dispositivo ou sistema que fornece o sinal de *clock*. O controle do *timer* no modo *slave* é realizado por sinais externos que determinam quando o *timer* deve iniciar, parar ou reiniciar. Esse modo é frequentemente utilizado em contadores, onde o *timer* conta pulsos provenientes de um sinal externo. É particularmente útil para aplicações que exigem uma sincronização precisa com eventos externos ou outros sistemas temporizadores.

Por outro lado, no **modo *gated***, o funcionamento do *timer* é condicionado por um sinal de habilitação. Quando o sinal de habilitação está ativo, o *timer* conta normalmente; quando o sinal está inativo, a contagem é interrompida ou congelada. Esse modo permite que a contagem do *timer* ocorra apenas durante períodos específicos, quando o sinal de habilitação está presente. Portanto, a contagem é feita de forma condicional, avançando somente quando o sinal de habilitação está ativo. O modo *gated* é adequado para situações onde é necessário registrar a duração de um evento ou operação somente quando certas condições são atendidas.

A escolha entre o modo *slave* e o modo *gated* depende das necessidades específicas da aplicação. O modo *slave* é preferido quando se deseja a sincronização com um *clock* externo para garantir a operação coordenada do *timer* com outros dispositivos ou eventos. Por outro lado, o modo *gated* é mais apropriado quando a contagem ou medição de tempo precisa ocorrer sob condições específicas definidas pelo sinal de habilitação. Ambos os modos oferecem flexibilidade e funcionalidade para diferentes cenários em sistemas digitais, permitindo que os *timers* atendam a uma variedade de requisitos e condições operacionais.

TIMERS: FUNCIONALIDADES ESPECÍFICAS

Eles podem ser classificados de acordo com suas funcionalidades específicas, que incluem *Input Capture*, *Output Compare* e *PWM* (do inglês, *Pulse Width Modulation*). Além disso, tipos de *timers* como o RTC (do inglês, *Real-Time Clock*) e o PIT (do inglês, *Periodic Interrupt Timer*) são importantes em contextos específicos e têm suas próprias características distintas. Esta classificação de *timers* revela as capacidades diversas desses componentes em sistemas digitais,

atendendo a diferentes requisitos de controle e temporização em projetos eletrônicos e sistemas embarcados.

A funcionalidade ***Input Capture*** permite que o *timer* registre o valor atual do contador quando ocorre um evento externo, como uma transição de sinal. Outra funcionalidade importante é o ***Output Compare***, que possibilita ao *timer* comparar seu valor atual com um valor predefinido armazenado em um registrador. Quando o valor do *timer* atinge o valor de comparação, uma ação específica é executada, como gerar um pulso ou alterar o estado de uma saída. A ***PWM*** é uma funcionalidade crítica que permite ao *timer* gerar sinais de saída com uma largura de pulso variável. O controle da duração do pulso em cada ciclo permite ajustar a proporção entre o tempo em que o sinal está ativo e o tempo em que está inativo.

Além dessas funcionalidades, existem tipos específicos de timers que oferecem capacidades especializadas. O **RTC** é um *timer* projetado para manter a contagem contínua do tempo real, mesmo quando o sistema está desligado. Ele é utilizado para fornecer informações precisas sobre data e hora, mantendo a temporização constante e confiável em sistemas que precisam dessa precisão. O **PIT**, por outro lado, é um *timer* mais básico, frequentemente utilizado para gerar interrupções periódicas ou uma única interrupção no modo *one-shot*. No modo *one-shot*, o PIT gera uma interrupção após um intervalo de tempo específico e para automaticamente, necessitando de reinício manual para gerar outra interrupção. No modo periódico, ele gera interrupções a intervalos regulares, sendo ideal para aplicações que requerem acionamento contínuo e repetitivo de eventos.

STM32H7A3

Os microcontroladores STM32H7A3 incluem módulos de temporizadores (*timers*) avançados que oferecem alta flexibilidade e desempenho para controle em tempo real. Estes *timers* são projetados para atender a diferentes aplicações, como geração de sinais PWM, medição de entrada, contagem de eventos e controle de motores. Por trás desses timers, há um módulo dedicado que gera sinais de relógio de diferentes naturezas, garantindo a sincronização e precisão necessárias para cada tipo de aplicação.

Fontes de distribuição de sinais de relógio

O microcontrolador STM32H7A3Z possui um conjunto diversificado de fontes de relógio para garantir a operação estável e eficiente dos seus periféricos. Ele inclui , além de um oscilador interno de alta velocidade (HSI, do inglês *High-Speed Internal*) e um oscilador de baixa velocidade interno (LSI, do inglês *Low-Speed Internal*). Adicionalmente, o microcontrolador conta com um oscilador de baixa velocidade CSI (do inglês *Clock Security System Internal*), que oferece sinais de relógio estáveis de baixa velocidade quando as outras fontes não estão disponíveis. O sistema de segurança dos sinais de relógio (CSS, do inglês *Clock Security System*) é responsável por monitorar a integridade dos sinais de relógio, especialmente o oscilador HSE, garantindo a confiabilidade do sistema.

No STM32H7A3Z, o periférico RCC (do inglês *Reset and Clock Control*) é responsável pela gestão da geração de sinais de relógio e *reset* em todo o microcontrolador, abrangendo tanto a geração quanto a distribuição desses sinais de acordo com as necessidades dos periféricos. É constituído por dois blocos, o bloco de reset e o bloco de geração de sinais de relógio. No que diz respeito ao Bloco de *Reset*, ele é projetado para gerar sinais de reinicialização (*reset*) tanto locais quanto do sistema, oferecendo a capacidade de realizar reinicializações bidirecionais de pinos. Isso permite que o microcontrolador ou dispositivos externos sejam reinicializados conforme necessário. Além disso, o bloco de reinicialização suporta eventos de reinicialização através dos *watchdogs* e é capaz de gerenciar reinicializações ativadas por *Power-On Reset* (POR) e *Brown-Out Reset* (BOR), com controle supervisionado pelo módulo de energia (PWR). *Power-On Reset* (POR) é um tipo de reinicialização que ocorre automaticamente quando o microcontrolador é ligado, enquanto *Brown-Out Reset* (BOR) é uma reinicialização que acontece quando a tensão de alimentação cai abaixo de um nível crítico predeterminado, protegendo o microcontrolador contra funcionamento inadequado ou falhas operacionais que podem ocorrer devido a uma queda na tensão de alimentação.

Em relação ao Bloco de Geração de Sinais de Relógio, RCC é responsável pela geração e distribuição de sinais de relógio para todo o dispositivo. O bloco é equipado com três PLLs independentes, que podem utilizar proporções inteiras ou fracionárias, e permite o ajuste das proporções fracionárias em tempo real. Para otimizar o consumo de energia, o bloco inclui um sistema de **controle de clock** (em inglês, ***clock gating***), incluindo sinais de controle individualizados para habilitar ou desabilitar o sinal de relógio para cada periférico do microcontrolador. A geração de sinais de relógio é facilitada por dois osciladores externos: o oscilador de alta velocidade externo (HSE, do inglês *High-Speed External*), que suporta uma ampla gama de cristais de frequência de 4 a 50 MHz, e o oscilador de baixa velocidade externo (LSE, do inglês *Low-Speed External*), destinado a cristais de 32 kHz. Além desses, há quatro osciladores internos: o oscilador interno de alta velocidade (HSI, do inglês *High-speed internal oscillator*), o oscilador RC de 48 MHz (HSI48), o oscilador interno de baixa potência (CSI) e o oscilador interno de baixa velocidade (LSI, do inglês *Low-speed internal oscillator*). O bloco também inclui saídas de *clock* com *buffer* para dispositivos externos e gera dois tipos distintos de linhas de interrupção: uma dedicada para o gerenciamento da segurança do *clock* e uma linha geral para outros eventos. Por fim, o bloco gerencia a geração de *clock* nos modos `Stop` e `Standby` e adota uma estratégia de gerenciamento de energia que proporciona uma operação eficiente e econômica ao manter o microcontrolador funcional para tarefas críticas enquanto reduz o consumo geral de energia (modo autônomo do domínio *SmartRun*).

A figura do Manual oferece uma visão geral da geração de diversas fontes de sinais de relógio para o processador e periféricos (no lado direito), mostrando como esses sinais são derivados dos osciladores internos e externos (no lado direito). Na figura, os blocos e os símbolos de

multiplexador representam pontos configuráveis por meio dos registradores de configuração do módulo RCC. Os nomes com uma seta vertical, acima desses elementos, correspondem aos campos específicos dos registradores. Por exemplo, `HSION` é um *bit* no registrador RCC_CR.

8.7.1 RCC source control register (RCC_CR)

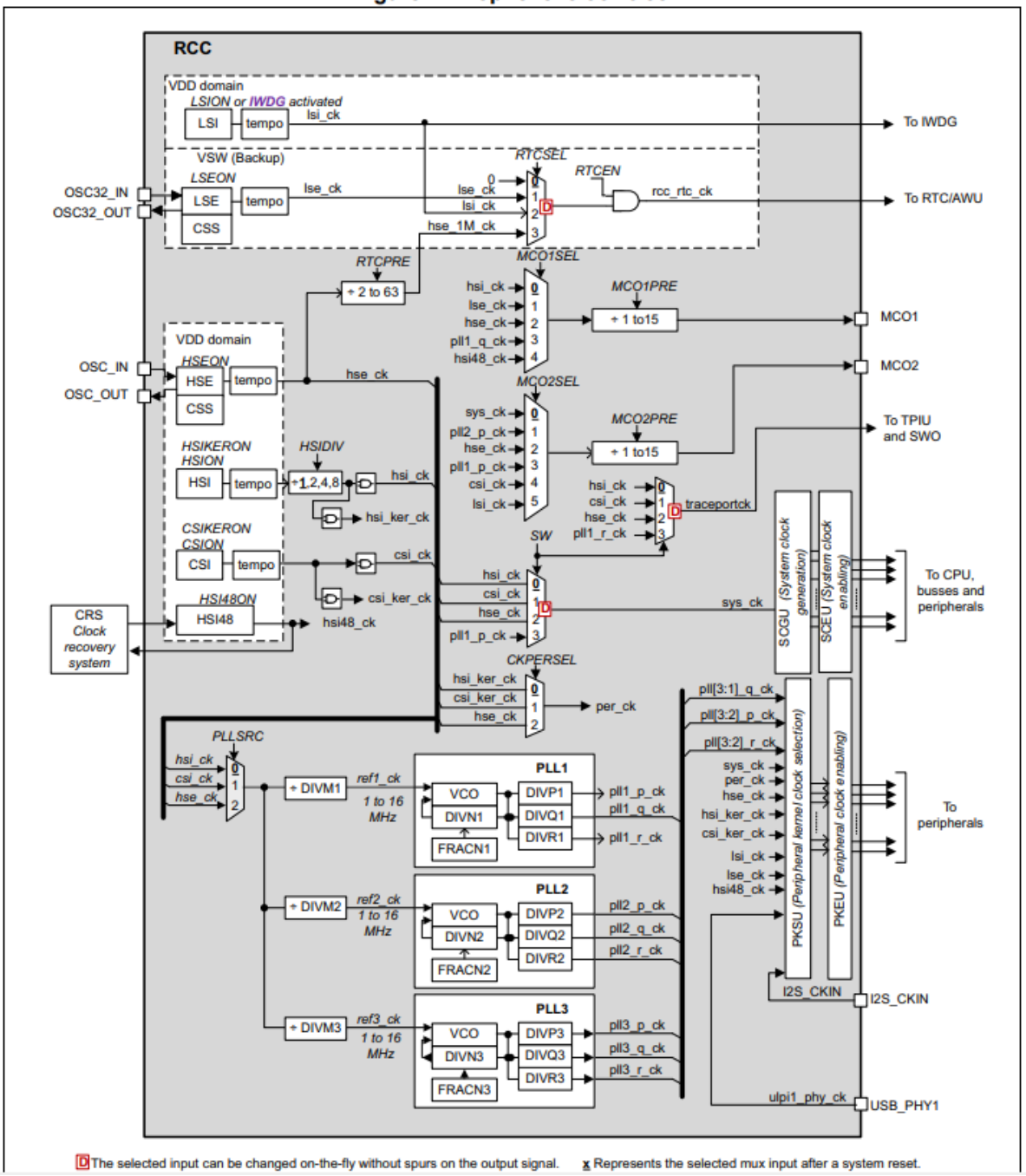
Address offset: 0x000

Reset value: 0x0000 0025

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	PLL3RDY	PLL3ON	PLL2RDY	PLL2ON	PLL1RDY	PLL1ON	Res.	Res.	Res.	HSEEXT	HSECSSON	HSEBYP	HSERDY	HSEON
		r	rw	r	rw	r	rw				rw	rs	rw	r	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CDCKRDY	CPUCKRDY	HSI48RDY	HSI48ON	Res.	Res.	CSIKERON	CSIRDY	CSION	Res.	HSIDIVF	HSIDIV[1:0]		HSIRDY	HSIKERON	HSION
r	r	r	rw			rw	r	rw		r	rw	rw	r	rw	rw

Figure 47. Top-level clock tree



Outros registradores, relevantes para a configuração de fontes de sinais de relógio e integrados no módulo RCC, são RCC_BDCR, RCC_CFGR, RCC_CSR, RCC_PLLCKSELR, RCC_CDCCIPR, RCC_CDCFGR1, RCC_CDCFGR2 e RCC_SRDCFGR.

8.7.24 RCC Backup domain control register (RCC_BDCR)

Address offset: 0x070

Reset value: 0x0000 0000

Reset by Backup domain reset.

Access: 0 ≤ wait state ≤ 7, word, half-word and byte access. Wait states are inserted in case of successive accesses to this register.

After a system reset, the RCC_BDCR register is write-protected. To modify this register, the DBP bit in the *PWR control register 1 (PWR_CR1)* must be set to 1. RCC_BDCR bits are only reset after a Backup domain reset (see *Section 8.4.6: Backup domain reset*). Any other internal or external reset does not have any effect on these bits.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	VSWRST
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTCEN	Res.	Res.	Res.	Res.	Res.	RTCSEL[1:0]		LSEEXT	LSECSSD	LSECSSON	LSEDRV[1:0]		LSEBYP	LSERDY	LSEON
rw						rwo	rw	rw	r	rs	rw	rw	rw	r	rw

8.7.5 RCC clock configuration register (RCC_CFGR)

Address offset: 0x010

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MCO2SEL[2:0]			MCO2PRE[3:0]				MCO1SEL[2:0]			MCO1PRE[3:0]				Res.	Res.
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TIMPRE	Res.	RTCPRE[5:0]						STOPKERWUCK	STOPWUCK	SWS[2:0]			SW[2:0]		
rw		rw	rw	rw	rw	rw	rw	rw	rw	r	r	r	rw	rw	rw

8.7.25 RCC clock control and status register (RCC_CSR)

Address offset: 0x074

Reset value: 0x0000 0000

Access: 0 ≤ wait state ≤ 7, word, half-word and byte access

Wait states are inserted in case of successive accesses to this register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	LSIRDY	LSION
														r	rw

Bits 31:2 Reserved, must be kept at reset value.

Bit 1 **LSIRDY**: LSI oscillator ready

Set and reset by hardware to indicate when the low-speed internal RC oscillator is stable.
This bit needs 3 cycles of **lsi_ck** clock to fall down after LSION has been set to 0.
This bit can be set even when LSION is not enabled if there is a request for LSI clock by the clock security system on LSE or by the low-speed watchdog or by the RTC.
0: LSI clock is not ready (default after reset)
1: LSI clock is ready

Bit 0 **LSION**: LSI oscillator enable

Set and reset by software.
0: LSI is OFF (default after reset)
1: LSI is ON

8.7.9 RCC PLLs clock source selection register (RCC_PLLCKSELR)

Address offset: 0x028

Reset value: 0x0202 0200

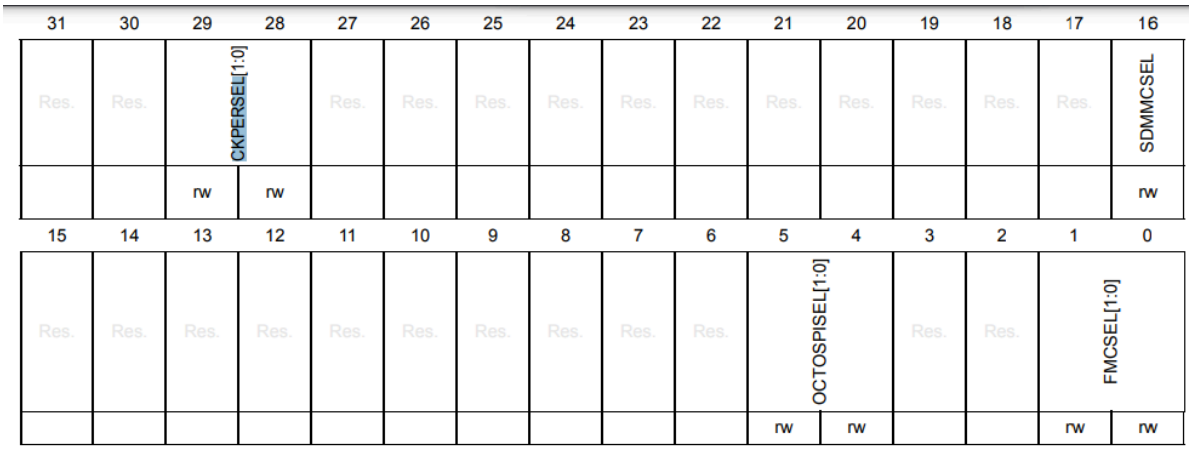
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	DIVM3[5:0]						Res.	Res.	DIVM2[5:4]	
						rw	rw	rw	rw	rw	rw			rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIVM2[3:0]				Res.	Res.	DIVM1[5:0]						Res.	Res.	PLLSRC[1:0]	
rw	rw	rw	rw			rw	rw	rw	rw	rw	rw			rw	rw

8.7.17 RCC CPU domain kernel clock configuration register (RCC_CDCCIPR)

Address offset: 0x04C

Reset value: 0x0000 0000

Changing the clock source on-the-fly is allowed and does not generate any timing violation. However the user must make sure that both the previous and the new clock sources are present during the switching, and during the whole transition time. Refer to *Clock switches and gating*.



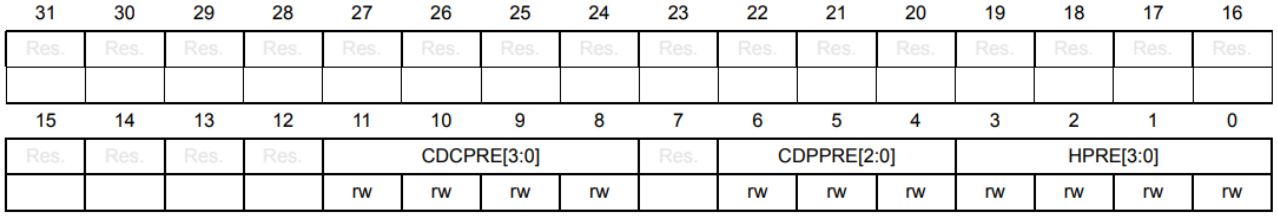
Bits 31:30 Reserved, must be kept at reset value.

Bits 29:28 **CKPERSEL[1:0]**: **per_ck** clock source selection
00: **hsi_ker_ck** selected as **per_ck** clock (default after reset)
01: **csi_ker_ck** selected as **per_ck** clock
10: **hse_ck** selected as **per_ck** clock
11: reserved, the **per_ck** clock is disabled

8.7.6 RCC CPU domain clock configuration register 1 (RCC_CDCFGR1)

Address offset: 0x018

Reset value: 0x0000 0000



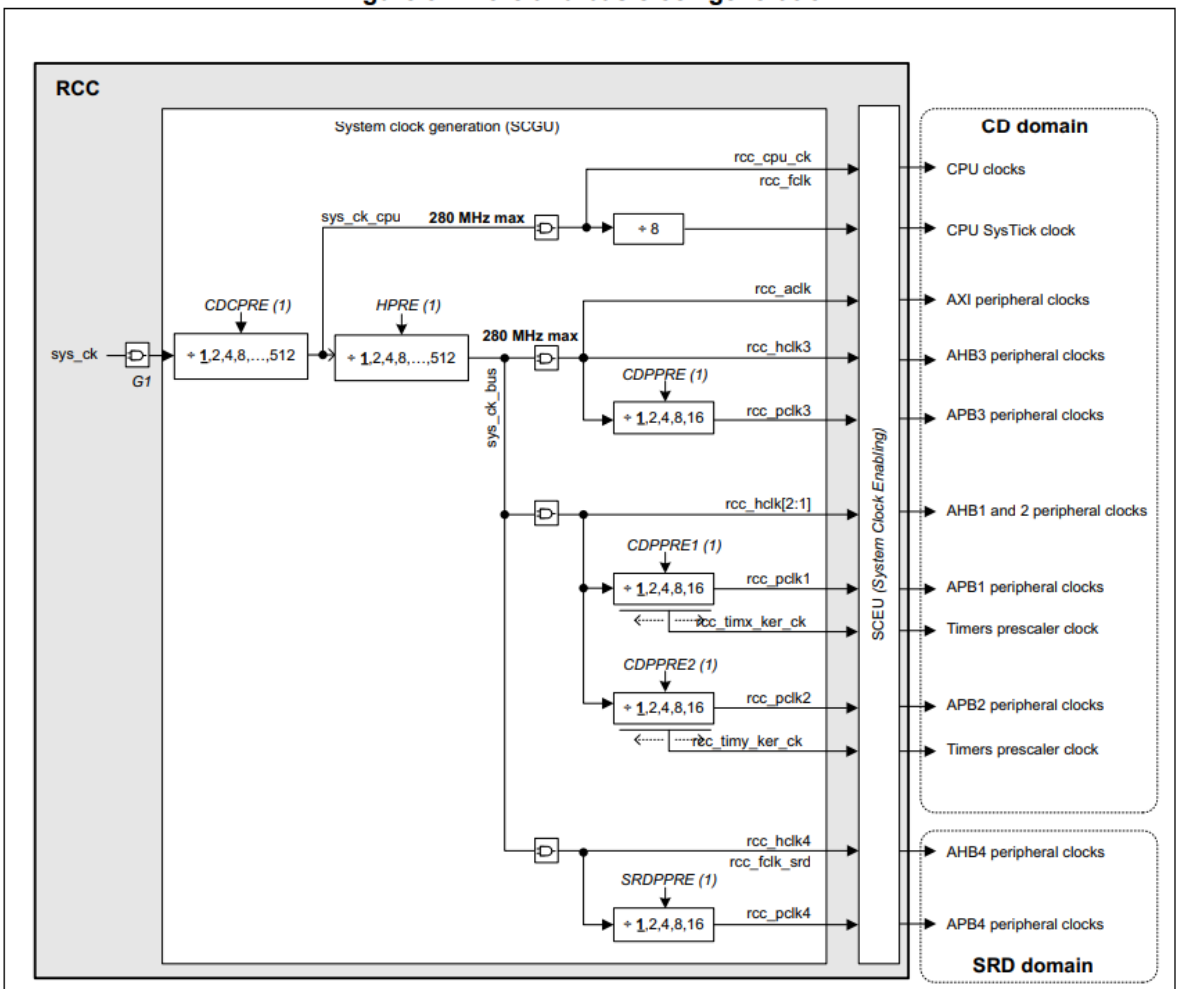
Segundo a descrição funcional do Manual de Referência, após uma reinicialização do sistema, o HSI é selecionado como o relógio do sistema e todos os PLLs são desligados. Quando uma fonte de relógio é utilizada para o sistema, o *software* não pode desativar essa fonte usando os *bits* xxxON. No entanto, o relógio do sistema pode ser interrompido pelo *hardware* quando o sistema entra nos modos *Stop* ou *Standby*.

Enquanto o sistema está ativo, o aplicativo do usuário pode escolher entre quatro fontes para o relógio do sistema (sys_ck): HSE, HSI, CSI ou pll1_p_ck/PLLCLK. Esta seleção é controlada pela programação do campo `RCC_CFGR_SW` do registrador de configuração do relógio `RCC_CFGR`. A troca de uma fonte de relógio para outra só ocorre quando a fonte de destino está pronta, o que significa que o relógio deve estar estável após um atraso de inicialização ou o PLL deve estar bloqueado. Caso a fonte de relógio selecionada ainda não esteja pronta, a troca será realizada assim que a fonte de relógio estiver estabilizada. Os *bits* de estado `RCC_CFGR_SWS` indicam qual relógio

está atualmente sendo usado como relógio do sistema. Além disso, outros *bits* de estado `RCC_CR_*RDY` no registrador RCC_CR mostram quais relógios estão prontos.

A Figura 54 do Manual de Referência ilustra uma visão mais detalada da distribuição de *clock* para a CPU e os barramentos. Todos os divisores (nomes com setas verticasi) apresentados no diagrama de blocos podem ser ajustados rapidamente sem causar violações de tempo. Esse recurso oferece uma solução eficiente para adaptar as frequências dos barramentos às necessidades específicas da aplicação, permitindo a otimização do consumo de energia. O *prescaler* RCC_CDCFGR1_CDCPRE é utilizado para ajustar a frequência do *clock* da CPU, mas esse ajuste também afeta a frequência de clock de toda a matriz de barramento. Da mesma forma, o *prescaler* RCC_CDCFGR1_HPRE permite ajustar o *clock* para a matriz de barramento do domínio da CPU, com impacto também na frequência de *clock* da matriz de barramento do domínio *SmartRun*. A maioria dos *prescalers* é controlada pelos registradores RCC_CDCFGR1, RCC_CDCFGR2 e RCC_SRDCFGR. Todos campos de *bits* que podem ser alterados dinamicamente são seguidos de (1).

Figure 54. Core and bus clock generation

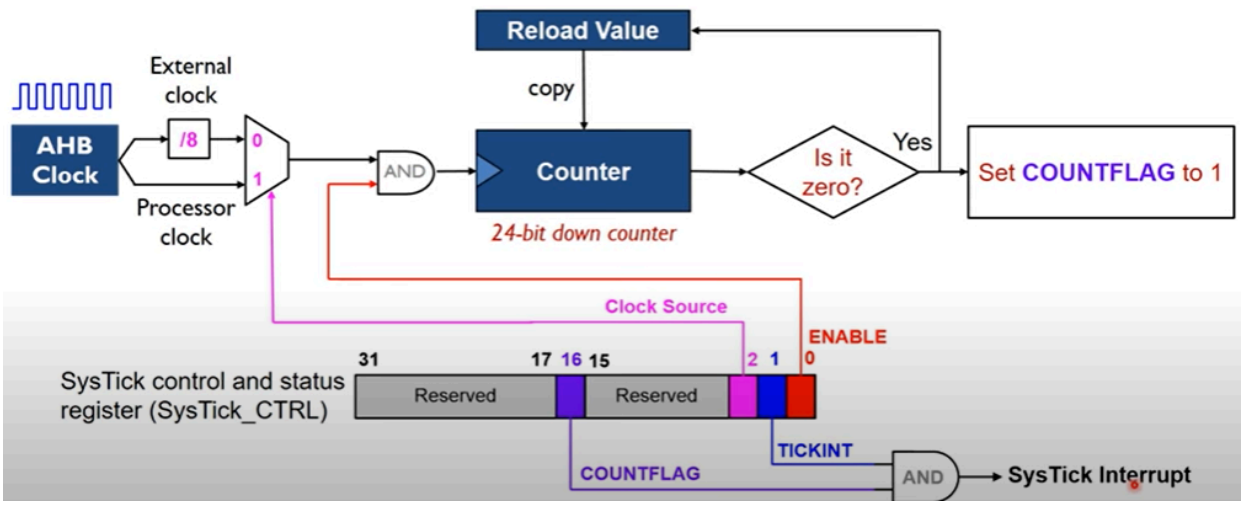


O microcontrolador é equipado com dois conjuntos principais de barramentos, como mostra a figura 1 do *Datasheet*: o *Advanced High-performance Bus* (AHB) e o *Advanced Peripheral Bus* (APB). O

barramento **AHB** conecta o núcleo do processador e a memória a periféricos de alta velocidade, como a memória FLASH e a SRAM, e é subdividido em grupos como AHB1, AHB2, AHB3 e AHB4. Cada um desses grupos atende a diferentes periféricos e requisitos de desempenho. O barramento **APB**, por sua vez, conecta periféricos de menor velocidade e com requisitos de largura de banda menos críticos, como periféricos de comunicação e temporizadores, e é subdividido em APB1, APB2, APB3 e APB4. Cada um desses barramentos é projetado para atender às especificidades e necessidades de largura de banda dos dispositivos conectados. Para complementar, o STM32H7A3Z utiliza **matrizes de interconexão** para o roteamento flexível dos sinais entre os barramentos e os periféricos.

Tick do Sistema

Muitos microcontroladores incorporam um temporizador especial, denominado *tick do sistema*, ou simplesmente ***Systick***. O *SysTick* é um temporizador de sistema projetado especificamente para gerar interrupções periódicas com uma configuração extremamente simples, com a finalidade de gerar tempos de espera com bastante precisão, ou ainda, controlar a cadência de sistemas operacionais em tempo real (RTOS). O *Systick* dos microcontroladores ARM é padronizado nas várias famílias *Cortex*. Por padrão, ele é um contador (*Counter*) `SYST_CVR` descrescente de 24 *bits*, ligado a um dos barramentos AHB, sem *postscaler* e com um *prescaler* com apenas duas opções: sem divisão ou divisão por oito. Ele é carregado com um valor definido (*Reload Value*) em um registrador `SYST_RVR` e realiza contagem decrescente. Ao chegar a zero, ele é carregado novamente com o valor e, se habilitado, gera uma interrupção. O valor de contagem que é carregado na função de inicialização do microcontrolador gerada pelo *Cube* é tal que, para a frequência de *clock* configurada, o zero seja atingido a cada 1ms, mas este valor pode ser modificado na configuração do *Cube* ou dentro do código do usuário. Quando o processador é interrompido enquanto está no estado de *Debug*, o contador `SYST_CVR` pára de decrementar.



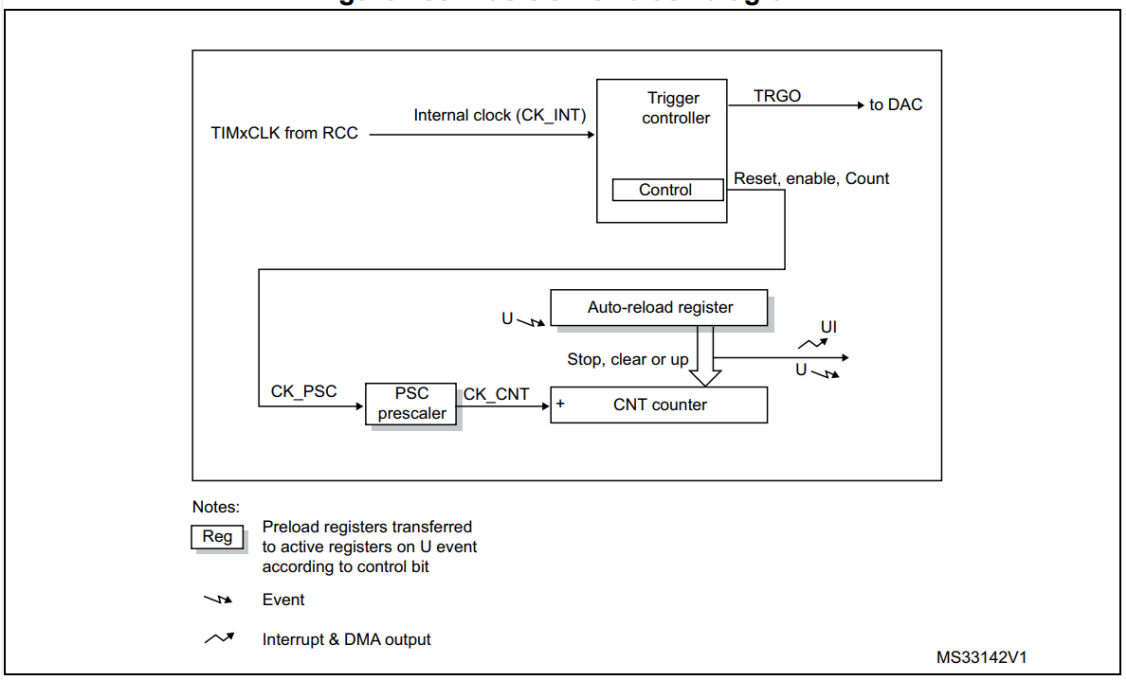
Em relação às fontes de sinais de relógio, configurável pelo registrador SYST_CSR (SysTick_CTRL na figura), o *SysTick* compartilha os mesmos sinais de *clock* do processador, refletindo a sua integração estreita com o núcleo do microcontrolador. Ele é controlado por um *clock* de referência, que pode ser o *clock* do próprio processador (SYST_CSR[2]=1) ou uma fonte de *clock* externa (SYST_CSR[2]=0), dependendo da implementação específica. Caso a implementação utilize um *clock* externo, é essencial que a documentação esclareça o relacionamento entre o clock do processador e a fonte externa. Esta documentação é crucial para a calibração precisa do sistema, considerando fatores como metaestabilidade, desvio do *clock* e *jitter*. A Figura 54 do Manual de Referência mostra que nos microcontroladores STM32HA3/B3 a fonte externa do *SysTick* é a fonte da CPU dividida de 8.

O *SysTick* é um componente integrado no núcleo do processador ARM Cortex-M e faz parte do suporte de *hardware* da arquitetura Cortex-M. Seus eventos são gerados de forma síncrona e tratados como exceções, sem uma linha de interrupção (IRQ) dedicada como todas as exceções. No entanto, a prioridade de atendimento dos eventos do *SysTick* é gerenciada pelo NVIC (Controlador de Interrupções e Exceções). Como o *SysTick* é uma parte da ISA ARM, detalhes sobre sua programação estão disponíveis no Manual de Referência de ARMv7-M.

TIM6/TIM7

O temporizador programável principal de TIM6 e TIM7 é um contador de 16 *bits* de contagem progressiva, que trabalha em conjunto com um registrador de *auto-reload*. O *clock* do contador pode ser dividido por um *prescaler*, e tanto o contador quanto o registrador de *auto-reload* e o registrador do *prescaler* podem ser acessados por leitura e escrita em *software*, mesmo enquanto o contador está em funcionamento.

Figure 488. Basic timer block diagram



A unidade de base de tempo do temporizador inclui:

- Registro do contador ou TMR (TIMx_CNT)
- Registro do *prescaler* (TIMx_PSC)
- Registro de *auto-reload* ou MOD (TIMx_ARR)

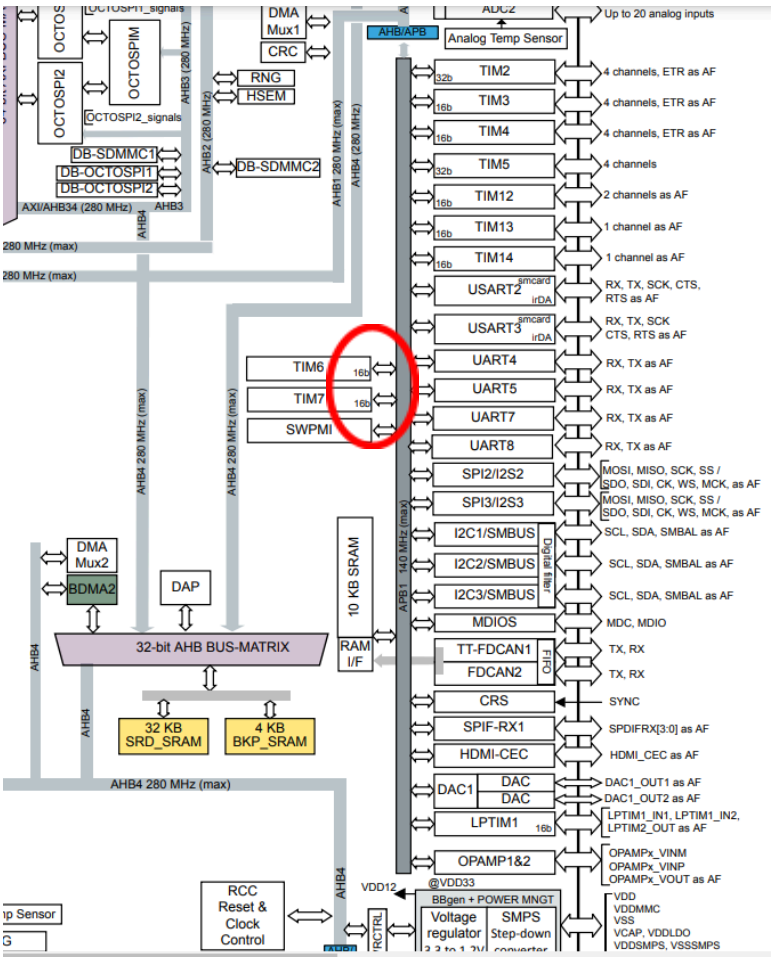
Adicionalmente, são integrados registradores de controle e estado:

- 2 Registradores de Controle (TIMx_CR1 e TIMx_CR2)
- Registrador de habilitação de interrupção e DMA (TIMx_DIER)
- Registrador de estado (TIMx_SR)
- Registrador de geração de evento (TIMx_EGR)

Conforme a descrição funcional no Manual de Referência, o registrador TIMx_ARR pode ser pré-carregado (em inglês "*buffered*"). O conteúdo do registrador de pré-carregamento é acessado a cada acesso de escrita ou leitura de TIMx_ARR. O conteúdo do registrador de pré-carregamento é transferido para TIMx_ARR permanente ou em cada evento de atualização, conforme configurado pelo *bit* TIMx_CR1_ARPE de habilitação de pré-carregamento de *auto-reload* no registrador TIMx_CR1. O evento de atualização (em inglês *Update EVent*) ocorre quando o contador atinge o valor de estouro (*overflow*) e o *bit* TIMx_CR1_UDIS no registrador TIMx_CR1 está configurado como 0. Este evento pode ser gerado por *software* através de TIMx_EGR.

O contador é acionado pela saída do *prescaler* CK_CNT, que só é ativada quando o *bit* TIMx_CR1_CEN de habilitação do contador no registrador TIMx_CR1 está setado em ‘1’. É importante notar que o sinal de habilitação real do contador é ativado um ciclo de *clock* após o *bit* CEN ser definido.

Os dosi temporizadores estão cconectados no barramento APB1. Uma das fontes de *clock* para este barramento é o HSI, cuja frequência padrão é 64 MHz. Esta frequência pode ser dividida por 1, 2, 4 ou 8, usando o campo `RCC_CR_HSIDIV` no registrador `RCC_CR`. Para que o HSI seja utilizado como fonte de *clock*, ele deve estar habilitado e disponível. A habilitação do HSI é controlada pelo *bit* `RCC_CR_HSION` e sua disponibilidade é verificada pelo *bit* `RCC_CR_HSIRDY`, ambos localizados no registrador `RCC_CR`. Por padrão, o HSI é habilitado automaticamente durante a reinicialização (*reset*).



Real Time Clock (RTC)

Outro tipo de temporizador bastante comum em microcontroladores é o **circuito de relógio em tempo real** (do inglês, ***real time clock***, sigla **RTC**), que mantém o registro do tempo corrente (hora, minuto e segundo, e opcionalmente dia, mês e ano), sendo basicamente um relógio digital, que pode ter data e/ou hora ajustadas e consultadas pelo microcontrolador. Para isto, o circuito utiliza um oscilador (muitas vezes separado do oscilador principal por razões que serão explicadas adiante) que fornece uma frequência exata de 1Hz, a qual é aplicada a um contador de módulo-60 para contar os segundos. O *reset* deste contador produz um pulso de *clock* para o próximo contador

módulo-60, que conta os minutos. Este processo segue em cascata para o contador de horas (módulo-24) e o de dias (geralmente de 16 ou 32 bits, contando dias corridos). Geralmente os dias são contados continuamente, e cabe ao *software* fazer o cálculo da data em dias, meses e anos usando os dias corridos, a partir de uma data de referência. Alguns RTCs apenas contam os segundos a partir de uma data de referência, e cabe ao *software* calcular minutos, horas, dias, meses e anos. O RTC não faz parte da ISA ARM, e sua implementação depende apenas do fabricante.

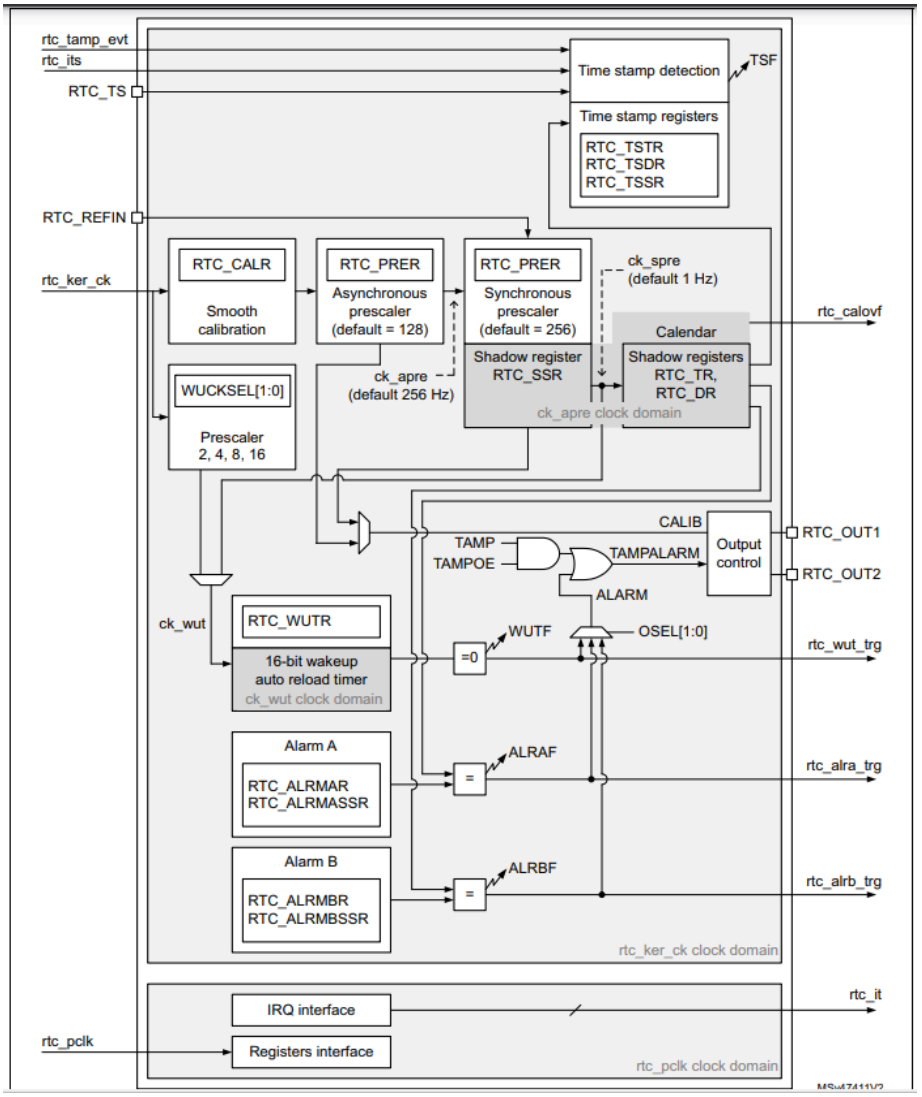
Como a informação de data e hora precisa ser atualizada continuamente, o oscilador e os contadores do RTC (mas não necessariamente a interface) precisam funcionar mesmo na ausência de energia no microcontrolador. Para isso, o microcontrolador possui um pino para que seja conectada uma bateria de *backup*, além da alimentação normal. Assim, quando há energia no microcontrolador, o RTC funciona com esta fonte de energia. Ao desligar a energia, o RTC entra no modo *Backup* e usa a bateria para manter a contagem de tempo. Esta é a primeira razão para existir um oscilador dedicado no RTC, ou seja, para que ele possa continuar funcionando mesmo na ausência de energia no oscilador principal (geralmente este oscilador é projetado para baixo consumo). A segunda razão é que o oscilador principal opera a uma frequência alta (da ordem de unidades ou dezenas de MHz), o que demanda um divisor de frequência mais complexo para reduzir a mesma a 1Hz. Ao utilizar um oscilador de frequência mais baixa, o circuito divisor fica simplificado. Geralmente, é utilizado um oscilador a cristal na frequência de 32.768kHz, criado originalmente para relógios de pulso. Este valor de frequência foi escolhido porque equivale a 2^{15}, e por corresponder a uma potência de dois, um divisor de frequência para 1Hz é bastante simples de ser implementado, e faz parte do módulo RTC, pois precisa receber energia de *backup* quando a alimentação principal está desligada.

O RTC moderno dos microcontroladores STM32 possui proteção de escrita em registradores críticos e nos registradores de dados para garantir a integridade e a estabilidade do sistema de tempo real. Essa proteção é essencial para assegurar que o RTC mantenha a contagem precisa do tempo, evitando qualquer alteração inadvertida ou não autorizada que poderia comprometer o funcionamento correto do sistema. Além disso, a proteção previne modificações acidentais que poderiam resultar de erros de código ou de configuração, garantindo que apenas configurações válidas e seguras possam ser aplicadas.

O RTC é um componente essencial no microcontrolador STM32H7A3ZIT6-Q que gerencia os modos de baixo consumo de energia, oferecendo uma função automática de *wakeup*. Independentemente do estado do dispositivo, seja em modo de execução, modo de baixo consumo ou mesmo durante um *reset*, o RTC continua a funcionar, desde que a tensão de alimentação esteja dentro da faixa operacional. Este temporizador/contador independente em formato BCD (código decimal codificado em binário) oferece um relógio/calendário com alarmes programáveis, garantindo que a contagem do tempo e as interrupções associadas ocorram sem interrupções. Além disso, o RTC opera de maneira eficiente no modo *Backup*, assegurando sua funcionalidade contínua.

Para garantir a integridade e a segurança do sistema, registradores críticos, como o registrador de configuração da fonte de relógio do RTC RCC_BDCR, possuem proteção contra escrita. Essa proteção ajuda a impedir alterações não autorizadas e a proteger o sistema contra possíveis ataques que possam explorar vulnerabilidades na configuração do RTC. Dado que o RTC opera

frequentemente em frequências muito baixas e requer uma configuração precisa, essa proteção é essencial para manter a estabilidade e a precisão do relógio. Para realizar alterações nesses registradores críticos, é necessário configurar o *bit* PWR_CR1_DBP e aguardar que o *bit* se estabilize em “1”.



O sinal de *clock* do RTC, denominado RTCCLK, passa por um estágio de dois divisores, cujos *prescalers* são configuráveis através do registrador RTC_PRER. Esses divisores, o assíncrono `RTC_PRER_PREDIV_A` e o síncrono `RTC_PRER_PREDIV_S`, trabalham em conjunto para gerar sinais de *clock* com frequência de 1 Hz, utilizados na atualização dos registradores de calendário (RTC_DR) e de tempo (RTC_TR). O registrador `RTC_DR` armazena os valores de ano, mês, dia e dia da semana, enquanto o `RTC_TR` contém os valores de hora, minuto e segundo, todos representados no formato BCD. A separação dos divisores em dois não só reduz o consumo de energia, mas também permite a inclusão de um contador de sub-segundos, o RTC_SSR, que utiliza o sinal de *clock* proveniente do divisor `RTC_PRER_PREDIV_A`.

Os acessos de escrita a esses registradores são permitidos apenas no modo de inicialização. Durante esse modo, o *bit* RTC_ICSR_INIFT deve estar em “1”, e a proteção de escrita deve ser desbloqueada ao escrever 0xCA seguido de 0x53 no registrador RTC_WPR. Após a configuração

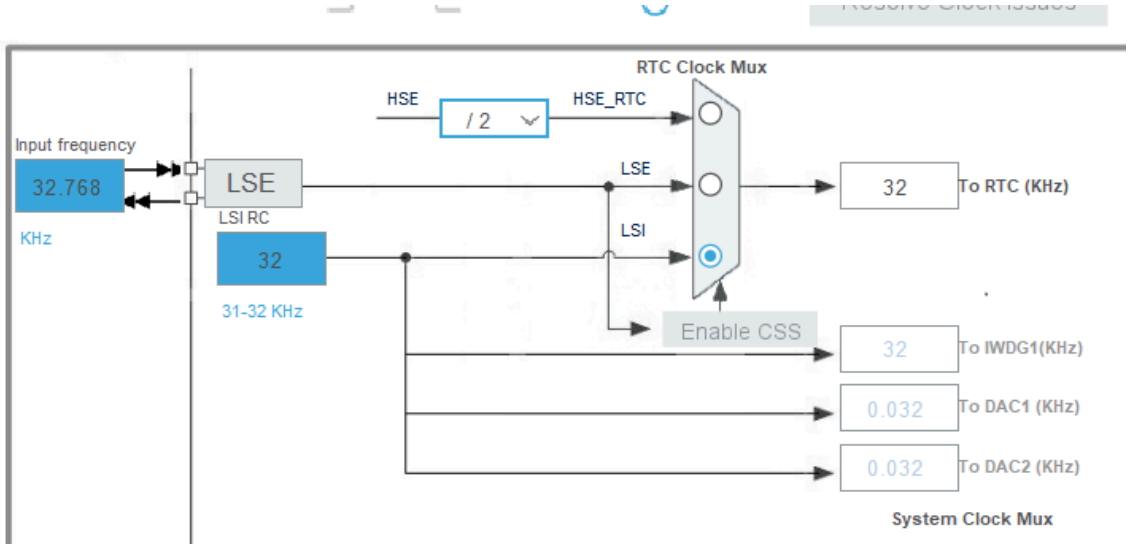
inicial dos registradores de calendário, alarme e/ou *wakeup* (interrupções periódicas), deve-se escrever 0xFF em RTC_WPR e resetar o *bit* RTC_ICSR_INIT em "0". A partir deste momento, os contadores operam livremente e os acessos de leitura aos registradores são feitos indiretamente através dos registradores *shadow*, com uma latência de até 4 vezes o período de RTCCLK. Estes registradores permitem que o RTC continue a contagem do tempo sem interrupções enquanto as leituras e escritas são feitas, e evitam condições de corrida e dados inconsistentes durante atualizações.

O RTC é equipado de 2 alarmes programáveis, designados como alarme A e alarme B. Para habilitar o alarme programável, o *bit* RTC_CR_ALRAE/RTC_CR_ALRBE deve ser ativado. Quando o alarme é configurado, o *bit* RTC_CR_ALRAF/RTC_CR_ALRBF será definido como '1' se os valores de subsegundos, segundos, minutos, horas, data ou dia da semana corresponderem aos valores programados nos registradores de alarme RTC_ALRMASSR/RTC_ALRMBSSR e RTC_ALRMAR/RTC_ALRMBR. Cada campo do calendário pode ser selecionado de forma independente através dos *bits* MSKx no registrador RTC_ALRMAR/RTC_ALRMBR e dos *bits* MASKSSx no registrador RTC_ALRMASSR/RTC_ALRMBSSR. A interrupção do alarme é ativada configurando o *bit* RTC_CR_ALRAIE (IRQ41).

Além disso, o *flag* de despertar (em inglês, *wakeup*) periódico é gerado por um contador programável de 16 *bits* com recarga automática (em inglês *auto-reload*), cujo intervalo pode ser expandido para 17 *bits*. A função de *wakeup* é habilitada através do *bit* RTC_CR_WUTE. A entrada de *clock* para o temporizador de despertar, ck_wut, pode ser o RTCCLK dividido por 2, 4, 8 ou 16, configurável por RTC_CR_WUCKSEL. Quando o RTCCLK é LSE (32.768 kHz), isso permite configurar o período de interrupção de *wakeup* de 122µs a 32s, com uma resolução de até 61µs. Alternativamente, pode-se usar ck_spre (geralmente um *clock* interno de 1 Hz), permitindo um tempo de *wakeup* de 1s a cerca de 36 horas com uma resolução de um segundo. Após completar a sequência de inicialização, o temporizador começa a contagem regressiva. Quando a função de *wakeup* está habilitada, a contagem regressiva permanece ativa em modos de baixo consumo de energia. Quando atinge 0, o *flag* RTC_SR_WUTF é setado no registrador de estado RTC_SR, e o contador de *wakeup* é automaticamente recarregado com seu valor de recarga (valor do registrador RTC_WUTR). O *flag* RTC_SR_WUTF deve então ser limpo por *software* via os correspondentes *bits* no registrador RTC_SCR. Quando a interrupção de *wakeup* periódico é habilitada ao configurar o *bit* RTC_CR_WUTIE (IRQ3), ela pode retirar o dispositivo dos modos de baixo consumo de energia.

A fonte de clock RTCCLK é configurável através do campo RCC_BDCR_RTCSEL e pode ser selecionada entre o oscilador LSE, o *clock* do oscilador LSI ou o *clock* do HSE. O registrador RCC_BDCR, no entanto, é protegido contra acessos de escrita para evitar modificações não autorizadas. Para desbloquear essa proteção, é necessário configurar o *bit* PWR_CR1_DBP como "1". Tipicamente, usa-se o oscilador LSE, pois com um fator de divisão assíncrono definido como 128 e um fator de divisão síncrono como 256, é possível obter uma frequência de clock interna de 1 Hz (ck_spre) a partir de uma frequência LSE de 32,768 kHz. A ativação do LSE é realizada pelo *bit* RCC_BDCR_LSEON. Após a configuração do *bit* PWR_CR1_DBP e a ativação do LSE, deve-se aguardar até que essas ações sejam completamente efetivadas. Além disso, é importante observar

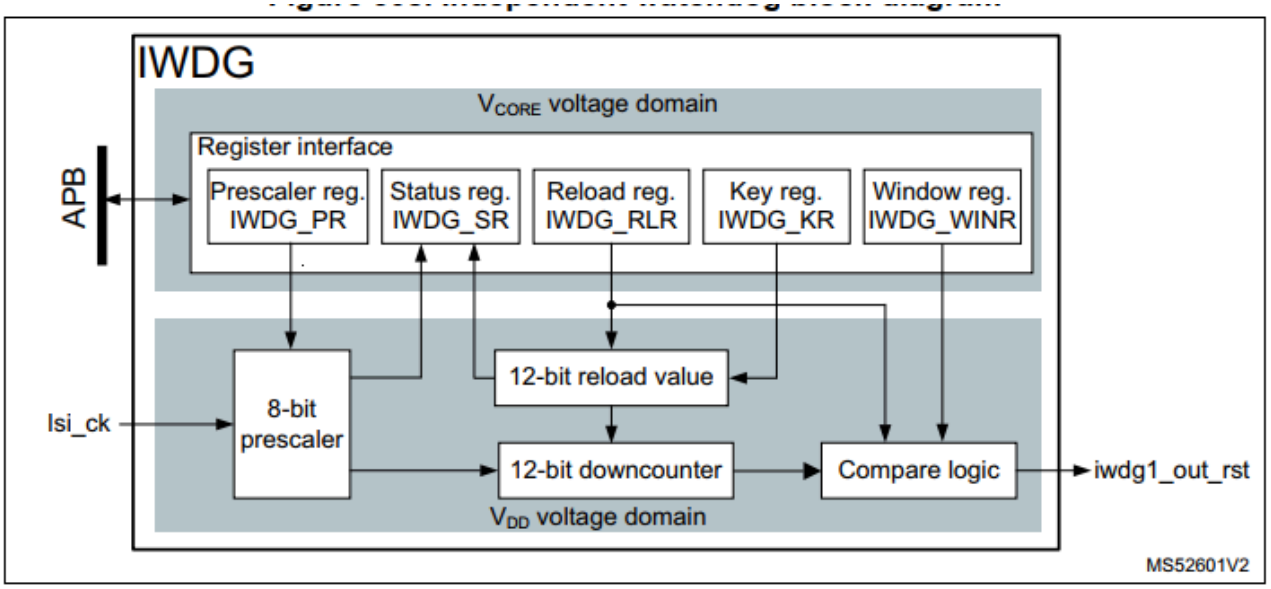
que as atualizações nos registradores RTC_TR, RTC_DR e outros registradores de dados são permitidas somente quando o *bit* RTC_ICSR_INIT está configurado como “1”. Durante esse período, os registradores não são incrementados dinamicamente.



O RTC integrado no STM32H7A3Z oferece várias funcionalidades adicionais que ampliam suas capacidades além da simples contagem de tempo. Entre essas funcionalidades, o RTC inclui um mecanismo de calibração para ajustar sua precisão, garantindo uma medição de tempo mais exata ao longo do tempo, mesmo com variações na frequência do clock. Além disso, o RTC possui uma função de *timestamp* que permite registrar o momento exato de eventos importantes ou alterações no sistema, facilitando a auditoria e o rastreamento de eventos históricos. O módulo também suporta a detecção de *tamper*, um recurso de segurança que ajuda a identificar tentativas de interferência física no dispositivo, aumentando a proteção contra acessos não autorizados ou alterações indevidas no sistema.

Watchdog

O *watchdog* é um *timer* de contagem decrescente, projetado para ser de simples configuração, como o *SysTick*, cuja função é monitorar o sistema, reiniciando o mesmo automaticamente em caso de travamentos. Ele utiliza uma das fontes internas de *clock*, ou, em alguns casos (como o do presente roteiro), um *clock* simples interno e independente (oscilador RC), por segurança. O *clock* pode passar por um *prescaler*, que geralmente oferece apenas algumas opções de fatores de divisão, e o contador, após ativado, irá acionar a linha interna de *reset* da MCU quando o valor de contagem chegar a zero. Para evitar o *reset*, dentro do código o programador deve adicionar, ao longo do programa, linhas de código que reiniciam a contagem do *watchdog* antes que ele chegue a zero (*feed the dog*). Se o programa travar, o *feed* não irá mais acontecer, e, após o tempo necessário para a contagem chegar a zero, a MCU irá reiniciar.



O *feed* do *watchdog* é feito escrevendo-se um valor específico no registrador próprio para enviar comandos ao *watchdog* (registrador IWDG_KR na figura acima). O valor carregado é definido em outro registrador (registrador IWDG_RLR na figura acima). Um terceiro registrador define o valor do *prescaler* e ativa o *watchdog* (registrador IWDG_PR na figura acima). Alguns *watchdogs* ainda contam com um registrador de *window* (registrador IWDG_WINR na figura acima), que define um valor máximo no contador para que o *feed* ocorra. Assim, caso o *feed* ocorra muito cedo (com o valor do contador acima do valor da janela), o *reset* também ocorre. Com o *windowed watchdog*, o sistema é reiniciado não apenas se o *watchdog* não receber o *feed*, mas também se receber o *feed* fora de uma janela de tempo específica. Isso significa que o *feed* do *watchdog* deve ocorrer dentro de um intervalo de tempo permitido, ou seja, não pode ser muito cedo nem muito tarde. Essa funcionalidade impede que o código realize o *feed* do *watchdog* de maneira inadvertida ou muito frequentemente (por exemplo, em um *loop* travado), o que pode ocorrer em situações de falha de *software*.

A H.A.L. E O *STM32CUBEMX*

Muitos fabricantes de microcontroladores oferecem uma IDE própria, ou bibliotecas de funções para suporte mais avançado de seus produtos. A ST disponibiliza uma IDE baseada em Eclipse que inclui as bibliotecas de funções para suporte. Além disso, a IDE conta com um *plugin* da ST denominado **STM32CubeMX**, que permite o desenvolvedor introduzir as funções de configuração dos vários periféricos e incluir as bibliotecas de suporte através de uma interface gráfica. A interface permite a configuração de vários periféricos através do uso de caixas de seleção, caixas de múltipla escolha e entradas de texto e números. A interface gráfica grava as opções do usuário em um arquivo de extensão "**.ioc**". Este arquivo é um texto contendo todas as configurações para a inicialização de periféricos, em uma sintaxe própria.

Agora que os fundamentos da estrutura e da programação de um microcontrolador foram devidamente estudados, podemos passar a trabalhar com um nível mais elevado de código, sem perder o controle sobre os periféricos que é devido em aplicações de sistemas embarcados. Vamos

usar a ***Hardware Abstraction Layer***, ou **Camada de Abstração de *Hardware***, ou ainda com a sigla **HAL**.

Uma HAL é uma interface que permite que o *software* interaja com o *hardware* de maneira uniforme e independente das especificidades do *hardware* subjacente. Ou seja, ela **abstrai** o conceito de *hardware*, permitindo que um código seja mais facilmente portado para outros microcontroladores. Com a abstração, pode-se utilizar funções mais genéricas e em um nível de linguagem mais elevado que o de máquina.

A HAL usa exaustivamente o conceito de *device pointer*. Um *device pointer* é um ponteiro para uma *struct*, cujos elementos são os parâmetros de configuração de um determinado tipo de periférico (o *device*). Um projeto pode conter múltiplos *device pointers* do mesmo tipo, um para cada periférico, havendo um elemento da *struct* que determina a qual dos periféricos disponíveis o *device pointer* se refere, sendo ele elemento denominado **instância** (***instance***).

O outro conceito utilizado pela HAL é o de consistência na sequência de registradores em um mesmo tipo de periférico. Por exemplo, todos os periféricos GPIOx da família H7 possuem a mesma sequência de registradores: MODER, OTYPER, OSPEEDR, PUPDR, IDR, ODR, BSRR, etc. Assim, basta saber o endereço-base de cada porta GPIO para que se possa definir o endereço de qualquer um de seus registradores, somando-se seu *offset*.

Uma função HAL usa a instância do *device pointer* passado a ela para definir o endereço-base e os endereços de todos os registradores envolvidos. Assim, a mesma função HAL atende a todos os periféricos do mesmo tipo, bastando o programador indicar o *device pointer* correspondente. Ou seja, a HAL utiliza a CMSIS como base e implementa funções em um nível de abstração pouco acima da mesma.

Como exemplo, podemos analisar o código gerado pelo STM32CubeMX para inicializar o *timer* TIM6. Inicialmente, na área inicial de "Private variables" é declarado o *device pointer*:

```
TIM_HandleTypeDef htim6;
```

onde "TIM_HandleTypeDef" é a *struct* própria para os periféricos de *timer*, e a sintaxe do ponteiro é "h" para HAL, "tim" para *timer* e "6" para o número 6.
A seguir, o *Cube* gera a função de inicialização (as linhas de comentários que definem as áreas de código de usuário foram aqui omitidas):

```
static void MX_TIM6_Init(void)
{
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 htim6.Instance = TIM6;
 htim6.Init.Prescaler = 48000-1;
 htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim6.Init.Period = 1000-1;
 htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
 if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
 {
   Error_Handler();
```

```
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) !=
HAL_OK)
 {
   Error_Handler();
 }
```

Note que inicialmente a função preenche os elementos do *device pointer* com os valores adequados: Instância, Prescaler, Modo de contagem, Período de contagem, se usa o Preload. Depois chama a função HAL "HAL_TIM_Base_Init", passando como parâmetro o endereço do ponteiro (&htim6). Esta função, de acordo com o manual da HAL, inicializa o *timer*, mas sem iniciar a contagem. Ela retorna um código de erro, que é o valor zero (ou HAL_OK) quando a configuração é bem sucedida.
Depois a função configura uma extensão do *timer*, relativa a sincronização com outros *timers*, no caso desabilitando o uso em modo "escravo", usado para interligar dois *timers* de 16 *bits* como se fossem um único *timer* de 32 *bits*.
Se o Timer 7 fosse utilizado, o *Cube* criaria um *device pointer* chamado htim7, e uma função de inicialização do *timer*, preenchendo o mesmo com os parâmetros adequados, e a instância "TIM7", usando a **mesma função** "HAL_TIM_Base_Init" passando o *device pointer* &htim7 como parâmetro.

Um conceito importante ao trabalhar com a HAL é o `Deinit` (ou "*deinitialization*"). A função `Deinit` tem como objetivo principal liberar recursos e restaurar um periférico ao seu estado inicial, preparando-o para uma reconfiguração segura ou desligamento adequado. Utilizar `Deinit` seguido de `Init` é uma prática comum na programação de sistemas embarcados. Essa abordagem assegura que o periférico ou módulo esteja em um estado conhecido e estável antes de ser reconfigurado, minimizando riscos de conflitos e comportamento inesperado. Ao reinicializar o periférico, você garante que ele opere com configurações atualizadas e em conformidade com os requisitos atuais do sistema.

O uso combinado de CMSIS e HAL proporciona uma abordagem poderosa e eficiente para o desenvolvimento em microcontroladores STM32. O STM32CubeMX, com seu editor gráfico intuitivo, simplifica a configuração dos periféricos, permitindo a configuração rápida e visual de pinos, clocks e periféricos necessários para uma tarefa específica. Isso reduz significativamente o tempo de configuração e diminui a probabilidade de erros na configuração do hardware. Após configurar os periféricos, a interface CMSIS oferece uma eficiente e direta maneira de programar o fluxo de controle do sistema, incluindo o gerenciamento de interrupções e a implementação de ISRs. Ao usar CMSIS para programar as rotinas de interrupção e o controle de fluxo, os desenvolvedores se beneficiam de uma interface de baixo nível otimizada para desempenho, o que resulta em código mais eficiente e com menor sobrecarga. A combinação da configuração simplificada proporcionada pelo STM32CubeMX e da programação de baixo nível facilitada pelo CMSIS permite um desenvolvimento mais ágil, preciso e eficiente, maximizando o desempenho e a confiabilidade do sistema.

ISR e *CALLBACK*

As ISRs (do inglês *Interrupt Service Routine*) e os callbacks são mecanismos essenciais em sistemas embarcados e em programação de sistemas operacionais que lidam com eventos e execuções assíncronas, porém têm diferenças importantes em sua aplicação e funcionamento.

Uma **Interrupção de Serviço** (ISR) é uma rotina especial que é executada automaticamente pelo processador quando um evento específico ocorre, como um sinal de interrupção enviado por um periférico de *hardware*, como um temporizador ou uma comunicação serial. O principal objetivo de uma ISR é lidar com eventos críticos que precisam ser processados imediatamente. Uma ISR é acionada diretamente pelo *hardware*, e sua execução é gerenciada pelo sistema de interrupções do microcontrolador. Como vimos no Roteiro 3, ao entrar na ISR, o processador salva o contexto do programa atual, executa a ISR para tratar o evento e depois restaura o contexto original para continuar a execução do programa principal.

Por outro lado, um ***callback*** é um mecanismo de programação que permite que uma função ou método seja registrado para ser chamado automaticamente em resposta a um evento específico ou após a conclusão de uma operação. Diferentemente das ISRs, *callbacks* são mecanismos de alto nível que são registrados e chamados por sistemas de *software* ou *frameworks*. Eles não lidam diretamente com interrupções de *hardware*, mas sim com o fluxo de execução do *software*, sendo mais comuns em aplicações de programação orientada a eventos e invocados pelo sistema que gerencia o fluxo de controle do programa.

Em microcontroladores, especialmente quando se usa camadas de abstração de *hardware* (HALs, do inglês *Hardware Abstraction Layers*), os *callbacks* são uma técnica comum para gerenciar eventos de forma eficiente e modular. A construção e o uso de *callbacks* em HALs permitem que os desenvolvedores escrevam código que responde a eventos de *hardware* de maneira mais flexível e organizada, sem precisar lidar diretamente com os detalhes do *hardware*. Tipicamente, quando ocorre um evento, uma solicitação de interrupção é gerada. O sistema de interrupção do microcontrolador é responsável por invocar a rotina de atendimento à interrupção (ISR). A ISR verifica o estado do periférico e determina se um *callback* previamente registrado deve ser chamado, além de pré-processar os dados necessários para lidar com o evento pela função de *callback*.

Como exemplo, vamos explorar como as interrupções (ISRs) e *callbacks* são configurados e utilizados para gerenciar temporizadores no STM32H7A3ZIT6-Q. Utilizaremos a implementação no STM32CubeIDE para entender a implementação e o fluxo de controle entre ISRs e *callbacks*. O Manual de Usuário de HAL detalha as funções de *callback* associadas aos temporizadores do STM32H7A3ZIT6-Q, como mostra o excerto a seguir.

88.2.9 TIM Callbacks functions

This section provides TIM callback functions:

- TIM Period elapsed callback
- TIM Output Compare callback
- TIM Input capture callback
- TIM Trigger callback
- TIM Error callback

This section contains the following APIs:

- *HAL_TIM_PeriodElapsedCallback()*
- *HAL_TIM_PeriodElapsedHalfCpltCallback()*
- *HAL_TIM_OC_DelayElapsedCallback()*
- *HAL_TIM_IC_CaptureCallback()*
- *HAL_TIM_IC_CaptureHalfCpltCallback()*
- *HAL_TIM_PWM_PulseFinishedCallback()*
- *HAL_TIM_PWM_PulseFinishedHalfCpltCallback()*
- *HAL_TIM_TriggerCallback()*
- *HAL_TIM_TriggerHalfCpltCallback()*
- *HAL_TIM_ErrorCallback()*
- *HAL_TIM_RegisterCallback()*
- *HAL_TIM_UnRegisterCallback()*

Essas funções são configuradas no STM32CubeIDE e podem ser chamadas pela função `HAL_TIM_IRQHandler`. O `HAL_TIM_IRQHandler` está definido no arquivo "Drivers/STM32H7xx_HAL_Driver/stm32h7xx_hal_tim.h" e é responsável por processar eventos de temporizador.

```
371 #if (USE_HAL_TIM_REGISTER_CALLBACKS == 1)
372   void (* Base_MspInitCallback)(struct __TIM_HandleTypeDef *htim);             /*!< TIM Base Msp Init Callback
373   void (* Base_MspDeInitCallback)(struct __TIM_HandleTypeDef *htim);           /*!< TIM Base Msp DeInit Callback
374   void (* IC_MspInitCallback)(struct __TIM_HandleTypeDef *htim);              /*!< TIM IC Msp Init Callback
375   void (* IC_MspDeInitCallback)(struct __TIM_HandleTypeDef *htim);            /*!< TIM IC Msp DeInit Callback
376   void (* OC_MspInitCallback)(struct __TIM_HandleTypeDef *htim);              /*!< TIM OC Msp Init Callback
377   void (* OC_MspDeInitCallback)(struct __TIM_HandleTypeDef *htim);            /*!< TIM OC Msp DeInit Callback
378   void (* PWM_MspInitCallback)(struct __TIM_HandleTypeDef *htim);             /*!< TIM PWM Msp Init Callback
379   void (* PWM_MspDeInitCallback)(struct __TIM_HandleTypeDef *htim);           /*!< TIM PWM Msp DeInit Callback
380   void (* OnePulse_MspInitCallback)(struct __TIM_HandleTypeDef *htim);        /*!< TIM One Pulse Msp Init Callback
381   void (* OnePulse_MspDeInitCallback)(struct __TIM_HandleTypeDef *htim);      /*!< TIM One Pulse Msp DeInit Callback
382   void (* Encoder_MspInitCallback)(struct __TIM_HandleTypeDef *htim);         /*!< TIM Encoder Msp Init Callback
383   void (* Encoder_MspDeInitCallback)(struct __TIM_HandleTypeDef *htim);       /*!< TIM Encoder Msp DeInit Callback
384   void (* HallSensor_MspInitCallback)(struct __TIM_HandleTypeDef *htim);      /*!< TIM Hall Sensor Msp Init Callback
385   void (* HallSensor_MspDeInitCallback)(struct __TIM_HandleTypeDef *htim);    /*!< TIM Hall Sensor Msp DeInit Callback
386   void (* PeriodElapsedCallback)(struct __TIM_HandleTypeDef *htim);           /*!< TIM Period Elapsed Callback
387   void (* PeriodElapsedHalfCpltCallback)(struct __TIM_HandleTypeDef *htim);   /*!< TIM Period Elapsed half complete Callback
388   void (* TriggerCallback)(struct __TIM_HandleTypeDef *htim);                 /*!< TIM Trigger Callback
389   void (* TriggerHalfCpltCallback)(struct __TIM_HandleTypeDef *htim);         /*!< TIM Trigger half complete Callback
390   void (* IC_CaptureCallback)(struct __TIM_HandleTypeDef *htim);              /*!< TIM Input Capture Callback
391   void (* IC_CaptureHalfCpltCallback)(struct __TIM_HandleTypeDef *htim);      /*!< TIM Input Capture half complete Callback
392   void (* OC_DelayElapsedCallback)(struct __TIM_HandleTypeDef *htim);         /*!< TIM Output Compare Delay Elapsed Callback
393   void (* PWM_PulseFinishedCallback)(struct __TIM_HandleTypeDef *htim);       /*!< TIM PWM Pulse Finished Callback
394   void (* PWM_PulseFinishedHalfCpltCallback)(struct __TIM_HandleTypeDef *htim); /*!< TIM PWM Pulse Finished half complete Callback
395   void (* ErrorCallback)(struct __TIM_HandleTypeDef *htim);                   /*!< TIM Error Callback
396   void (* CommutationCallback)(struct __TIM_HandleTypeDef *htim);             /*!< TIM Commutation Callback
397   void (* CommutationHalfCpltCallback)(struct __TIM_HandleTypeDef *htim);     /*!< TIM Commutation half complete Callback
398   void (* BreakCallback)(struct __TIM_HandleTypeDef *htim);                   /*!< TIM Break Callback
399   void (* Break2Callback)(struct __TIM_HandleTypeDef *htim);                  /*!< TIM Break2 Callback
400 #endif /* USE_HAL_TIM_REGISTER_CALLBACKS */
401 } TIM_HandleTypeDef;
```

A função `HAL_TIM_IRQHandler` está conectada às rotinas de interrupção (ISRs) dos temporizadores do microcontrolador por meio das definições das ISRs. Quando uma interrupção ocorre, o fluxo de controle é direcionado para a ISR correspondente, que então chama a `HAL_TIM_IRQHandler` para tratar o evento. Quando utilizamos o STM32CubeMX, ao criarmos um projeto do tipo STM32CubeIDE, para configurar o nosso projeto, o *software* automatiza a geração de código de configuração, incluindo as definições de rotinas de interrupção (ISRs). Se configuramos a habilitação de IRQ54 correspondente ao temporizador TIM6, é gerada

automaticamente a seguinte conexão entre ISR `TIM6_DAC_IRQHandler` com `HAL_TIM_IRQHandler` em “Core/Src/stm32h7xx_it.c”.

```
/**
  * @brief This function handles TIM6 global interrupt, DAC1_CH1 and DAC1_CH2 underrun error interrupts.
  */
void TIM6_DAC_IRQHandler(void)
{
  /* USER CODE BEGIN TIM6_DAC_IRQn 0 */

  /* USER CODE END TIM6_DAC_IRQn 0 */
  HAL_TIM_IRQHandler(&htim6);
  /* USER CODE BEGIN TIM6_DAC_IRQn 1 */

  /* USER CODE END TIM6_DAC_IRQn 1 */
}

```

Para garantir flexibilidade e evitar erros de compilação, o STM32CubeIDE define todos os *callbacks* relacionados a temporizadores como funções com o qualificativo `weak`. Isso permite que os desenvolvedores sobreponham essas funções padrão com suas próprias implementações personalizadas. Por exemplo, o *callback* `HAL_TIM_PeriodElapsedCallback`, descrito no Manual, pode ser substituído pelo desenvolvedor para atender aos requisitos específicos da aplicação. O Manual de Usuário de HAL especifica que `HAL_TIM_PeriodElapsedCallback` é utilizado para capturar interrupções periódicas sem interromper a execução do código principal, operando em modo não-bloqueante.

HAL_TIM_PeriodElapsedCallback

Function name

void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef * htim)

Function description

Period elapsed callback in non-blocking mode.

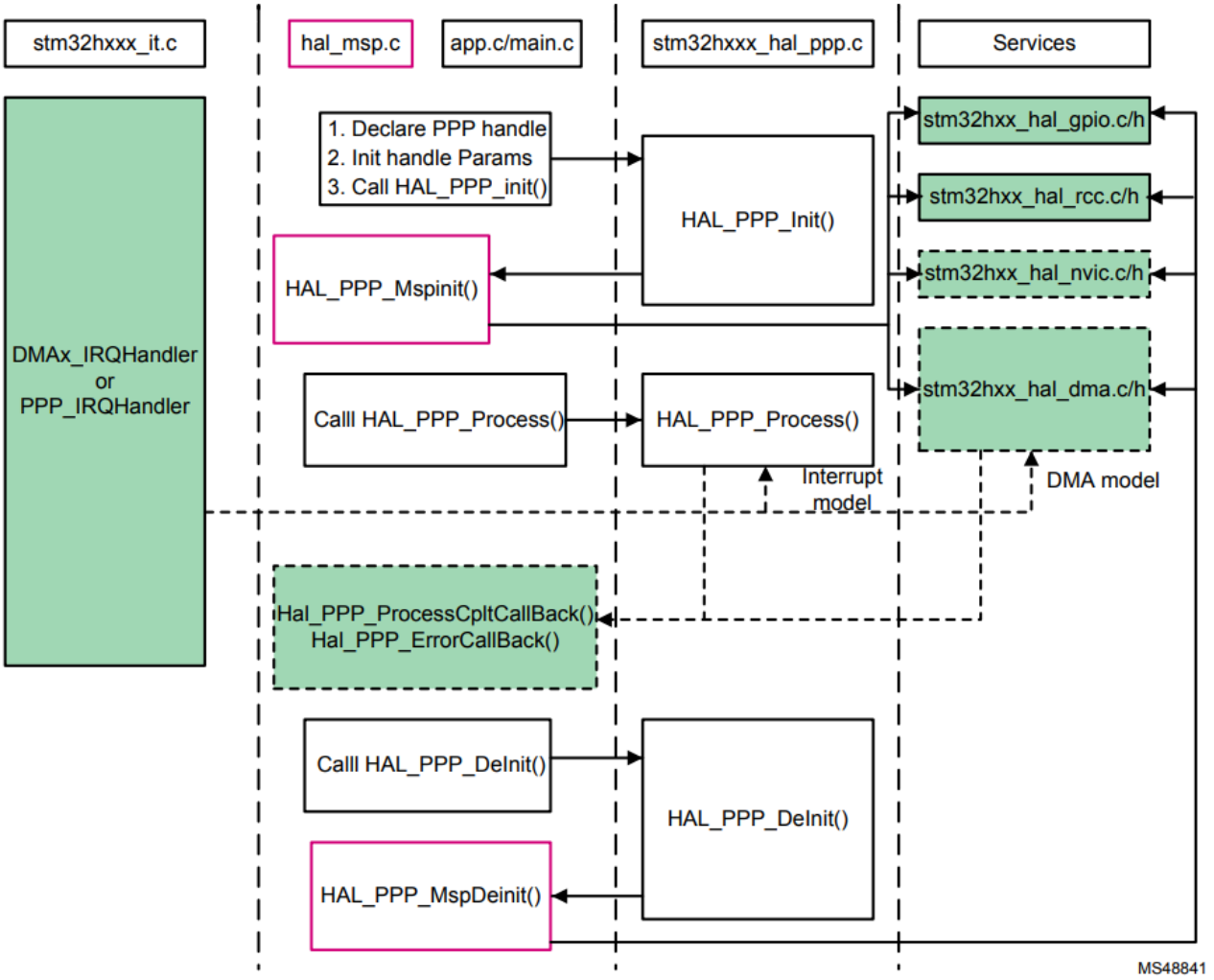
Parameters

- **htim:** TIM handle

Return values

- **None:**

O protótipo de todas as funções de callback, bem como suas descrições funcionais, estão disponíveis no Manual de Usuário de HAL. O desenvolvedor deve selecionar o *callback* mais adequado para sua aplicação e implementar sua própria versão para personalizar o comportamento do tratamento de eventos. Essa abordagem permite que o desenvolvedor adapte a lógica de interrupção de acordo com os requisitos específicos do aplicativo, garantindo uma integração eficiente e personalizada com o *hardware* do microcontrolador.


stm32hxxx_it.c
hal_msp.c
app.c/main.c
stm32hxxx_hal_ppp.c
Services
DMAx_IRQHandler
or
PPP_IRQHandler
1. Declare PPP handle
2. Init handle Params
3. Call HAL_PPP_init()
HAL_PPP_Init()
stm32hxx_hal_gpio.c/h
stm32hxx_hal_rcc.c/h
stm32hxx_hal_nvic.c/h
stm32hxx_hal_dma.c/h
HAL_PPP_Mspinit()
Calll HAL_PPP_Process()
HAL_PPP_Process()
Interrupt
model
DMA model
Hal_PPP_ProcessCpltCallBack()
Hal_PPP_ErrorCallBack()
Calll HAL_PPP_DeInit()
HAL_PPP_DeInit()
HAL_PPP_MspDeinit()
MS48841