DISCIPLINA EA701
Introdução aos Sistemas Embarcados

ROTEIRO 5: Temporizadores Digitais

TIM6/TIM7, SYSTICK, LPTIM1/LPTIM2, RTC e *WATCHDOG*

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

FEEC / UNICAMP

Revisado e modificado em fevereiro de 2025 por Ting com auxílio do Chatgpt
Revisado em agosto de 2024

INTRODUÇÃO 2
PROJETOS-EXEMPLO 3
Projeto com o TIM6 usando CMSIS 4
Projeto com o SysTick usando CMSIS 9
Projeto com o LPTIM usando CMSIS 13
Projeto com o Watchdog usando STM32CubeMX e CMSIS 18
Projeto com RTC usando STM32CubeMX e CMSIS 27
FUNDAMENTOS TEÓRICOS 37
PRINCÍPIOS DE OPERAÇÃO 37
MODOS DE OPERAÇÃO 40
INTEGRIDADE 40
MECANISMOS DE PROTEÇÃO DOS REGISTRADORES 41
PINOS MULTIPLEXÁVEIS 42
FONTES DE SINAIS DE RELÓGIO 43
TEMPORIZADORES DEDICADOS 44
Temporizador de Interrupções Periódicas 44
Temporizador de Interrupções Periódicas Síncronas 44
Temporizador de Pulso Único 45
Temporizador de Vigilância 45
Relógio em Tempo Real 46
STM32H7A3 48
Fontes de Sinais de Relógio 48
TIM6/TIM7 55
Systick (Tick do Sistema) 57
Temporizador de Baixo Consumo (LPTIM) 58
Watchdogs 62
Real Time Clock (RTC) 64
GPIO: Pinos Multiplexáveis 68

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 básicos, temporizadores avançados, temporizadores de vigilância (em inglês, *watchdog*) e relógios em tempo real (em inglês, *Real Time Clock* – RTC). **Temporizadores básicos** 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. O **RTC**, por sua vez, 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 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 quando a fonte principal de energia está desligada. 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

Você já se perguntou como um temporizador digital, realmente funciona? O que faz um temporizador digital contar o tempo com tanta precisão? Seria a programação deles algo simples ou envolve camadas mais profundas de controle? Vamos descobrir neste roteiro as configurações dos

temporizadores do microcontrolador STM32H73A para geração precisa de interrupções periódicas (em inglês, *periodic interrupt timer*), geração de interrupções periódicas síncronas como sinal de relógio do processador (em inglês, *System Tick*), disparo único (em inglês, *one-shot*), relógio digital (em inglês, *real time clock*) e vigilância (em inglês, *watchdog*)?

Projeto com o TIM6 usando CMSIS

Nos Roteiros 1 e 2, controlamos a alternância do LED verde manualmente ou usando um laço de espera programado, sem qualquer precisão no tempo entre cada mudança de estado. Mas agora que sabemos que o STM32H7A3 possui temporizadores digitais integrados e que todos eles podem gerar interrupções via NVIC, surge uma questão interessante: "É possível deixarmos o próprio microcontrolador determinar, com precisão, o instante exato em que o LED deve mudar de estado?" Essa abordagem automatiza o controle de tempo e nos permite obter uma precisão muito maior. Mas para isso, precisamos responder algumas perguntas-chave: (1) Como configurar um temporizador para gerar interrupções periódicas? (2) Como configurar NVIC e ativar sua linha de interrupção no NVIC? (3) Como programar a função ISR para alternar o estado do LED no momento exato? Vamos então criar um projeto onde o temporizador de propósito geral, TIM6, gera uma interrupção periódica e faz o LED verde piscar a uma frequência fixa de 1Hz, sem desperdício de processamento e com total precisão! Está pronto para dominar essa técnica essencial e tornar seus projetos ainda mais eficientes? Vamos começar?

1. Crie um projeto com o nome "Timer_CMSIS", com suporte ao CMSIS, como foi feito nos Roteiros 2, 3 e 4 (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* com uma grande gama de modos de operação 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, a frequência do sinal de *clock* geral está configurada em 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*. Antes da configuração, devemos gerar manualmente um evento de atualização para garantir que o TIM6 esteja corretamente inicializado e sincronizado no microcontrolador STM32H7. Para isso, ativamos o *bit* `TIM6_EGR_UG` e aguardamos a conclusão da operação.

```
TIM6->EGR |= TIM_EGR_UG_Msk;  //forca atualizacao imediata de TIM6
while (TIM6->EGR & TIM_EGR_UG); //aguarda a conclusao do evento de atualizacao
```

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, limpara a flag em TIM6_SR 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;
TIM6->SR = ~TIM_SR_UIF_Msk;
```

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

`C:/users/ea701/STM32Cube/Repository/STM32Cube_FW_H7_V1.11.2/Drivers/CMSIS/Include/core_cm7.h`.

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

A macro `TIM6_DAC_IRQn` está definida em `C:/users/ea701/STM32Cube/Repository/STM32Cube_FW_H7_V1.11.2/Drivers/CMSIS/Device/ST/STM32H7xx/Include/stm32h7a3xxq.h`.

Percebeu que não configuramos o controlador EXTI para rotear o sinal de interrupção como ocorreu com os sinais provenientes dos pinos de entrada dos módulos GPIO no Roteiro 3? Tem alguma ideia por onde os sinais circulam? Se não surgir nenhuma explicação plausível, não se apavore. Vamos entender isso mais adiante,

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 declarada a função de nome `TIM6_DAC1_IRQHandler` como ISR da interrupção do TIM6.

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

Antes da função *main*, insira a seguinte função:

```
void TIM6_DAC1_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 estado 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 (;;);”.

Faça ações combinadas de “Pause” e “Resume” para ver a variação do conteúdo de TIM6_CNT e descobrir experimentalmente 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 limpá-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? Essas variações nos intervalos de piscadas estão de acordo com suas expectativas?

Projeto com o *SysTick* usando CMSIS

Se você precisasse desenvolver um sistema operacional em tempo real (em inglês, *Real Time Operating System*), onde a cada 1ms fosse necessário verificar novos processos e alternar entre aqueles em execução, o que você faria? Observe que nesse cenário, precisão absoluta nas interrupções periódicas não é apenas desejável. É essencial! Mas há um detalhe ainda mais crítico: essas interrupções precisam estar perfeitamente sincronizadas com o sinal de relógio do processador para garantir que o sistema funcione de forma eficiente e previsível. Como garantir essa sincronização? Felizmente, microcontroladores com arquitetura ARM Cortex-M já possuem um temporizador dedicado para essa tarefa: o SysTick (do inglês *System Tick*). Esse temporizador de 24 *bits* tem a grande vantagem de operar exatamente na frequência do processador, garantindo a máxima precisão e sincronização.

Com essa ferramenta à disposição, a ferramenta oferece o controle necessário para garantir a precisão do agendamento e o desempenho do sistema. Agora, vamos explorar como configurar o SysTick para gerar interrupções periódicas a cada 1ms, síncronas com o sinal de relógio do processador. Aprenderemos a usar essas interrupções para implementar uma função *Delay*, que interrompe a execução do programa por um tempo determinado em milissegundos. Siga as instruções a seguir.

1. Crie o projeto "Delay_1ms", seguindo o mesmo procedimento utilizado no projeto anterior. 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. Você deve ter notado que não configuramos o controlador NVIC nem o EXTI, e que o bit de estado `SYST_CTRL_COUNTFLAG` do registrador `SysTick->CTRL` não foi limpo dentro da rotina de serviço `SysTick_Handler`. Você conseguiria explicar por quê neste caso o projeto funcionou corretamente? Sei que pode parecer confuso, mas não se preocupem! Vamos explorar isso juntos mais adiante e entender por quê, neste caso específico, não precisamos configurar o NVIC nem limpar o *bit* de estado do SysTick.

10. Faça o "Build" e transfira, no modo "Debug", o executável para o microcontrolador.

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 "SysTick_Handler**"**, semelhante ao que foi feito no projeto anterior. Valide essa abordagem implementando-a e verificando seu funcionamento.

Projeto com o LPTIM usando CMSIS

Imagine um sistema de controle de acesso por cartão. Ao inserir o cartão, o leitor gera um pulso único que aciona a abertura da catraca por um curto período, permitindo a passagem do usuário. Neste caso, o pulso único com uma duração exata desejada garante que a catraca seja aberta apenas uma vez por cartão, evitando que pessoas não autorizadas passem pela catraca. Já pensou como gerar um pulso único, controlando com precisão o momento exato de duas interrupções consecutivas em um temporizador de interrupções periódicas que vimos nos dois projetos anteriores? O desafio é criar um pulso com nível oposto ao do sinal original durante o restante do tempo. Parece complicado? Pois é! Mas com o o temporizador de baixo consumo (em inglês, *Low*

Power Timer - LPTIM) do microcontrolador STM32H7A3, essa tarefa se torna muito mais fácil! Vamos descobrir juntos como isso pode ser feito passo-a-passo?

1. Crie o projeto "OneShot_Bare_CMSIS", seguindo o mesmo procedimento utilizado nos projetos anteriores. Por padrão, o sinal de relógio do barramento APB4, onde os temporizadores LPTIM2 e LPTIM3 estão conectados, é de 64MHz após o *reset* com a seleção do HSI como fonte de clock.

2. Vamos definir a macro `pulse_length` e configurar o pino PB13 como saída para a forma de onda de pulso único gerada pelo LPTIM2, utilizando sua funcionalidade de saída de função especial/alternativa, em vez de um GPIO, no início da função "main":

```
#define pulse_length 1000
int main(void)
{
	// Ativar o clock do GPIOB
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN;
	// Configurar PB13 como saída alternativa (AF3 para LPTIM2_OUT)
	GPIOB->MODER &= ~(GPIO_MODER_MODE13_Msk);
	GPIOB->MODER |= GPIO_MODER_MODE13_1; // Alternate function mode
	GPIOB->AFR[1] &= ~(GPIO_AFRH_AFSEL13_Msk); // AF3 para PB13
	GPIOB->AFR[1] |= (3 << GPIO_AFRH_AFSEL13_Pos);
}
```

3. Ativamos o sinal de relógio que alimenta LPTIM2 com RCC_APB4ENR, adicionando a seguinte instrução:

```
	RCC->APB4ENR |= RCC_APB4ENR_LPTIM2EN_Msk;
```

4. Antes de modificar qualquer registrador do LPTIM2, o *reset* do periférico deve ser feito através do registrador RCC_APB4RSTR antes de ativá-lo, garantindo um estado inicial limpo. Adiciona-se as seguintes instruções:

```
	RCC->APB4RSTR |= RCC_APB4RSTR_LPTIM2RST_Msk;
	RCC->APB4RSTR &= ~RCC_APB4RSTR_LPTIM2RST_Msk;
```

5. Para configurar o registrador LPTIM_CFGR, é necessário que LPTIM2 esteja desabilitado em LPTIM_CR

```
	LPTIM2->CR &= ~LPTIM_CR_ENABLE_Msk;
```

6. Selecione, em seguida, o *clock* interno como a base de tempo de LPTIM2

```
	LPTIM2->CFGR &= ~LPTIM_CFGR_CKSEL_Msk;
	LPTIM2->CFGR &= ~LPTIM_CFGR_COUNTMODE_Msk; // CNT incrementa com clock interno
```

7. Configure o divisor de frequência do *clock* interno de 64MHz em 32.

```
	LPTIM2->CFGR &= ~LPTIM_CFGR_PRESC_Msk;
	LPTIM2->CFGR |= (0x5UL << LPTIM_CFGR_PRESC_Pos);
```

8. Configure a fonte de disparo do pulso único ser por *software* a seguir.

```
LPTIM2->CFGR &= ~LPTIM_CFGR_TRIGEN_Msk;
```

9. Configure a forma de onda ser do modo “pulso único” e o nível lógico deste pulso é “1” se o valor do contador LPTIM_CNT for maior que LPTIM_CMP, acrescentando

```
LPTIM2->CFGR &= ~LPTIM_CFGR_WAVPOL_Msk; // Use nivel logico da comparacao
LPTIM2->CFGR &= ~LPTIM_CFGR_WAVE_Msk;
```

10. Habilite LPTIM2 com LPTIM_CR para configurar o restante dos registradores com

```
LPTIM2->CR |= LPTIM_CR_ENABLE;
```

11. Configure o valor do registrador LPTIM_CMP para determinar o momento exato em que o pulso único terá início. Quando o contador LPTIM_CNT atingir este valor, inicia-se o pulso único.

```
LPTIM2->CMP = pulse_length / 2;
```

12. Configure o valor do registrador LPTIM_ARR para determinar o momento exato em que o pulso termina. Quando o contador chegar nesse valor, o valor do contador é resetado em zero e o valor do contador volta a ser menor do que o valor setado em LPTIM_CMP.

```
LPTIM2->ARR = pulse_length*2; // Período (tamanho do pulso)
```

13. Por fim, configure o modo de operação one-shot para LPTIM2.

```
LPTIM2->CR |= LPTIM_CR_SNGSTRT;
```

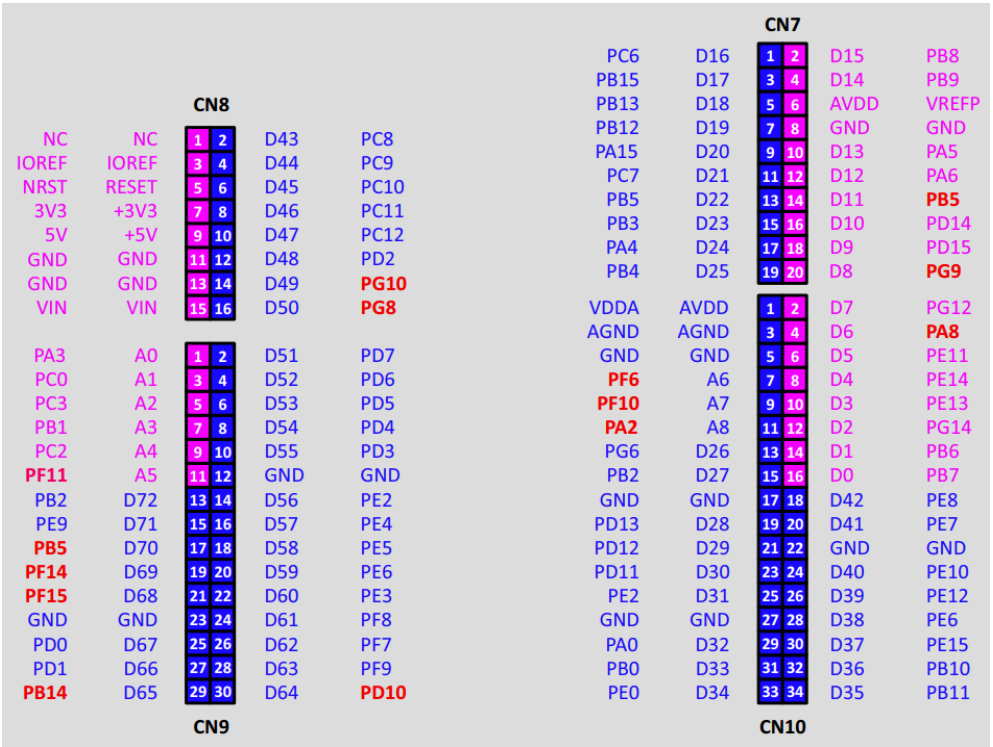
14. Segue-se o código completo do arquivo “main.c”:

```
#include <stdint.h>
#include <stm32h7a3xxq.h>
#define pulse_length 1000
int main(void)
{
   // Por padrao, frequencia do sinal de relogio eh 64MHz
   //Ativar o clock do GPIOB
   RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN;
   // Configurar PB13 como saída alternativa (AF3 para LPTIM2_OUT - Datasheet pg. 69)
   GPIOB->MODER &= ~(GPIO_MODER_MODE13_Msk);
   GPIOB->MODER |= GPIO_MODER_MODE13_1; // Alternate function mode
   GPIOB->AFR[1] &= ~(GPIO_AFRH_AFSEL13_Msk); // AF3 para PB13 (pg. 521)
   GPIOB->AFR[1] |= (3 << GPIO_AFRH_AFSEL13_Pos); // AF3 para PB13 (pg. 521)
   // Habilitar o clock do LPTIM2 (pg. 457)
   RCC->APB4ENR |= RCC_APB4ENR_LPTIM2EN_Msk;
   // Resetar LPTIM2 para configuração inicial (pg. 434)
   RCC->APB4RSTR |= RCC_APB4RSTR_LPTIM2RST_Msk;
   RCC->APB4RSTR &= ~RCC_APB4RSTR_LPTIM2RST_Msk;  // default depois de Reset
   // Garantir que o LPTIM2 está desativado antes de configurar LPTIM_CFGR, pg. 1781)
   LPTIM2->CR &= ~LPTIM_CR_ENABLE_Msk;
   // Habilitar o clock interno do LPTIM2
   LPTIM2->CFGR &= ~LPTIM_CFGR_CKSEL_Msk;
   // Selecionar o prescaler (divisor=32 - pg. 1766)
   LPTIM2->CFGR &= ~LPTIM_CFGR_PRESC_Msk;
```

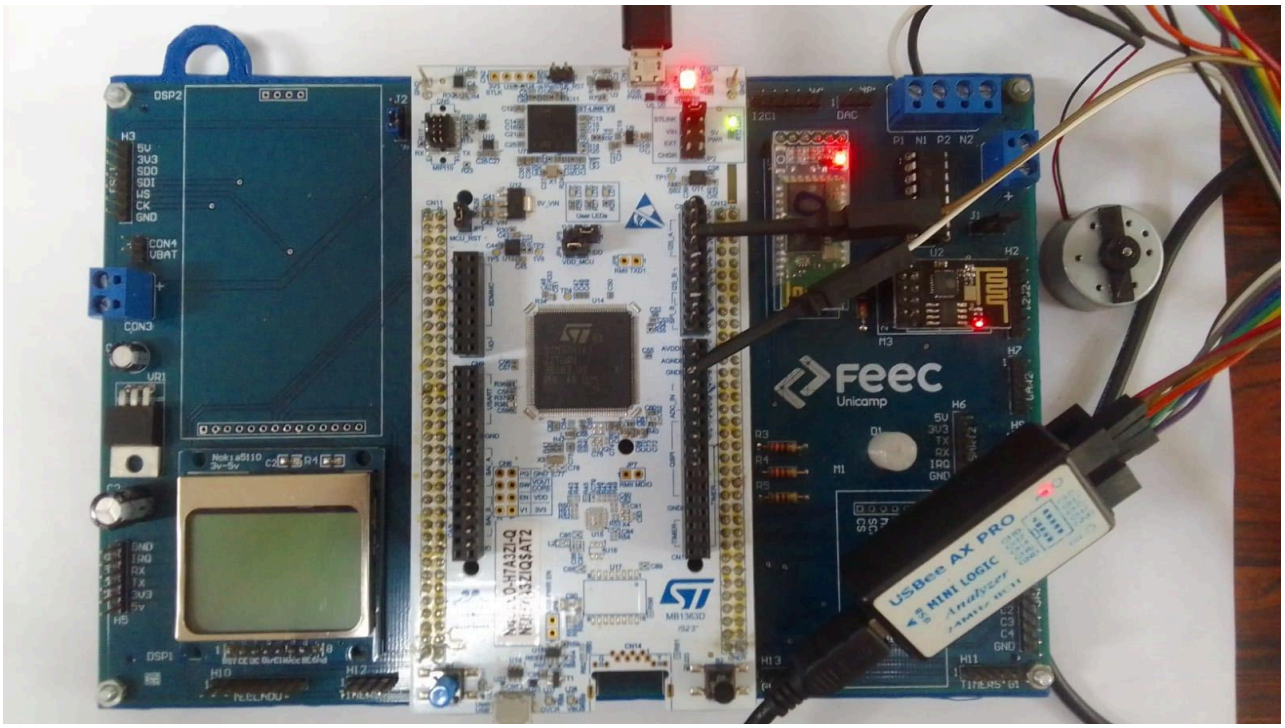
```
    LPTIM2->CFGR |= (0x5UL << LPTIM_CFGR_PRESC_Pos);
    // Configurar a fonte de trigger   (pg. 1766)
    // Manual: "The LPTIM counter is started as soon as one of the
    //CNTSTRT or the SNGSTRT bits is set by software."
    LPTIM2->CFGR &= ~LPTIM_CFGR_TRIGEN_Msk; // Trigger por software
    // Configurar a forma de onda de saída (pg. 1770)
    LPTIM2->CFGR &= ~LPTIM_CFGR_WAVPOL_Msk; // Use nivel logico da comparacao
    LPTIM2->CFGR &= ~LPTIM_CFGR_WAVE_Msk; // Desativa "Set-once"
    // Habilitar o LPTIM2
    LPTIM2->CR |= LPTIM_CR_ENABLE;
    // Definir período e comprimento do pulso (somente com LPTIM habilitado, pag 1783)
    LPTIM2->ARR = pulse_length*2; // Período (tamanho do pulso)
    LPTIM2->CMP = pulse_length / 2; // Inicio do pulso
    // Ativar o modo de contador: one-shot mode (pg. 1767)
    LPTIM2->CR |= LPTIM_CR_SNGSTRT;
    /* Loop forever */
    for(;;);
}
```

15. Faça o “Build” e transfira, no modo “Debug”, o código executável de extensão .elf para o microcontrolador.

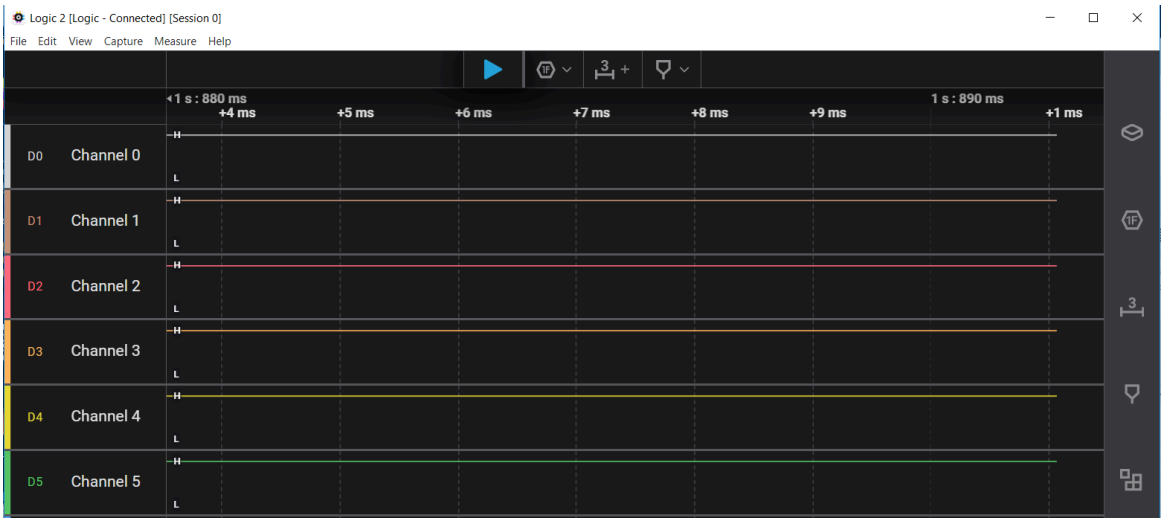
16. Conecte o Canal 0 do analisador lógico no pino PB13 da placa NUCLEO-144 e o seu terra num ponto GND da placa através dos conectores fêmeas Zio, CN7-5 e CN10-5, presentes na placa



A seguinte imagem ilustra uma forma de conexão.



17. Conecte o analisador a uma porta USB do *desktop* e abra o aplicativo Logic 2.4.x. Assim que o aplicativo reconhecer a conexão, uma janela será exibida, permitindo que você visualize o sinal nos pinos através dos canais correspondentes aos pinos conectados. Os detalhes de operação do analisador lógico podem ser verificados no vídeo. Utilize o maior valor de amostras por segundo que for possível e configure a aquisição para cerca de 10ms.



18. Coloque um *breakpoint* na seguinte linha de instrução e inicie a execução (“Resume”).

```
LPTIM2->CFGR &= ~LPTIM_CFGR_WAVE_Msk; // Desativa "Set-once"
```

19. Ao parar na linha de instrução, aumente a escala de tempo para dezenas de segundos e inicialize a captura do analisador clicando em cima do triângulo azul (“Start”) na barra de ferramenta superior. Retome (“Resume”) a execução do programa e monitore a forma de onda. Assim que aparecer um pulso, páre o analisador clicando em cima do quadrado azul (“Stop”).

20. Reduza a escala de tempo para microsegundos até que seja visível a largura do pulso único gerado. Meça a largura e verifique se é condizente com as configurações feitas nos registradores do temporizador LPTIM2.

21. Como podemos modificar o programa para gerar um pulso negativo (nível lógico baixo)? Não se preocupe se a resposta não estiver clara agora. Exploraremos as configurações em detalhes mais adiante.

22. Qual alteração no código transformará um pulso único em um degrau? Se a resposta ainda não surgiu, não se preocupe! Continue a leitura com atenção, pois a solução será explicada.

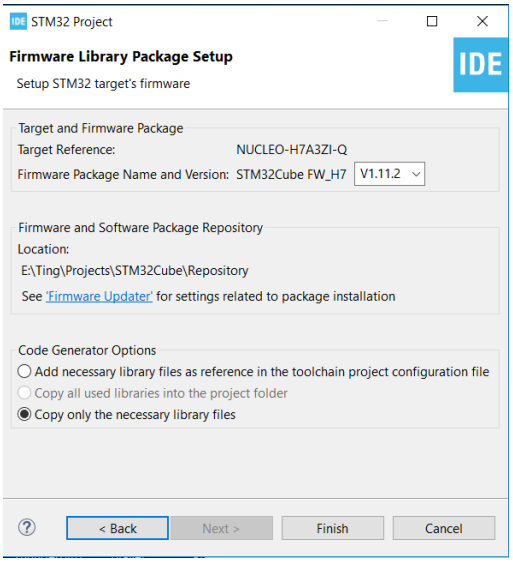
Projeto com o *Watchdog* usando STM32CubeMX e CMSIS

Imagine um carro com freios inteligentes: o microcontrolador recebe seus comandos e ajusta a pressão dos freios, garantindo uma frenagem precisa e segura. Mas e se o *software* desse microcontrolador travar? O resultado seria catastrófico: falha nos freios e um acidente! Em sistemas críticos como esse, detectar falhas não é suficiente. É preciso agir rápido, garantir que o sistema volte a funcionar antes que o problema se torne real. Desenvolver um sistema que detecta automaticamente falhas e se recupera sem intervenção humana parece coisa de ficção alguns anos atrás, certo? Mas essa tecnologia existe e está ao seu alcance! Uma estratégia usada em sistemas embarcados é o temporizador de vigilância (em inglês, *watchdog timer*), um verdadeiro cão de guarda digital. Esse temporizador monitora o sistema sem parar e, se algo der errado, ele age rápido: espera um pouco para ter certeza da falha e, se o problema persistir, reinicia o sistema. Com a tecnologia atual, muitos microcontroladores, como o STM32H7A3, já vêm com esse temporizador integrado, simplificando (e muito!) essa tarefa.

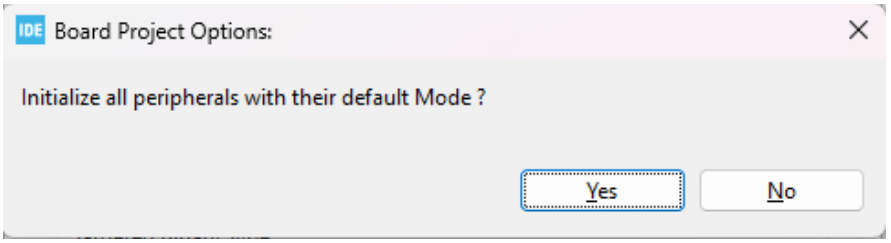
Vamos ilustrar a configuração de um *watchdog* do microcontrolador STM32H73A para criar um projeto onde você terá o controle da segurança do sistema! Considerando que o evento de "pressionamento do botão azul" seja "evidência da normalidade na operação", o *watchdog* será alimentado (em inglês, *feed*) pelo evento de pressionamento e o LED amarelo se acenderá para confirmar que o botão foi pressionado. Se o botão azul não for pressionado em intervalos menores que 5 segundos, o *watchdog* entenderá que houve uma falha e reiniciará o sistema. Após a reinicialização, o LED verde permanecerá apagado por um tempo, indicando que o sistema foi reiniciado. Em seguida, o sistema voltará ao seu estado normal de monitoramento.

Vamos aprender, passo-a-passo, como configurar o *watchdog* independente, IWDG, do STM32H7A3 para realizar essa tarefa? O IWDG é um temporizador independente, com seu próprio oscilador interno (LSI), de forma que ele opera mesmo se o *clock* principal do sistema falhar. No universo dos microcontroladores STM32, ajustar os sinais de relógio independentes para o módulo IWDG pode parecer um desafio e está fora do escopo desta disciplina. Mas não se preocupe! O STM32CubeIDE, com seu editor gráfico intuitivo STM32CubeMX, torna essa tarefa muito mais simples. Com ele, você pode visualizar e ajustar cada detalhe do sistema de sinais de relógio, desde a frequência do núcleo até a velocidade dos periféricos, tudo de forma interativa e acessível, e gerar códigos adequados.

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 STM32CubeMX seja incluído. Dê o nome de "IWDG" 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.

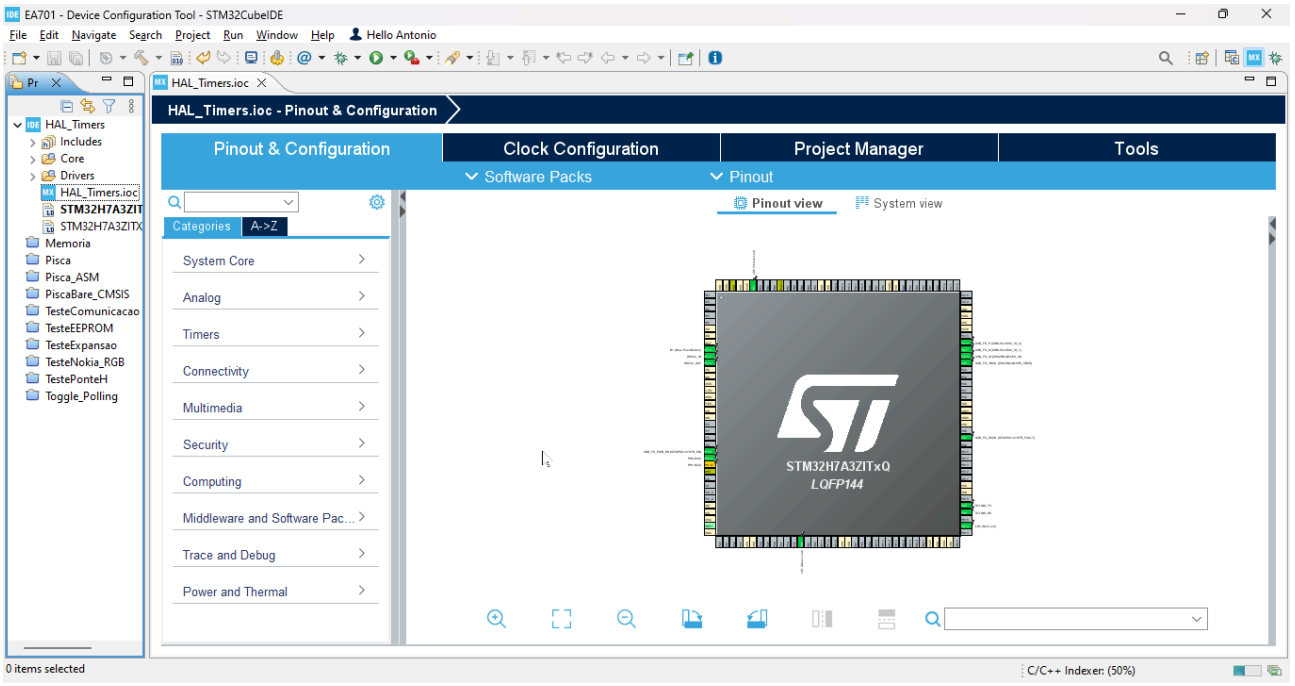


Como estamos trabalhando com uma placa que dispõe de alguns periféricos ligados ao microcontrolador, aparecerá uma janela perguntando se deseja iniciar os periféricos na configuração "default". Clique em "No" para que o código gerado pelo STM32CubeIDE não inclua a inicialização dos periféricos da placa com os pinos desabilitados.



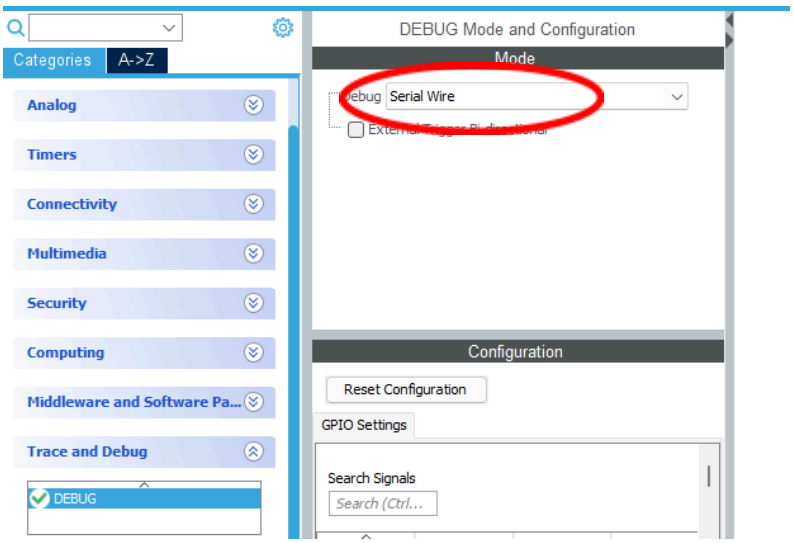
Após a instalação dos pacotes de suporte, o IDE entra na perspectiva de Inicialização, com o arquivo `IDWG.ioc` aberto no modo gráfico numa aba (este é o *plugin* STM32CubeMX do IDE). A

aba mostra graficamente numa janela todas as configurações de periféricos presentes em `IWDG.ioc` 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 intuitivos para aproximar e inspecionar cada detalhe, cada pino e cada conexão. 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).



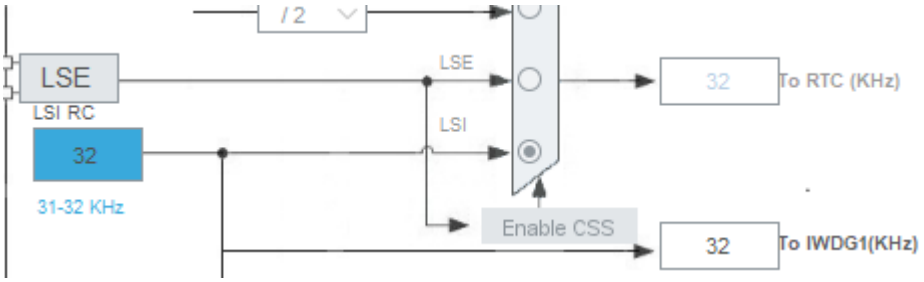
2. Dê um *zoom* na imagem do circuito integrado (clicar no ícone Maximize no canto direito superior) para que ele preencha a tela. Pode-se também clicar com o *mouse* sobre ele e arrastar para ajustar a posição. Vamos revisar os itens e configurar alguns deles. Clique sobre o item "*Trace and Debug*" 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*" na lista do item "*Debug*".

Ao selecionar "*Serial Wire*", o modo SWD de depuração, utilizado pelo depurador ST-LINK, é ativado. Observe como dois pinos (PA13 e PA14) na imagem "*Pinout view*" do microcontrolador agora aparecem em verde, com as funções selecionadas devidamente identificadas. Além disso, os dois pinos são automaticamente adicionados à lista de "*Configuration*". Esta etapa é essencial se usarmos o *software* de alocação de pinos, pois estes pinos estão fisicamente conectados ao ST-LINK presente na placa NUCLEO-144. Ao ativá-los manualmente, garantimos que o IDE reconheça sua função de depuração e evite que sejam alocados para outras funções inadvertidamente no STM32CubeMX.

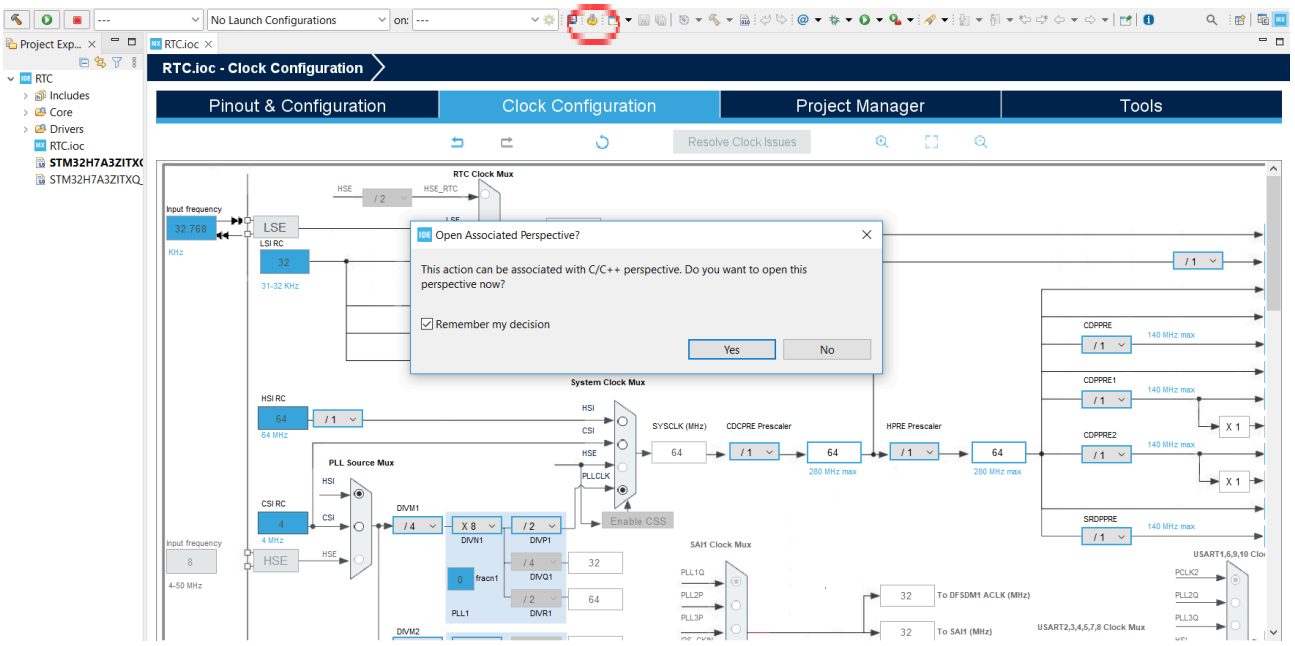


Vale ressaltar que, ao configurar os periféricos de um microcontrolador STM32 no STM32CubeMX, é fundamental observar a representação visual do circuito. Cada periférico ativado no *software* tem seus pinos correspondentes destacados em verde na tela, facilitando a configuração, a visualização e a depuração de projetos.

3. 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 kHz dentro. Note que este é o oscilador interno (independente) usado pelo *watchdog* independente (IWDG) do STM32H7A3.



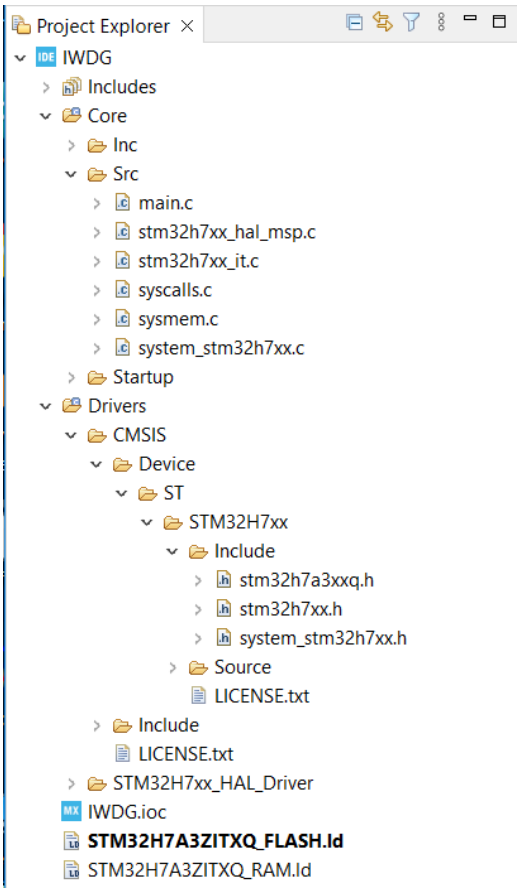
4. Salve o projeto e gere o código de inicialização em C clicando no ícone "*Device Configuration Tool Code Generation*". Se for pela primeira vez, aparecerá uma janela *popup* perguntando se deve lembrar a decisão de chaveamento para a Perspectiva C/C++. Selecione "Yes" para que se abra a Perspectiva C/C++ assim que concluir a geração de códigos.



5. Os códigos gerados pelo STM32CubeMX apresentam uma estrutura diferente dos projetos que implementamos até agora. As principais diferenças são:

- Estrutura de pastas
  - As pastas do desenvolvedor (`Inc` e `Src`) são movidas para dentro da pasta `Core`.
- Arquivos incluídos
  - Os arquivos-cabeçalho que incluíamos manualmente nos projetos anteriores agora estão integrados na pasta `Drivers/CMSIS/`.
  - O arquivo `stm32h7xx_it.c` é adicionado à pasta `Src` para armazenar as rotinas de tratamento de interrupções separadamente.
- Conteúdo do arquivo `main.c`
  - O arquivo `main.c` possui uma nova estrutura, dividida em segmentos pelos comentários `/* USER CODE BEGIN XXX */` e `/* USER CODE END XXX */`, onde `XXX` especifica o tipo de dado a ser inserido.
  - Os dados inseridos entre esses comentários são preservados quando os códigos são regenerados após alterações na configuração.

Aprenderemos como utilizar cada segmento presente no `main.c` sob demanda.



Para entender como os módulos realmente funcionam, vamos criar nossas próprias funções em vez de usarmos as funções de inicialização dos módulos geradas automaticamente pelo STM32CubeMX no arquivo `main.c`. Essa prática nos permitirá aprofundar nosso conhecimento sobre o funcionamento desses módulos e ter uma compreensão muito mais clara de como o código é executado. Porisso, antes de adicionarmos os nossos códigos, vamos remover algumas linhas de instruções geradas. Em vez de simplesmente remover as linhas de instrução do código gerado, elas foram apenas comentadas (selecione a região a ser removida e aperte simultaneamente as teclas 'CTRL' e '/'). Essa abordagem permite que visualizemos claramente o que foi descartado, facilitando a compreensão das alterações feitas e fornecendo um histórico do código original. Além disso, caso seja necessário reverter alguma alteração, basta descomentar as linhas desejadas, tornando o processo mais rápido e eficiente.

(a) protótipos das funções

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
//static void MX_GPIO_Init(void);
```

(b) chamadas dad funções

```
/* Initialize all configured peripherals */
//  MX_GPIO_Init();
```

(c) definição das funções

```
///**
//  * @brief GPIO Initialization Function
//  * @param None
//  * @retval None
```

```
//  */
//static void MX_GPIO_Init(void)
//{
//  GPIO_InitTypeDef GPIO_InitStruct = {0};
///* USER CODE BEGIN MX_GPIO_Init_1 */
///* USER CODE END MX_GPIO_Init_1 */
//
//  /* GPIO Ports Clock Enable */
//  __HAL_RCC_GPIOC_CLK_ENABLE();
//  __HAL_RCC_GPIOF_CLK_ENABLE();
//  __HAL_RCC_GPIOH_CLK_ENABLE();
//  __HAL_RCC_GPIOB_CLK_ENABLE();
//  __HAL_RCC_GPIOD_CLK_ENABLE();
//  __HAL_RCC_GPIOG_CLK_ENABLE();
//  __HAL_RCC_GPIOA_CLK_ENABLE();
//  __HAL_RCC_GPIOE_CLK_ENABLE();
//
//  /*Configure GPIO pin Output Level */
//  HAL_GPIO_WritePin(USB_FS_PWR_EN_GPIO_Port, USB_FS_PWR_EN_Pin, GPIO_PIN_RESET);
//
//  /*Configure GPIO pin Output Level */
//  HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin, GPIO_PIN_RESET);
//
//  /*Configure GPIO pin Output Level */
//  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
//
//  /*Configure GPIO pin : B1_Pin */
//  GPIO_InitStruct.Pin = B1_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
//
//  /*Configure GPIO pin : USB_FS_PWR_EN_Pin */
//  GPIO_InitStruct.Pin = USB_FS_PWR_EN_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  HAL_GPIO_Init(USB_FS_PWR_EN_GPIO_Port, &GPIO_InitStruct);
//
//  /*Configure GPIO pins : LD1_Pin LD3_Pin */
//  GPIO_InitStruct.Pin = LD1_Pin|LD3_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
//
//  /*Configure GPIO pins : STLINK_RX_Pin STLINK_TX_Pin */
//  GPIO_InitStruct.Pin = STLINK_RX_Pin|STLINK_TX_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  GPIO_InitStruct.Alternate = GPIO_AF7_USART3;
//  HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
//
//  /*Configure GPIO pin : USB_FS_OVCR_Pin */
//  GPIO_InitStruct.Pin = USB_FS_OVCR_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  HAL_GPIO_Init(USB_FS_OVCR_GPIO_Port, &GPIO_InitStruct);
//
//  /*Configure GPIO pin : USB_FS_ID_Pin */
//  GPIO_InitStruct.Pin = USB_FS_ID_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  GPIO_InitStruct.Alternate = GPIO_AF10_OTG1_HS;
//  HAL_GPIO_Init(USB_FS_ID_GPIO_Port, &GPIO_InitStruct);
```

```
  //
  //  /*Configure GPIO pins : USB_FS_N_Pin USB_FS_P_Pin */
  //  GPIO_InitStruct.Pin = USB_FS_N_Pin|USB_FS_P_Pin;
  //  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
  //  GPIO_InitStruct.Pull = GPIO_NOPULL;
  //  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  //  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
  //
  //  /*Configure GPIO pin : LD2_Pin */
  //  GPIO_InitStruct.Pin = LD2_Pin;
  //  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  //  GPIO_InitStruct.Pull = GPIO_NOPULL;
  //  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  //  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
  //
  ///* USER CODE BEGIN MX_GPIO_Init_2 */
  ///* USER CODE END MX_GPIO_Init_2 */
  //}
```

6. Vamos incluir os protótipos das funções que definiremos depois da linha `/* USER CODE BEGIN PFP */`

```
void delay(void);
void GPIO_Config(void);
void IWDG_Config(void);
```

7. Vamos agora adicionar a definição da função `IWDG_Config` segundo o procedimento recomendado no Manual de Referência logo abaixo da linha `/* USER CODE BEGIN 4 */`

```
void IWDG_Config(void) {
   // Habilita o IWDG e configura timeout de 5s
   IWDG1->KR = 0xCCCC;  // Habilita o IWDG
   IWDG1->KR = 0x5555;  // Desbloqueia acesso aos registradores
   IWDG1->PR = 0b100;   // Prescaler  64 (LSI típico ~32kHz)
   IWDG1->RLR = 2500;   // Timeout de ~5s: (64 * 25000) / 32000 = ~5s
   while (IWDG1->SR & (IWDG_SR_PVU_Msk || IWDG_SR_RVU_Msk ||
             IWDG_SR_WVU_Msk)); // aguardar a conclusao
   IWDG1->KR = 0xAAAA;  // Alimenta o watchdog
}
```

Habilita-se o módulo escrevendo 0xCCCC no registrador de chaves IWDG_KR. Em seguida, escreve-se 0x5555 para desbloquear acesso aos registradores de IWDG. Com um sinal de relógio na frequência de 32kHz, configuramos o divisor de frequência em 64 pelo registrador IWDG_PR e o valor de recarga em 2500 pelo registrador IWDG_RLR. Assim, teremos os 32kHz divididos por 64, resultando em 500Hz a frequência do *clock* do contador de IWDG, e o período de recargas em 25000/500=5s. Aguarda-se até que sejam concluídas as configurações monitorando os estados no registrador IWDG_SR. Para iniciar o monitoramento, realizamos a primeira recarga do contador escrevendo o valor hexadecimal 0xAAAA no registrador IWDG_KR. Essa ação "alimenta" o *watchdog*, iniciando a contagem regressiva.

8. Em seguida, incluímos a definição da função `GPIO_Config` para habilitar os pinos que controlam os LEDs amarelo e verde e o botão azul.

```
void GPIO_Config(void) {
   // Habilita clock dos GPIOs B e E
   RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN |
```

```
                RCC_AHB4ENR_GPIOCEN |
                      RCC_AHB4ENR_GPIOEEN;
    // Configura PB0 (LED verde) como saída
        GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk);
        GPIOB->MODER |= GPIO_MODER_MODE0_0;
        GPIOB->OTYPER &= ~GPIO_OTYPER_OT0_Msk; // PB0 como push-pull
    // Configura PE1 (LED amarelo) como saída
        GPIOE->MODER &= ~(GPIO_MODER_MODE1_Msk);
        GPIOE->MODER |= GPIO_MODER_MODE1_0;
        GPIOE->OTYPER &= ~GPIO_OTYPER_OT1_Msk; // PB0 como push-pull
    // Configura PB13 (botão azul) como entrada (pull-down)
    GPIOC->MODER &= ~(GPIO_MODER_MODE13_Msk);  //Entrada Digital
}
```

9. Adicionamos também a definição da função `delay` cujo tempo de atraso foi obtido empiricamente.

```
void delay() {
    uint32_t count = 50000000;
    while (count--);
}
```

10. Vamos atualizar a função `main` para que realize a tarefa. Em primeiro lugar, insira as chamadas das seguintes funções de inicialização dos módulos GPIO e IWDG

```
/* Initialize all configured peripherals */
//  MX_GPIO_Init();
GPIO_Config();
IWDG_Config();
```

Depois da linha `/* USER CODE BEGIN 2 */`, adicione as seguintes instruções para que após um *reset*, o programa mantém o LED verde apagado por um tempo antes de acendê-lo

```
 delay();
 GPIOB->BSRR = (1U << 0); // Acende LED verde
```

No laço de espera `while`, insira o seguinte bloco de instruções depois da linha `/* USER CODE BEGIN 3 */`

```
// Se botão azul for pressionado
if (GPIOC->IDR & (1U << 13)) {
     GPIOE->BSRR = (1U << 1); // Acende LED amarelo
     IWDG1->KR = 0xAAAA;  // Alimenta o watchdog
} else {
     GPIOE->BSRR = (1U << (1 + 16)); // Apaga LED amarelo
}
```

O código "alimenta" o *watchdog* quando o botão azul é acionado, escrevendo 0xAAAA no registrador IWDG_KR. Ao mesmo tempo, o LED amarelo é aceso. Se o botão azul não for acionado, o LED amarelo será apagado. Se o botão não for apertado por um intervalo mario que 5s, o *watchdog* irá reiniciar o programa, apagando o LED e esperando um intervalo de tempo perceptível para acendê-lo.

11. Realize o "Build" e transfira o código executável no modo "Debug" para o microcontrolador. Execute ("Resume") o programa. Após o LED verde acender, aperte o botão em intervalos menores

que 2s. Após 5s sem pressionar o botão, o LED se apagará e o programa será reiniciado. O LED verde permanecerá apagado até ser explicitamente aceso por uma instrução do programa, evidenciando que o sistema foi reiniciado.

12. Uma característica importante do módulo IWDG é o registrador de chave IWDG_KR. Ele exige que uma chave específica seja escrita para que os demais registradores do IWDG possam ser modificados. Essa medida de segurança aumenta um pouco a complexidade, mas traz benefícios importantes. Você consegue imaginar quais? Se não souber, não se preocupe, discutiremos a importância desse registrador mais adiante.

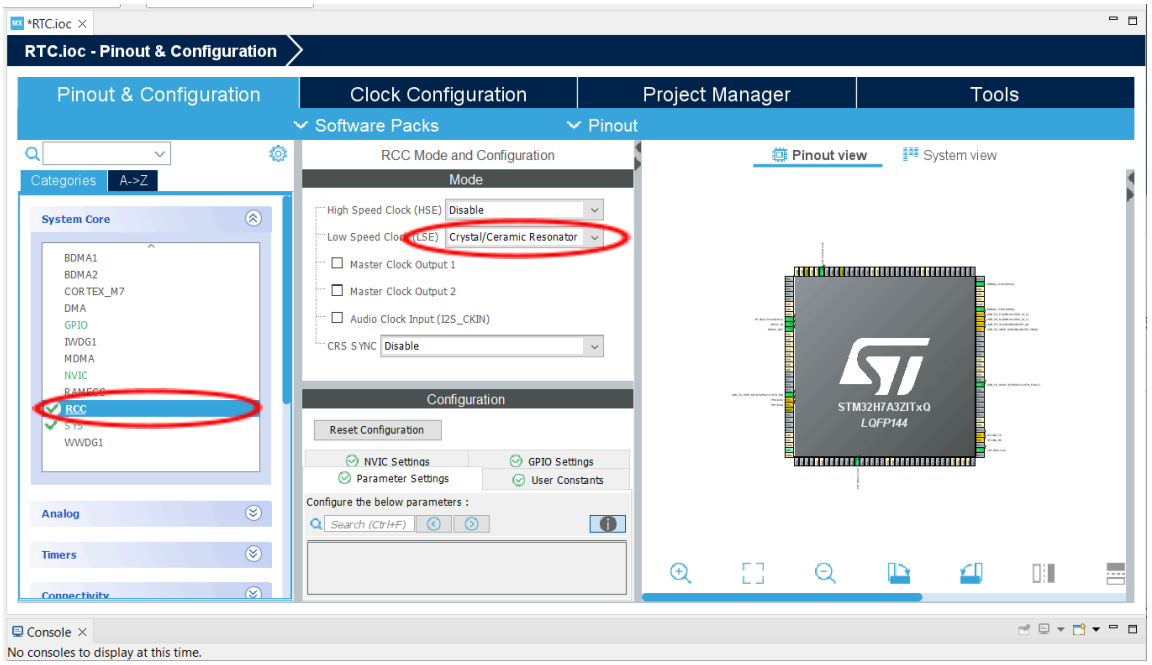
Projeto com RTC usando STM32CubeMX e CMSIS

Agora vamos explorar o uso do RTC (do inglês *Real-Time Clock*) para controlar com precisão e confiabilidade a alternância do estado do LED amarelo da placa NUCLEO-144 a 1Hz. Além disso, configuramos o microcontrolador para que o LED verde seja acionado e alterne o seu estado em um horário específico todos os dias. Mas por que usar o RTC em vez dos temporizadores convencionais que já estudamos? A resposta está nas vantagens que ele oferece: maior precisão, compensação térmica, eficiência energética e facilidade na implementação de funções de relógio. Diferente dos temporizadores comuns, o RTC também conta com mecanismos de proteção contra alterações indevidas como temporizador *watchdog*, garantindo a integridade do sistema ao evitar modificações acidentais ou maliciosas. Outro grande diferencial do RTC do STM32H7A3 é sua fonte de clock independente como *watchdog*, permitindo que ele continue funcionando mesmo que o processador e outros periféricos parem. Isso o torna essencial para aplicações críticas, como sistemas de alarme confiáveis, que continuam operando sem falhas, relógios digitais de alta precisão, que mantêm a marcação do tempo sem interrupções, e dispositivos de coleta de dados, que garantem medições regulares mesmo durante quedas de energia.

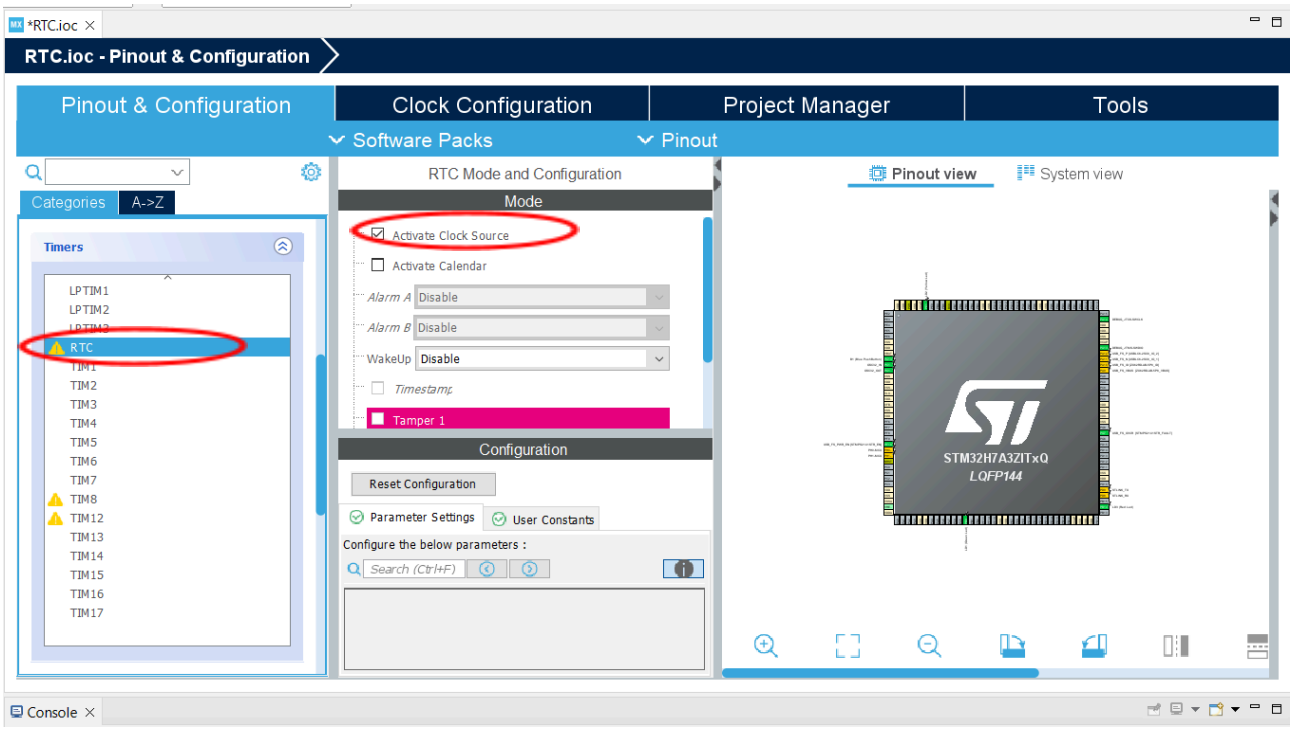
Que tal verificarmos essas vantagens e a complexidade de programação adicional, implementando juntos o temporizador de **1 segundo** de alta precisão e um alarme que soa diariamente no horário 23:46:12, seguindo um passo-a-passo detalhado?

1. Crie o projeto “RTC” da mesma forma que foi feito com o projeto “IWDG”, selecionando “STM32Cube” na opção “*Targeted Project Type*” para incluir o *plugin* STM32CubeMX. Selecione ainda “Serial Wire” como o modo “DEBUG” no editor gráfico do arquivo `RTC.ioc`.

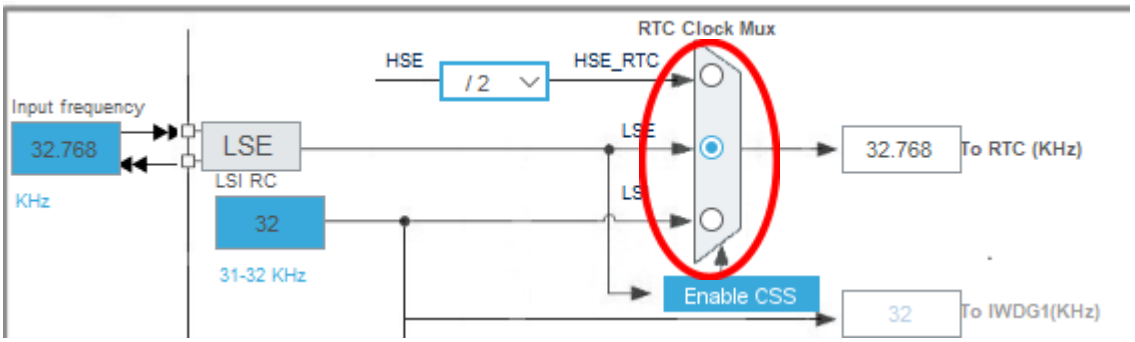
2. No mesmo edito, expanda o “*System Core*” no menu esquerdo, abra a configuração de “RCC” e selecione “Crystal/Ceramic Resonance” como a fonte do sinal de relógio LSE (do inglês, “*Low Speed Clock*”).



3. Expanda "*Timers*" no menu esquerdo, abra a configuração de "RTC" e ative a fonte de sinais de relógio para liberar a edição da fonte de sinais de relógio do RTC.



4. Clique na aba "*Clock Configuration*" para ver a "árvore" de distribuição de sinais de relógio. 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 como ilustra a seguinte figura.



5. Para entender como os módulos GPIO e RTC realmente funcionam, vamos criar nossas próprias funções em vez de usarmos as funções de inicialização geradas automaticamente pelo STM32CubeMX no arquivo `main.c`. Comentamos as seguintes linhas de instruções.

(a) a variável global usada para processar dados referentes a RTC

```
/* Private variables ---------------------------------------------------------*/
//RTC_HandleTypeDef hrtc;
```

(b) os protótipos dessas funções

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
//static void MX_GPIO_Init(void);
//static void MX_RTC_Init(void);
```

(c) as chamadas dessas funções

```
 /* Initialize all configured peripherals */
//  MX_GPIO_Init();
//  MX_RTC_Init();
```

(d) as definições dessas funções

```
///**
//  * @brief RTC Initialization Function
//  * @param None
//  * @retval None
//  */
//static void MX_RTC_Init(void)
//{
//
//  /* USER CODE BEGIN RTC_Init 0 */
//
//  /* USER CODE END RTC_Init 0 */
//
//  /* USER CODE BEGIN RTC_Init 1 */
//
//  /* USER CODE END RTC_Init 1 */
//
//  /** Initialize RTC Only
//  */
//  hrtc.Instance = RTC;
//  hrtc.Init.HourFormat = RTC_HOURFORMAT_24;
//  hrtc.Init.AsynchPrediv = 127;
//  hrtc.Init.SynchPrediv = 255;
//  hrtc.Init.OutPut = RTC_OUTPUT_DISABLE;
//  hrtc.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
//  hrtc.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;
//  hrtc.Init.OutPutRemap = RTC_OUTPUT_REMAP_NONE;
//  if (HAL_RTC_Init(&hrtc) != HAL_OK)
//  {
//    Error_Handler();
```

```
//  }
//  /* USER CODE BEGIN RTC_Init 2 */
//
//  /* USER CODE END RTC_Init 2 */
//
//}
///**
//  * @brief GPIO Initialization Function
//  * @param None
//  * @retval None
//  */
//static void MX_GPIO_Init(void)
//{
//  GPIO_InitTypeDef GPIO_InitStruct = {0};
///* USER CODE BEGIN MX_GPIO_Init_1 */
///* USER CODE END MX_GPIO_Init_1 */
//
//  /* GPIO Ports Clock Enable */
//  __HAL_RCC_GPIOC_CLK_ENABLE();
//  __HAL_RCC_GPIOF_CLK_ENABLE();
//  __HAL_RCC_GPIOH_CLK_ENABLE();
//  __HAL_RCC_GPIOB_CLK_ENABLE();
//  __HAL_RCC_GPIOD_CLK_ENABLE();
//  __HAL_RCC_GPIOG_CLK_ENABLE();
//  __HAL_RCC_GPIOA_CLK_ENABLE();
//  __HAL_RCC_GPIOE_CLK_ENABLE();
//
//  /*Configure GPIO pin Output Level */
//  HAL_GPIO_WritePin(USB_FS_PWR_EN_GPIO_Port, USB_FS_PWR_EN_Pin, GPIO_PIN_RESET);
//
//  /*Configure GPIO pin Output Level */
//  HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin, GPIO_PIN_RESET);
//
//  /*Configure GPIO pin Output Level */
//  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
//
//  /*Configure GPIO pin : B1_Pin */
//  GPIO_InitStruct.Pin = B1_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
//
//  /*Configure GPIO pin : USB_FS_PWR_EN_Pin */
//  GPIO_InitStruct.Pin = USB_FS_PWR_EN_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  HAL_GPIO_Init(USB_FS_PWR_EN_GPIO_Port, &GPIO_InitStruct);
//
//  /*Configure GPIO pins : LD1_Pin LD3_Pin */
//  GPIO_InitStruct.Pin = LD1_Pin|LD3_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
//
//  /*Configure GPIO pins : STLINK_RX_Pin STLINK_TX_Pin */
//  GPIO_InitStruct.Pin = STLINK_RX_Pin|STLINK_TX_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  GPIO_InitStruct.Alternate = GPIO_AF7_USART3;
//  HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
//
//  /*Configure GPIO pin : USB_FS_OVCR_Pin */
//  GPIO_InitStruct.Pin = USB_FS_OVCR_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
```

```
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  HAL_GPIO_Init(USB_FS_OVCR_GPIO_Port, &GPIO_InitStruct);
//
//  /*Configure GPIO pin : USB_FS_ID_Pin */
//  GPIO_InitStruct.Pin = USB_FS_ID_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  GPIO_InitStruct.Alternate = GPIO_AF10_OTG1_HS;
//  HAL_GPIO_Init(USB_FS_ID_GPIO_Port, &GPIO_InitStruct);
//
//  /*Configure GPIO pins : USB_FS_N_Pin USB_FS_P_Pin */
//  GPIO_InitStruct.Pin = USB_FS_N_Pin|USB_FS_P_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
//
//  /*Configure GPIO pin : LD2_Pin */
//  GPIO_InitStruct.Pin = LD2_Pin;
//  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
//  GPIO_InitStruct.Pull = GPIO_NOPULL;
//  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
//  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
//
///* USER CODE BEGIN MX_GPIO_Init_2 */
///* USER CODE END MX_GPIO_Init_2 */
//}
```

6. Vamos adicionar agora as nossas funções de configuração. Primeiro, os protótipos das nossas funções de inicialização dos pinos GPIO que alimentam os LEDs verde e amarelo e do RTC depois da linha `/* USER CODE BEGIN PFP */`

```
/* USER CODE BEGIN PFP */
void LD1_LD2_PInit(void);
void RTC_PInit(void);
/* USER CODE END PFP */
```

Em seguida, as suas chamadas depois da linha `/* USER CODE BEGIN 2 */`

```
 /* USER CODE BEGIN 2 */
 LD1_LD2_PInit();
 RTC_PInit();
 /* USER CODE END 2 */
```

7. Vamos iniciar com a definição da função `RTC_PInit` começando com a habilitação do sinal de relógio do RTC pelo registrador RCC_APB4ENR.

```
void RTC_PInit(void) {
      // Habilitar o sinal de ativacao de RTC (por padrao, eh ativado no reset)
      RCC->APB4ENR |= RCC_APB4ENR_RTCAPBEN_Msk;
}
```

9. Após um *reset* geral do sistema, os registradores do RTC são protegidos contra escritas acidentais. Essa proteção é controlada pelo *bit* DBP (do inglês *Disable Backup Power*) do registrador PWR_CR1 presente em um módulo de controle de energia (PWR). É preciso setar esse *bit* e aguardar a habilitação da escrita nos registradores do RTC com as seguintes instruções.

```
PWR->CR1 |= PWR_CR1_DBP;
while (!(PWR->CR1 & PWR_CR1_DBP)); // Aguarda a habilitação
```

10. Apenas após essa habilitação, podemos selecionar LSE como a fonte do sinal de relógio do RTC com o registrador BCC_BDCR. O código de ativação de LSE já foi gerado pelo STM32CubeMX. Devemos, no entanto, garantir uma configuração limpa do RTC antes da seleção, resetando todo o domínio de *backup* do RTC (registradores do RTC, os registradores de *backup* e a configuração da fonte de *clock* do RTC) através do *bit* VSRST do registrador RCC_BDCR e limpando a fonte de relógio antiga com o *reset* dos *bits* `RCC_BDCR_RTCSEL`

```
RCC->BDCR |= RCC_BDCR_VSWRST;
RCC->BDCR &= ~RCC_BDCR_RTCSEL;
```

Aí saimos do modo *reset* e setamos, por fim, LSE como a nova fonte de relógio

```
RCC->BDCR &= ~RCC_BDCR_VSWRST;
RCC->BDCR |= (RCC_BDCR_RTCEN_Msk |
              RCC_BDCR_RTCSEL_1);
```

11. Com o RTC habilitado, podemos configurar os registradores do RTC. Porém, por segurança, esses registradores são protegidos dos acessos de escrita. Apenas com a escrita da chave de desbloqueio no registrador RTC_WPR é liberado o acesso de escrita a eles:

```
RTC->WPR = 0xCAU;  // Desbloquear a protecao de escrita
RTC->WPR = 0x53U;
```

12. Com os registradores do RTC desprotegidos, podemos seguir o procedimento descrito no Manual de Referência para configurar o horário e o dia no RTC.

```
RTC->ICSR |= RTC_ICSR_INIT; // Configura o RTC para o modo de inicialização
while (!(RTC->ICSR & RTC_ICSR_INITF)); // Aguarda a inicialização do RTC
RTC->CR &= ~RTC_CR_FMT; // Configura o formato de hora para 24 horas
// Configura o pre-scaler do RTC
RTC->PRER = (127 << RTC_PRER_PREDIV_A_Pos) | (255 << RTC_PRER_PREDIV_S_Pos);
// Configurar a hora
RTC->TR = ((2 << RTC_TR_HT_Pos) | (3 << RTC_TR_HU_Pos) |   // Horas (23)
       (4 << RTC_TR_MNT_Pos) | (5 << RTC_TR_MNU_Pos) |   // Minutos (45)
       (1 << RTC_TR_ST_Pos) | (2 << RTC_TR_SU_Pos));     // Segundos (12)
// Configurar a data
RTC->DR = ((2 << RTC_DR_YT_Pos) | (5 << RTC_DR_YU_Pos) |   // Ano (2025 - 2000 = 25)
       (0 << RTC_DR_MT_Pos) | (2 << RTC_DR_MU_Pos) |    // Mês (Fevereiro = 2)
       (2 << RTC_DR_DT_Pos) | (4 << RTC_DR_DU_Pos));    // Dia (24)
RTC->ICSR &= ~RTC_ICSR_INIT; // Sair do modo de inicialização
```

Deve-se entrar no modo de inicialização usando o registrador RTC_ICSR, configurar o formato do horário através de RTC_CR, setar os divisores de frequência em RTC_PRER, e inicializar o RTC com o horário e data desejados através dos registradores RTC_TR e RTC_DR. Daí sair do modo de inicialização. Segundo a Nota de Aplicação, o fator de divisão do *prescaler* assíncrono é definido como 128, e o fator de divisão síncrono como 256, para obter uma frequência de *clock* interna de 1 Hz a partir de uma fonte LSE de frequência 32,768 kHz.

13. Após a inicialização do sistema, é possível configurar alarmes de tempo real para gerar eventos de interrupção que serão acionados em momentos específicos ou após um período determinado.

```
// Configura o alarme A para gerar eventos periódicos de 1s
RTC->CR &= ~RTC_CR_ALRAE; // Desabilita o Alarme A
RTC->CR &= ~RTC_CR_ALRAIE; // Desativa a interrupção do alarme A
// Limpar a flag de interrupcao do alarme A
RTC->SCR |= RTC_SCR_CALRAF_Msk;
while (!(RTC->ICSR & RTC_ICSR_ALRAWF)); // Aguarda prontidao para configuracao
RTC->ALRMAR &= ~(RTC_ALRMAR_PM);   //fomato 24 horas
// Programa o evento alarme para cada segundo
RTC->ALRMAR |= (RTC_ALRMAR_MSK4 | RTC_ALRMAR_MSK3 | RTC_ALRMAR_MSK2 | RTC_ALRMAR_MSK1);
// Desconsiderar data, hora, minuto, segundo
// Habilita o Alarme A
RTC->CR |= RTC_CR_ALRAE;
// Habilita a interrupcao do alarme
RTC->CR |= RTC_CR_ALRAIE;

// Configura o alarme B para gerar um unico evento 1 minuto após o reset
RTC->CR &= ~RTC_CR_ALRBE;  // Desabilita o alarme B
RTC->CR &= ~RTC_CR_ALRBIE; // Desativa a interrupcao do alarme B
// Limpar flag de interrupcao do alarme B
RTC->SCR |= RTC_SCR_CALRBF_Msk;
while (!(RTC->ICSR & RTC_ICSR_ALRBWF)); // Aguarda prontidao para configuracao
RTC->ALRMBR &= ~(RTC_ALRMBR_PM);  // Formato 24h
// Define o evento de alarme para ocorrer exatamente 1 minuto depois
RTC->ALRMBR = (RTC_ALRMAR_MSK4 |
              (2 << RTC_ALRMBR_HT_Pos) | (3 << RTC_ALRMBR_HU_Pos) |   // Horas (23)
              (4 << RTC_ALRMBR_MNT_Pos) | ((5 + 1) << RTC_ALRMBR_MNU_Pos) | // Minutos
              (1 << RTC_ALRMBR_ST_Pos) | (2 << RTC_ALRMBR_SU_Pos) );   // Segundos (12)
// Habilita o Alarme B
RTC->CR |= RTC_CR_ALRBE;
// Habilita interrupcao do Alarme B
RTC->CR |= RTC_CR_ALRBIE;
```

Seguimos o procedimento recomendado no Manual de Referência. Em primeiro lugar, desabilitamos o alarme e sua interrupção com RTC_CR, limpamos a *flag* de interrupção correspondente com RTC_SCR e aguardamos que o registrador RTC_ICSR indique que o alarme esteja pronto para programação. Em seguida, programamos pelos registradores RTC_ALRMAR ou RTC_ALRMBR os eventos de interrupção. E concluimos com a reabilitação do alarme e sua interrupção. Neste projeto, configuramos o alarme A para gerar eventos de interrupção periódica de 1s e o alarme B para gerar eventos de interrupção periódica diária às 23:46:12. Detalhes dessas configurações são encontradas na Nota de Aplicação do RTC.

14. Reative a proteção contra escrita, garantindo que os registradores do RTC não sejam alterados inadvertidamente após a configuração.

```
RTC->WPR = 0xFFU;   //Bloquear acesso de escrita
```

15. Habilite a interrupção do RTC Alarm no NVIC, permitindo que o microcontrolador atenda ao evento quando ele ocorrer. A posição do vetor de interrupção do RTC_Alarm_IRQn no STM32 é 41.

```
NVIC_SetPriority(RTC_Alarm_IRQn, 1); // Configura a prioridade da interrupção
NVIC_EnableIRQ(RTC_Alarm_IRQn); // Habilita a interrupção no NVIC
```

16. Habilite a interrupção do evento de entrada 17 do EXTI e configure a sensibilidade para a borda de subida

```
EXTI->IMR1 |= EXTI_IMR1_IM17_Msk;   // habilita a interrupcao do evento de entrada 17
EXTI->RTSR1 |= EXTI_RTSR1_TR17_Msk; // captura na borda de subida
```

O RTC usa a linha 17 do EXTI, como mostra a tabela de mapeamento de eventos internos nas linhas de entrada de EXTI, para sinalizar eventos de alarme, ou seja o evento interno `RTC Alarm` é roteado para a linha 17 do controlador de interrupções externas EXTI que é tratada pelo NVIC chamando a rotina de interrupção correspondente `RTC_Alarm_IRQHandler`. Portanto, é necessário habilitar a linha 17 do controlador EXTI pelo registrador EXTI_IMR1 e configurar a sensibilidade da linha pelo registrador EXTI_RTSR1.

17. Vamos programar a rotina de serviço `RTC_Alarm_IRQHandler`. Abra o arquivo `stm32h7xx_it.c` e insira os protótipos de duas funções após a linha `/* USER CODE BEGIN PFP */`

```
uint8_t RTC_IT_segundo ();
void RTC_reseta_IT_segundo ();
```

`Adicione os` seguintes códigos depois da linha /* USER CODE BEGIN 1 */

```
static uint8_t estado=0;
/**
* @brief Le o estado de um novo segundo
*/
uint8_t RTC_IT_segundo () {
	return estado;
}
/**
* @brief Reseta o estado do segundo
*/
void RTC_reseta_IT_segundo () {
	estado = 0;
}
/**
* @brief ISR para tratar o evento Alarme A e B
*/
void RTC_Alarm_IRQHandler (void) {
	if (RTC->SR & RTC_MISR_ALRAMF) {
		RTC->SCR |= RTC_SCR_CALRAF; // Limpar a flag de interrupção do alarme A
		// Alternar o estado do PTB0
		GPIOB->ODR ^= (1 << 0);
		estado = 1;
```

```
	}
	if (RTC->SR & RTC_MISR_ALRBMF) {
		RTC->SCR |= RTC_SCR_CALRBF; // Limpar a flag de interrupção do alarme B
		// Alternar o estado do PTE1
		GPIOE->ODR ^= (1 << 1);
	}
}
```

Essencialmente `RTC_Alarm_IRQHandler` limpa a *flag* gerada no RTC em RTC_SR e verifica a fonte do evento. Se o evento for do Alarme A, o LED verde (PB0) é alternado. Se o evento for do Alarme B, o LED amarelo (PE1) é alternado. Além disso, foi definido uma variável estática `estado` e criado duas funções para acessar esta variável. Veremos mais adiante que esta variável é usada pela função `main` no arquivo `main.c` para atualizar o horário que poderia ser exibido em um *display*, acessando o horário atualizado do RTC a cada segundo.

18. Vamos inserir no aquivo `main.c`, depois da rotina `RTC_PInit`, a função `LD1_LD2_PInit` que ativa os pinos que controlam os LEDs amarelo e verde.

```
void LD1_LD2_PInit (void) {
	{
		// Inicializa GPIOB, pino 0 (PB0)
		RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk;
		// PB0 como saida digital
		GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk);
		GPIOB->MODER |= GPIO_MODER_MODE0_0;
		// PB0 na configuracao push-pull
		GPIOB->OTYPER &= ~GPIO_OTYPER_OT0_Msk;
	}
	{
		// Inicializa GPIOE, pino 1 (PE1)
		RCC->AHB4ENR |= RCC_AHB4ENR_GPIOEEN_Msk;
		// PE1 como saida digital
		GPIOE->MODER &= ~(GPIO_MODER_MODE1_Msk);
		GPIOE->MODER |= GPIO_MODER_MODE1_0;
		// PE1 na configuracao push-pull
		GPIOE->OTYPER &= ~GPIO_OTYPER_OT1_Msk;
	}
}
```

Em primeiro lugar, configuramos com três registradores o pino PB0 que alimenta o LED verde. É necessário habilitar o sinal de relógio do módulo GPIOB através do registrador RCC_AHB4ENR, configurar a função digital do pino PB0 através do registrador GPIOB_MODER e o modo de saída de PB0 pelo registrador GPIOB_OTYPER.
De forma análoga, configuramos com três registradores o pino PE1que alimenta o LED amarelo.

Cabe observar que os parênteses que separam as instruções em dois blocos na função `LD1_LD2_PInit` não têm nenhum efeito funcional no código C. Eles são apenas uma forma de agrupar visualmente as instruções para melhorar a legibilidade e organização do código. Não é uma prática comum em programação C.

19. Por fim, vamos declarar duas estruturas de dados depois da linha `/* USER CODE BEGIN PV */`

```
static struct {
	uint8_t HH;
	uint8_t MM;
	uint8_t SS;
} horario;
struct {
	uint8_t ano;
	uint8_t mes;
	uint8_t dia;
} dia;
```

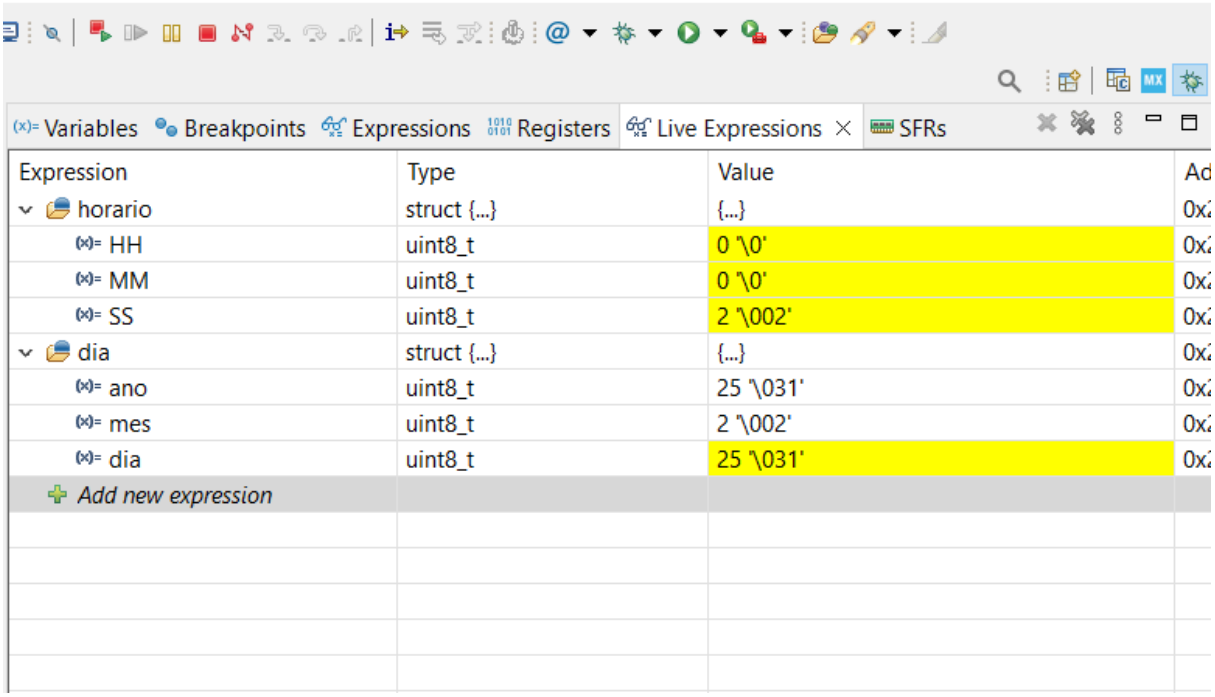
e inserir o seguinte bloco de códigos para atualizar o conteúdo de `horario` e `dia` a cada segundo depois da linha `/* USER CODE BEGIN 3 */`

```
if (RTC_IT_segundo()) {
	RTC_reseta_IT_segundo();
	// Ler a hora
	tmpreg = RTC->TR;
	// Extrair componentes de hora
	horario.HH = ((tmpreg & RTC_TR_HT) >> RTC_TR_HT_Pos) * 10
		   + ((tmpreg & RTC_TR_HU) >> RTC_TR_HU_Pos);
	horario.MM = ((tmpreg & RTC_TR_MNT) >> RTC_TR_MNT_Pos) * 10
		   + ((tmpreg & RTC_TR_MNU) >> RTC_TR_MNU_Pos);
	horario.SS = ((tmpreg & RTC_TR_ST) >> RTC_TR_ST_Pos) * 10
		   + ((tmpreg & RTC_TR_SU) >> RTC_TR_SU_Pos);
	// Ler a data
	tmpreg = RTC->DR;
	// Extrair componentes de data
	dia.ano = ((tmpreg & RTC_DR_YT) >> RTC_DR_YT_Pos) * 10
		+ ((tmpreg & RTC_DR_YU) >> RTC_DR_YU_Pos) ;  // Ajuste a partir de 2000
	dia.mes = ((tmpreg & RTC_DR_MT) >> RTC_DR_MT_Pos) * 10
		+ ((tmpreg & RTC_DR_MU) >> RTC_DR_MU_Pos);
	dia.dia = ((tmpreg & RTC_DR_DT) >> RTC_DR_DT_Pos) * 10
		+ ((tmpreg & RTC_DR_DU) >> RTC_DR_DU_Pos);
}
```

20. Faça “Build” e transfira o código executável para o microcontrolador no modo “Debug”.

21. Conecte o canal 0 e o terra do analinador lógico ao pino PB0 do conector ZIO e a um pino GND da placa. “Start” a captura do sinal no Logic do analisador e continue (“Resume”) a execução do programa no IDE. Páre a captura e meça a largura do pulso. O valor medido é condizente com o esperado? Justifique.

22. Abra a aba “Live Expression” na perspective Debug do IDE. Adicione as expressões `horario` e `dia`, como ilustra a figura abaixo. O que acontece com os valores na coluna “Value” se continuarmos (“Resume”) a execução do programa? Você conseguiria explicar o que você observou?



23. Qual o impacto no circuito se as duas últimas linhas de código, que configuram as interrupções externas (EXTI), forem removidas? Pelo que você observou, tente explicar com suas palavras o seguinte texto extraído do Manual de Referência:

"*The EXTI event inputs that are connected to the CPU NVIC are indicated in the Connection to NVIC column. For the EXTI events that do not have a connection to the NVIC, the peripheral interrupt is directly connected to the NVIC in parallel with the connection to the EXTI.*"

FUNDAMENTOS TEÓRICOS

Após explorarmos a variedade de temporizadores presentes em microcontroladores e sua importância em sistemas embarcados, vamos entender a fundo a teoria por trás dessa tecnologia. Serão abordados os componentes dos temporizadores utilizados nos projetos-exemplo, bem como as equações matemáticas que descrevem seu funcionamento.

PRINCÍPIOS DE OPERAÇÃO

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*:

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

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:

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

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*.

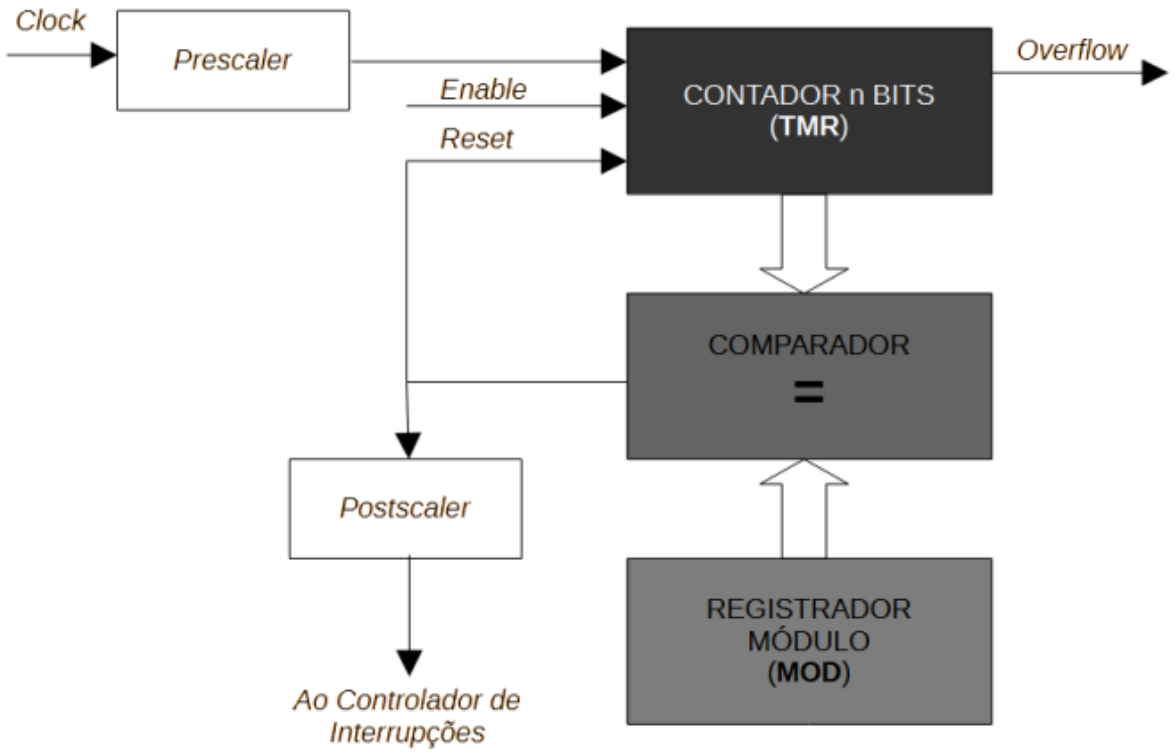


Diagrama de blocos de um temporizador básico

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 rígidamente 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.

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.

INTEGRIDADE

A integridade temporal e de dados em temporizadores é essencial para o funcionamento correto de circuitos e sistemas, como microcontroladores e processadores. A **integridade temporal** garante que a operação do temporizador (como a contagem de ciclos de *clock*) ocorra de forma sincronizada com o sistema, evitando erros de temporização, como contagens incorretas ou desincronizadas. A **integridade de dados**, por sua vez, protege contra erros ou corrupção dos dados armazenados nos registradores, que armazenam o valor de contagem dos temporizadores.

A metaestabilidade, um estado instável ou indefinido que pode ocorrer em *flip-flops* quando um sinal de entrada muda próximo à borda do sinal de *clock*, é uma falha comum em sistemas de temporização. Outra falha é a leitura incorreta do valor do temporizador durante a atualização, resultando em dados parciais ou corrompidos. Para garantir a precisão e confiabilidade dos dados e do tempo, aplicam-se técnicas que evitam essas falhas.

O ***reset* síncrono** é uma técnica comum para garantir a integridade temporal. Ele reinicia o temporizador ou a contagem de tempo de forma sincronizada com o sinal de *clock* do sistema, prevenindo inícios ou reinícios indesejados que comprometeriam a precisão da medição de tempo.

Para garantir a integridade de dados, utiliza-se **registradores sombra** (*shadow registers*). O registrador sombra é uma cópia do registrador original que armazena o valor do temporizador. A leitura do valor do temporizador é feita no registrador sombra, enquanto o temporizador real continua a ser atualizado sem interferência. Isso evita a corrupção de dados, pois o valor lido é estável e consistente, mesmo durante a atualização do temporizador. É importante destacar que a atualização do registrador sombra ocorre de forma síncrona com o sinal de *clock* do sistema. Isso significa que a transferência do valor do temporizador do registrador principal para o registrador sombra é realizada em um momento específico, determinado pela borda do *clock*. Essa sincronização garante que a leitura do valor do temporizador seja sempre consistente, mesmo durante a atualização.

MECANISMOS DE PROTEÇÃO DOS REGISTRADORES

A proteção contra acesso de escrita em registradores de periféricos é uma característica comum em microcontroladores modernos, especialmente aqueles voltados para aplicações que exigem segurança e confiabilidade. Essa é uma tendência crescente na indústria de sistemas embarcados, impulsionada pela necessidade de lidar com informações sensíveis e proteger os sistemas contra ataques cibernéticos, erros de *software*, e garantia de integridade de sistemas críticos, como sistemas médicos, automotivos, aeronáuticos e náuticos, onde falhas podem colocar em risco a vida de pessoas.

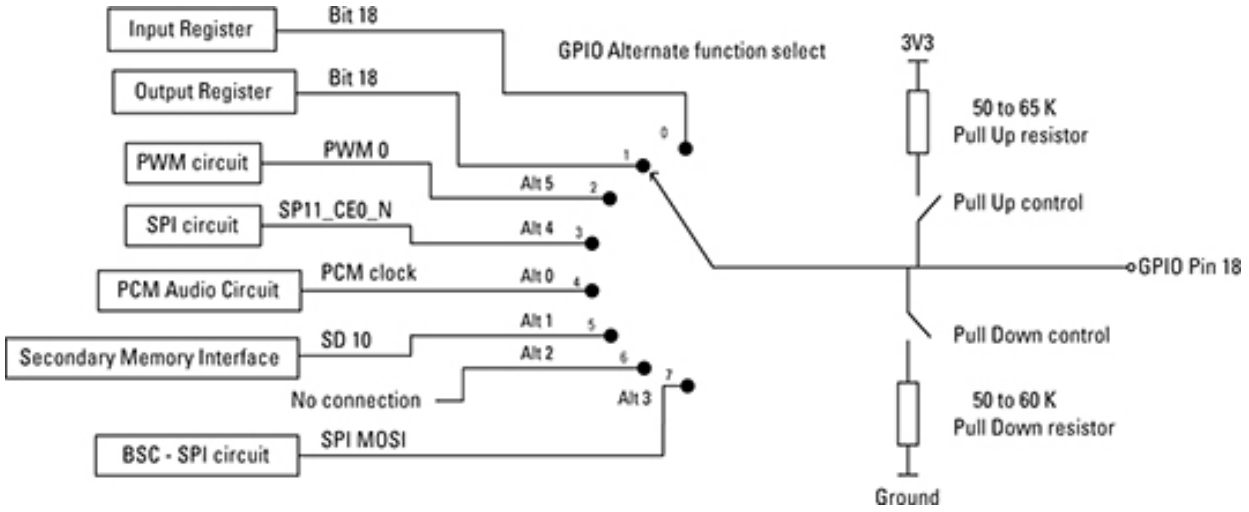
Para aumentar a confiabilidade e segurança desses sistemas, são incluídos registradores especiais que controlam o acesso a outros registradores. A forma de controle pode variar dependendo do microcontrolador e do periférico, mas geralmente envolvem o uso de **registradores de chave** (em inglês, *key registers*) ou outros mecanismos de controle de acesso. A ideia básica é que, para modificar a configuração de um periférico, o *software* precise seguir um protocolo específico, que pode incluir a escrita de um valor específico em um registrador de chave ou a habilitação de um *bit* de controle. Essa sequência é como uma senha secreta, conhecida apenas pelo *software* confiável. Apesar de introduzirem alguma complexidade na programação, os benefícios que esses mecanismos oferecem em termos de proteção contra erros e acesso não autorizado compensam essa complexidade.

Esses mecanismos de proteção estão presentes em muitos temporizadores, como *watchdog* e RTC, e controladores de memória, entre outros. A proteção no *watchdog* impede que ele seja desativado ou

configurado incorretamente, garantindo que o sistema seja reiniciado em caso de falha. A proteção de escrita nos registradores do RTC é uma funcionalidade essencial para garantir a integridade e a estabilidade de sistemas de tempo real, evitando falhas ou ataques que comprometam o funcionamento do sistema.

PINOS MULTIPLEXÁVEIS

A técnica de **multiplexação de pinos** permite a otimização do uso dos pinos disponíveis. Em vez de ter pinos dedicados exclusivamente a uma única função, a multiplexação possibilita que um único pino desempenhe várias funções diferentes, dependendo de como o sistema é configurado. Cada pino multiplexável em um microcontrolador pode ser configurado para ter uma função primária, que é a função padrão após o *reset*. Além dessa **função primária**, o pino pode ter uma ou mais **funções alternativas**, como interfaces de comunicação serial (SPI, I2C, UART), sinais de temporizadores ou canais de ADC/DAC. A seguinte figura ilustra a função primária (Output/Input) e funções alternativas de um pino de Raspberry Pi.



A multiplexação reduz a necessidade de um número elevado de pinos, o que simplifica o desenho da placa de circuito impresso (PCB) e reduz custos de fabricação. Além disso, proporciona uma grande flexibilidade, permitindo que o microcontrolador se adapte facilmente a diferentes aplicações e requisitos ao reconfigurar os pinos.

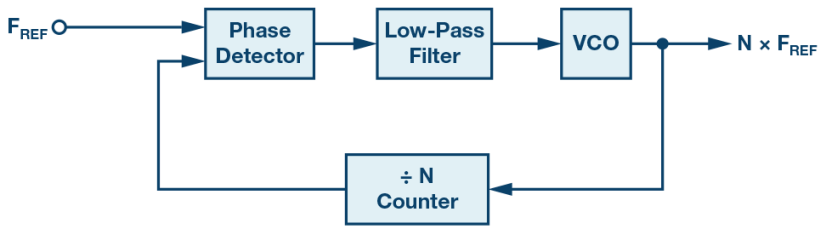
Para gerenciar a multiplexação, os microcontroladores utilizam registradores de configuração do módulo GPIO (do inglês *General Purpose Input/Output*) que permitem ao usuário selecionar a função desejada para cada pino. Cada função alternativa é associada a um código binário específico, e esses registradores controlam a configuração dos pinos com base nesses códigos, ajustando-os para a função principal ou para uma das funções alternativas. Circuitos internos, como multiplexadores, são responsáveis por direcionar os sinais para os pinos apropriados de acordo com a configuração definida. Muitos microcontroladores suportam ainda **funções adicionais** à função primária ou alternativa já selecionada. Essas funções adicionais são tipicamente habilitadas/configuradas diretamente nos registradores do *periférico* correspondente.

FONTES DE SINAIS DE RELÓGIO

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.



Em microcontroladores modernos, a alta densidade de circuitos integrados resulta em um grande volume de transições de sinal quando o dispositivo é energizado, o que leva a um consumo significativo de energia dinâmica. Para mitigar esse consumo, a técnica de "*clock gating*" é amplamente adotada. Essa técnica de gerenciamento de energia desativa o sinal de *clock* para partes do circuito que não estão em uso, evitando transições desnecessárias e, consequentemente, reduzindo o consumo de energia. Dessa forma, o "*clock gating*" complementa o funcionamento do PLL, otimizando o consumo de energia em sistemas que exigem alta precisão e estabilidade de *clock*.

TEMPORIZADORES DEDICADOS

No mundo da eletrônica e da programação embarcada, o tempo é um recurso fundamental, e os temporizadores são ferramentas essenciais para controlá-lo com precisão. Seja para piscar LEDs, controlar motores, gerar sinais PWM ou sincronizar protocolos de comunicação, os temporizadores estão presentes em inúmeras aplicações. Embora suas finalidades sejam diversas, todos os temporizadores compartilham um princípio básico: um contador que avança a cada pulso de relógio, permitindo medir e controlar intervalos de tempo com exatidão. No entanto, para atender às diferentes necessidades dos sistemas embarcados, os temporizadores incorporam recursos adicionais que os tornam ainda mais versáteis, como geração de interrupções periódicas, pulso único, *watchdogs* e relógios de tempo real (RTC). Nesta seção, exploraremos essas variantes de temporizadores e suas aplicações, analisando como cada um pode ser utilizado de forma eficiente para otimizar o desempenho dos sistemas embarcados.

Temporizador de Interrupções Periódicas

Um **temporizador de interrupções periódicas** é um tipo de temporizador que, além de contar pulsos de *clock* como um temporizador básico, possui um mecanismo de geração de interrupções em intervalos regulares de tempo. Esse recurso permite que o processador seja alertado automaticamente sem a necessidade de verificar constantemente o estado do temporizador, liberando-o para outras tarefas. Assim, o uso de temporizadores de interrupções periódicas reduz a carga da CPU, pois elimina a necessidade de *polling*, permitindo que o processador execute outras tarefas ou entre em modos de economia de energia enquanto aguarda a próxima interrupção. Isso melhora a eficiência do sistema e possibilita a criação de aplicações mais responsivas e com menor consumo energético.

Para isso, é necessário agregar ao temporizador básico o mecanismo de geração de interrupções que sinaliza ao processador ou a um controlador de interrupções quando o tempo programado é atingido. Entre as principais aplicações que temporizadores de interrupções periódicas suportam estão as tarefas que precisam ser executadas em intervalos de tempo bem definidos, como amostragem de sensores, o controle de alocação de tempo para multitarefas e geração de sinais de relógio para protocolos de comunicação serial.

Temporizador de Interrupções Periódicas Síncronas

Os **temporizadores de interrupções periódicas síncronas** são uma variação dos temporizadores periódicos clássicos, diferenciando-se pela capacidade de sincronizar os eventos temporais com o ciclo de execução do sistema. Isso garante que as interrupções ocorram em momentos bem definidos dentro do fluxo de execução do *software*, permitindo um processamento coordenado com o processador. Como resultado, minimizam-se atrasos inesperados, melhora-se a previsibilidade da execução e garante-se que tarefas sensíveis ao tempo sejam executadas com precisão em sistemas

embarcados. Além disso, a sincronização eficiente elimina a necessidade de verificações frequentes do estado do sistema, reduzindo o consumo de energia.

Enquanto os temporizadores periódicos clássicos geram interrupções em tempos fixos, independentemente do estado do processador, os temporizadores síncronos ajustam a geração das interrupções para alinhá-las aos sinais de relógio do processador, reduzindo latências indesejadas. Para isso, contam com um mecanismo de sincronização com o barramento do sistema, que ajusta o tempo da interrupção com base no ciclo de execução do processador. Uma abordagem comum para garantir essa sincronização é utilizar o mesmo sinal de relógio do processador como base para o contador do temporizador.

Graças ao seu alto grau de determinismo e precisão no tempo de resposta, a principal aplicação dos temporizadores de interrupções periódicas síncronas está no controle de tarefas em sistemas operacionais de tempo real (em inglês, *Real-Time Operating System* - RTOS), onde a previsibilidade na execução de processos é essencial.

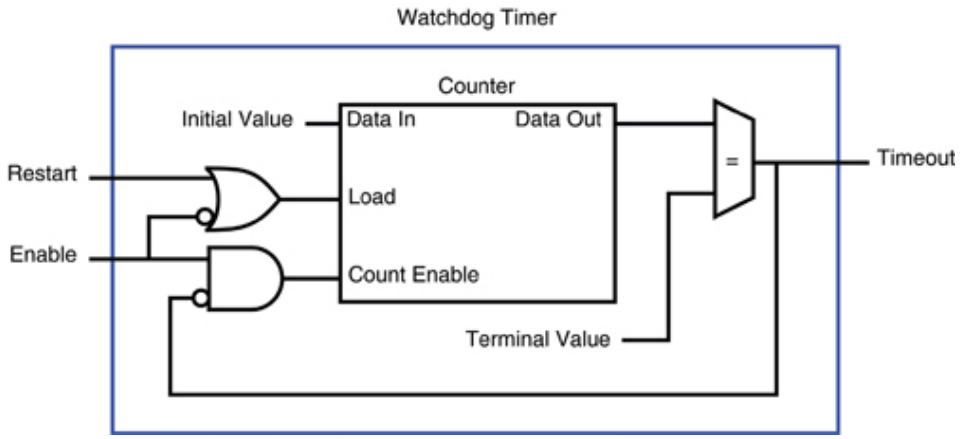
Temporizador de Pulso Único

Um **temporizador no modo de pulso único** (em inglês, *one-shot timer*), amplamente utilizado em circuitos de retardo, executa a contagem uma vez e responde a um único evento. Nestes circuitos, a execução de uma ação sobre um dispositivo é adiada por um intervalo de tempo após o desencadeamento dessa ação, ocorrendo apenas uma vez. Ele pode ser aplicado na implementação de um relé de retardo (em inglês, *time delay relay*), um dispositivo eletromecânico projetado para proporcionar um atraso controlado na comutação. Ao ser acionado, o relé utiliza o módulo de captura de entrada para registrar o valor do temporizador no momento do acionamento. Posteriormente, emprega o módulo de comparação de saída para continuamente comparar o valor do temporizador com um valor de referência. Esse valor de referência é determinado com base no valor capturado, e a comparação define o término do atraso. Essa abordagem oferece maior precisão e flexibilidade na implementação de atrasos controlados, além de permitir uma integração mais eficaz do relé com o controle global do sistema, em comparação com métodos de implementação mais tradicionais.

Temporizador de Vigilância

O **temporizador de vigilância**, mais conhecido por temporizador de ***watchdog***, é um dispositivo de *hardware* ou *software* que monitora o funcionamento de um sistema e garante que ele esteja funcionando corretamente. O temporizador de *watchdog* é iniciado quando o sistema é iniciado, e é reiniciado periodicamente pelo *software* do sistema. Se o *software* do sistema falhar e não reiniciar o temporizador dentro do tempo limite (do inglês, *timeout*), o temporizador assume que o sistema falhou e toma a ação corretiva, como resetar o sistema ou enviar uma mensagem de erro. Na figura que se segue, o sinal que reinicia o *watchdog* corresponde ao sinal “Restart”. O circuito *watchdog* possui um temporizador interno que é reiniciado cada vez que recebe o sinal “Restart” do *software* do sistema. Se o temporizador interno do *watchdog* não receber o sinal “Restart” dentro do intervalo de tempo pré-determinado, significa que algo não está funcionando como deveria no sistema

“vigiado”. Nesse caso, o circuito *watchdog* entra em ação e envia um sinal de “Timeout” (esgotamento de tempo) para o sistema, usualmente resetando o mesmo, a fim de fazer que o sistema retorne a um estado funcional.



Tipicamente, o circuito *watchdog* opera em conjunto com o processador para monitorar o funcionamento correto do sistema. Para isso, o programa em execução no processador precisa enviar periodicamente um sinal de “Restart” ao *watchdog*. Essa ação indica que o processador está funcionando corretamente. Caso o processador falhe e não envie o sinal dentro de um tempo limite, o *watchdog* assume que houve uma falha e reseta o sistema, evitando assim comportamentos inesperados. Devido à sua importância na prevenção de falhas, os temporizadores *watchdog* são amplamente utilizados em sistemas embarcados críticos, como em aplicações de automação industrial, controle de veículos e sistemas de segurança, para garantir a continuidade do funcionamento do sistema mesmo em caso de falha.

Relógio em Tempo Real

O circuito de **relógio em tempo real** (do inglês, *Real Time Clock* - RTC) 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. Muitos RTCs incluem alarmes programáveis que facilitam a implementação de funções de temporização. Este circuito é altamente relevante em sistemas embarcados por várias razões:

- Fornecimento de referência de tempo exata: O circuito RTC mantém a hora e a data com exatidão, mesmo quando o sistema é desligado ou desconectado da fonte de alimentação principal. Isso é importante para aplicações que precisam de um registro exato do tempo, como sistemas de monitoramento, sistemas de *log* de eventos e sistemas de tempo real.
- Gerenciamento de tarefas agendadas: O circuito RTC pode ser usado para agendar tarefas, como atualizações de *firmware*, coleta de dados e transmissão de dados. Isso permite que o sistema embarcado execute tarefas automaticamente, sem intervenção humana.
- Gerenciamento de energia: o circuito RTC pode ser usado para gerenciar o consumo de energia do sistema, desligando componentes desnecessários quando não estão sendo usados

e acionando-os quando necessário, economizando energia e prolongando a vida útil da bateria.

- Comunicação com outros dispositivos: O RTC pode ser usado como uma referência de tempo para dispositivos externos, como servidores ou outros sistemas embarcados. Isso permite que esses dispositivos sincronizem suas operações com o sistema embarcado.
- Detecção de falhas: O circuito RTC pode ser usado para detectar falhas no sistema, como a perda de sincronização do *clock* ou a interrupção da fonte de alimentação. Isso permite que o sistema embarcado tome medidas para mitigar essas falhas e garantir a continuidade do funcionamento.

Para isto, o circuito utiliza um oscilador 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 pode contar com um circuito de alarme que permite a geração automática de eventos de interrupção. Esse mecanismo funciona comparando continuamente o tempo atual do RTC com um valor de alarme previamente configurado. Quando ocorre uma coincidência entre os dois, o RTC aciona uma interrupção, permitindo que o sistema execute uma ação programada no momento exato. Essas interrupções podem ser utilizadas de diferentes formas:

- **Geração de eventos periódicos**: Permite a execução de tarefas em intervalos regulares, como a atualização de um *display* ou a coleta de dados em sistemas de monitoramento.
- **Alarmes programáveis**: Acionam eventos em horários específicos, úteis em aplicações como despertadores e sistemas de controle automatizado.
- **Despertar o microcontrolador**: Quando o microcontrolador está em **modo de suspensão** para economizar energia, o circuito de alarme pode ativá-lo no momento adequado, garantindo eficiência energética sem comprometer a funcionalidade do sistema.

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 pode ser encontrado tanto como um circuito integrado dedicado (como o DS1307, o DS3231 ou o PCF8563) quanto integrado ao próprio microcontrolador. Quando o RTC é um circuito integrado separado, ele se comunica com o microcontrolador através de uma interface digital. Nos microcontroladores que já possuem um RTC integrado, seus registradores são acessíveis da mesma forma que os demais periféricos internos, através de instruções de leitura e escrita na memória. Essa integração simplifica o acesso aos recursos do RTC, otimizando o código e facilitando a configuração e o controle do tempo.

STM32H7A3

Os microcontroladores STM32H7A3 incluem os circuitos de multiplexação no módulo GPIO e diversos 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 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 **controlador de habilitação 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)

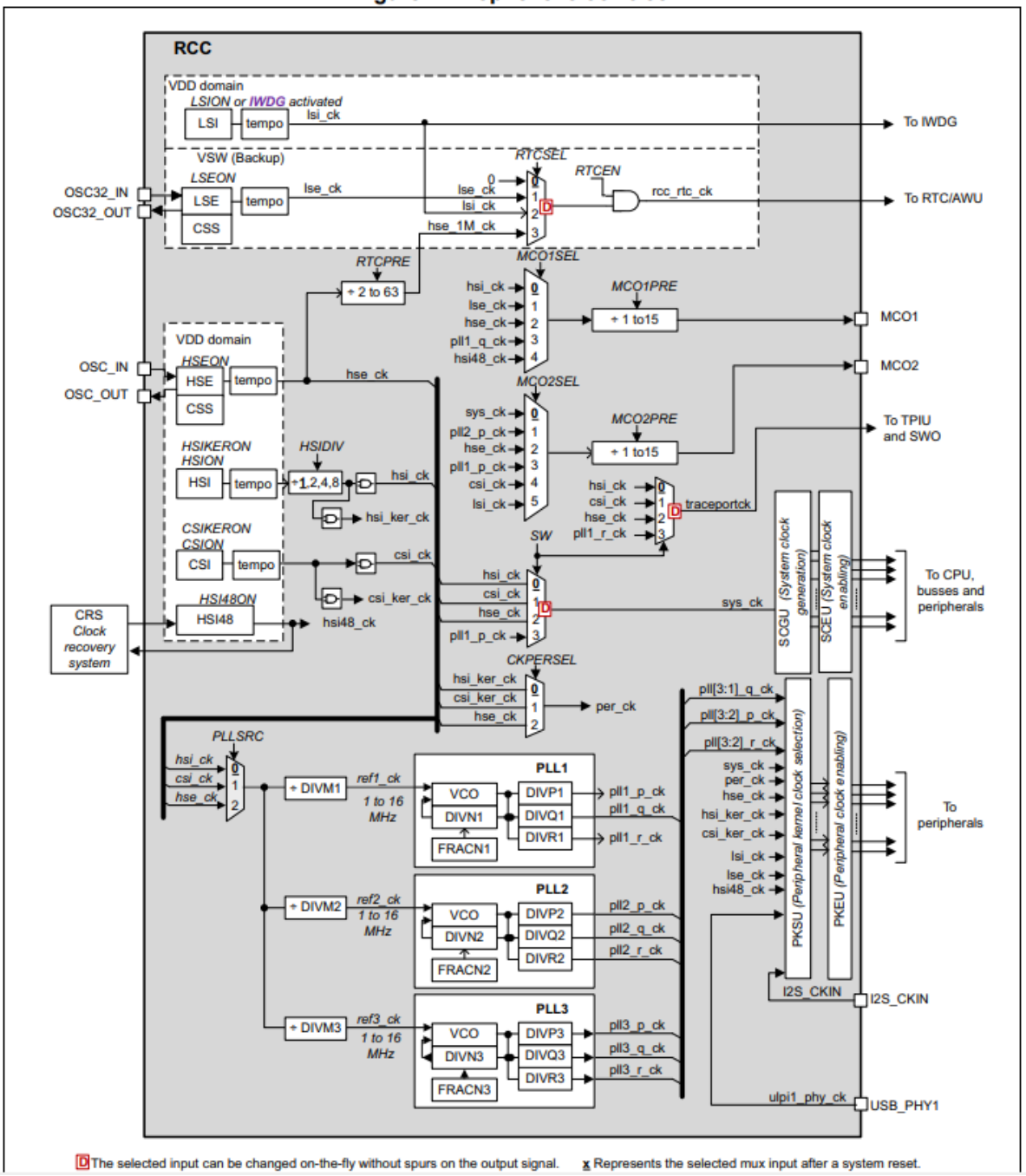
Address offset: 0x000

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

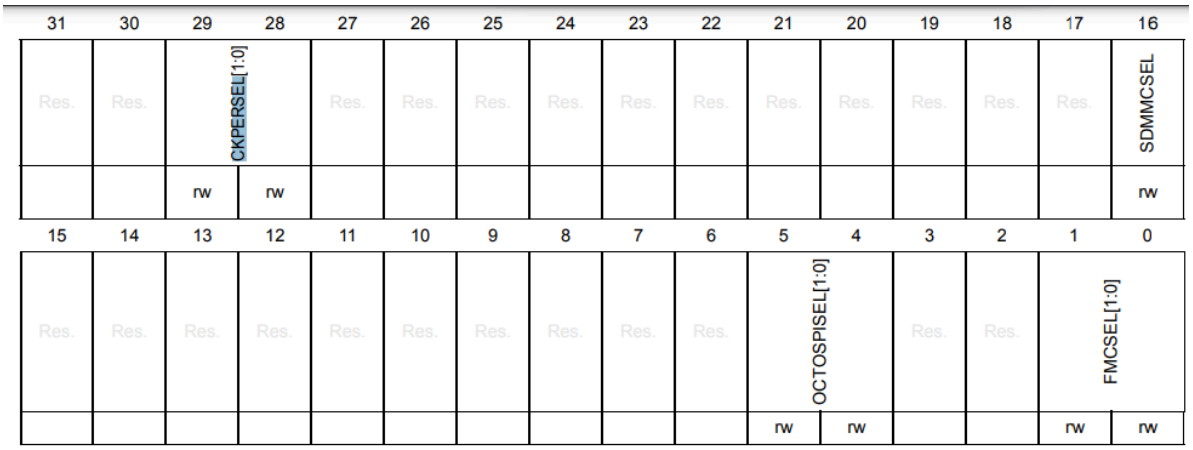
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	DIVM3[5:0]						Res.	Res.	DIVM2[5:4]	
						rw	rw	rw	rw	rw	rw			rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIVM2[3:0]				Res.	Res.	DIVM1[5:0]						Res.	Res.	PLLSRC[1:0]	
rw	rw	rw	rw			rw	rw	rw	rw	rw	rw			rw	rw

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*.



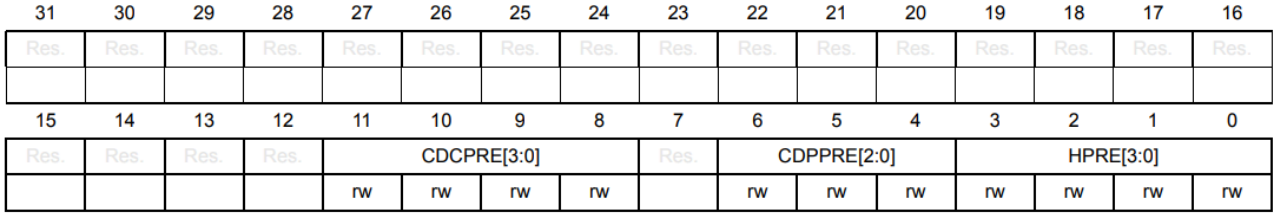
Bits 31:30 Reserved, must be kept at reset value.

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



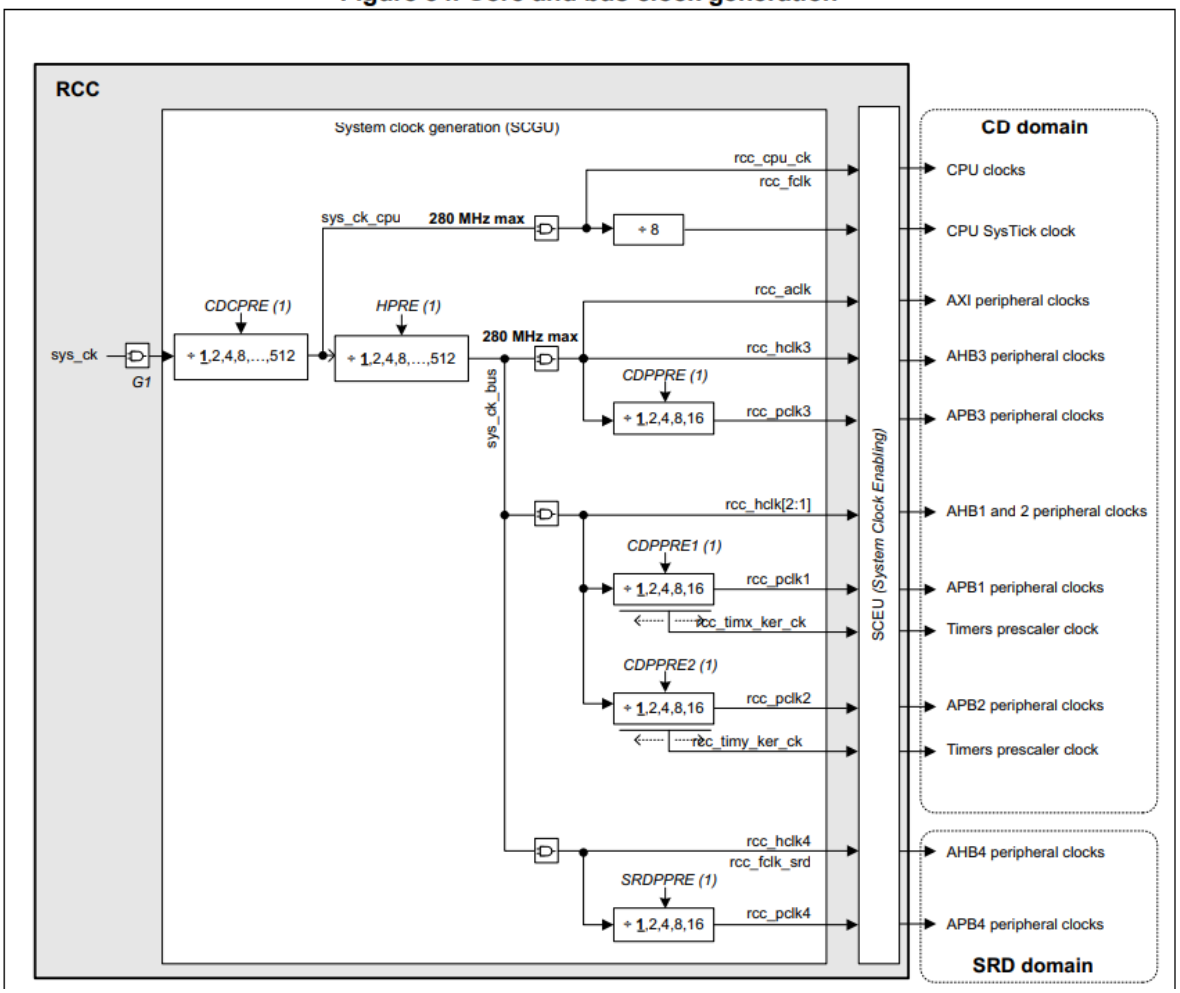
Segundo a descrição funcional do Manual de Referência, após uma reinicialização do sistema, o HSI é selecionado como o relógio do sistema e todos os PLLs são desligados. Quando uma fonte de relógio é utilizada para o sistema, o *software* não pode desativar essa fonte usando os *bits* xxxON. No entanto, 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 apresenta uma visão mais detalhada 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



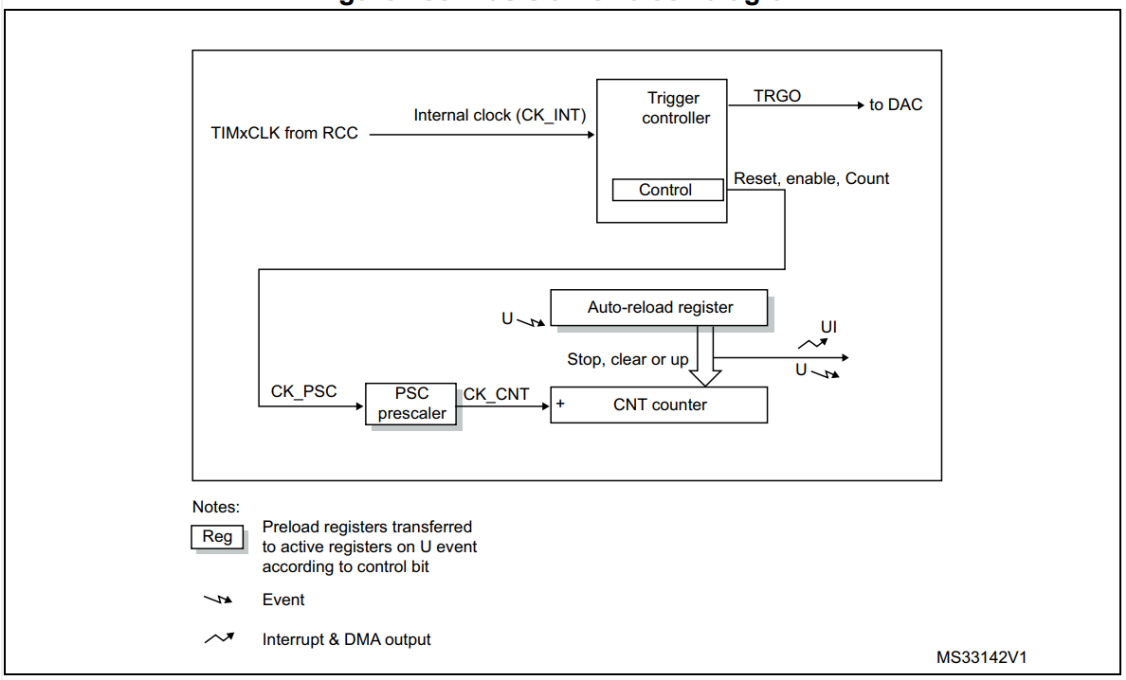
O microcontrolador é equipado com dois conjuntos principais de barramentos, como mostra a figura 1 do *Datasheet*: o *Advanced High-performance Bus* (AHB) e o *Advanced Peripheral Bus* (APB). O barramento **AHB** conecta o núcleo do processador e a memória a periféricos de alta velocidade, como a memória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

TIM6/TIM7

O componente 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:

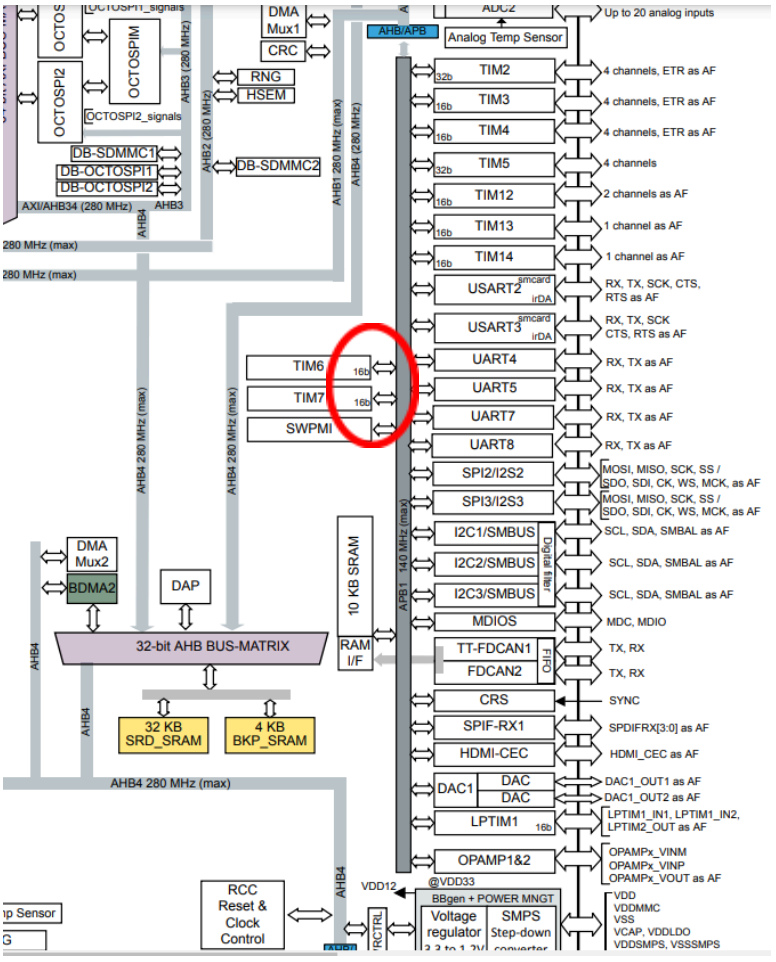
- 2 Registradores de Controle (`TIMx_CR1` e `TIMx_CR2`)
- Registrador de habilitação de interrupção e DMA (`TIMx_DIER`)
- Registrador de estado (`TIMx_SR`)
- Registrador de geração de evento (`TIMx_EGR`).

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 dois temporizadores estão conectados 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*).



O mecanismo de interrupção implementado nos temporizadores básicos TIM6 e TIM7 envolve a habilitação de uma interrupção no evento de atualização (*update* ou *overflow* do contador) pelo *bit*

`TIMx_DIER_UIE` do registrador `TIMx_DIER` e o tratamento direto dessa interrupção pelo NVIC (*Nested Vectored Interrupt Controller*), sem passar pelo EXTI (*Extended Interrupt/Event Controller*). Cada temporizador possui seu próprio canal de interrupção dedicado no NVIC, correspondente aos vetores de interrupção 54 e 55, conforme mostra o seguinte trecho da tabela do NVIC no Manual de Referência.

tim6_gbl_it	61	54	TIM6_DAC1	TIM6 global interrupt	0x0000 0118
dac1_unr_it				DAC1 underrun error interrupt	
tim7_gbl_it	62	55	TIM7	TIM7 global interrupt	0x0000 011C

Note que a presença de entradas dedicadas para as interrupções globais de TIM6 e TIM7 diretamente na tabela do NVIC confirma que o tratamento dessas interrupções ocorre sem a intermediação do EXTI. Portanto, a configuração de linhas EXTI e seus registradores (`EXTI_RTSR`, `EXTI_FTSR`, `EXTI_IMR`, etc.) não é relevante para o funcionamento das interrupções de TIM6 e TIM7.

A habilitação das interrupções de TIM6 e TIM7 no NVIC é realizada através da configuração dos *bits* correspondentes nos registradores de habilitação de interrupção do NVIC. A prioridade dessas interrupções também é configurável nos registradores de prioridade do NVIC. Com o canal de interrupção habilitado no NVIC, a interrupção será gerada quando os *bits* `TIMx_SR_UIF`, do registrador TIMx_SR, e `TIMx_DIER_UIE` forem '1'. Após o tratamento da interrupção, o *bit* `TIMx_SR_UIF` deve ser limpo para evitar novas ocorrências.

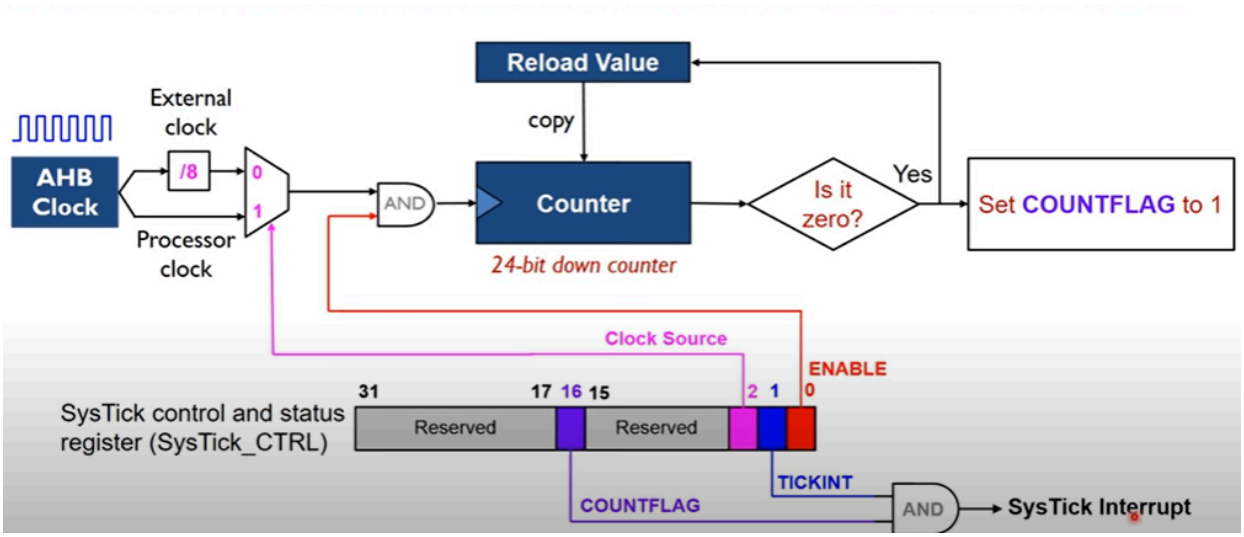
Systick (*Tick* do Sistema)

Muitos microcontroladores, incluindo os da família ARM Cortex-M, incorporam um temporizador especial chamado *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 (em inglês, *Real Time Operating System* – RTOS).

O *SysTick* é um componente integrado no processador ARM Cortex-M e faz parte do suporte de *hardware* da arquitetura Cortex-M. Ele tem um contador (em inglês, *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 um evento de interrupção. Quando o processador é interrompido enquanto está no estado de *Debug*, o contador `SYST_CVR` pára de decrementar. A fórmula geral para determinar o valor de SYST_RVR é (Período x frequência) - 1.

Nem sempre o SysTick opera na frequência esperada, pois ela pode ser derivada de fontes distintas. Além disso, em alguns sistemas embarcados, o *clock* pode variar dependendo da configuração de PLLs e divisores. O registrador `SYST_CALIB` contém um campo chamado `TENMS`, que indica

quantos ciclos de **clock** do SysTick correspondem a 10 ms. Isso é útil para calcular valores de temporização sem precisar conhecer a frequência exata do *clock*. Esse é o valor de contagem carregado na função de inicialização do microcontrolador gerada pelo STM32CubeMX, mas ele pode ser modificado no editor gráfico do STM32CubeMX ou dentro do código do usuário.



Em relação às fontes de sinais de relógio, configuráveis pelo registrador SYST_CSR (SysTick_CTRL na figura), o SysTick pode operar com duas fontes distintas: o *clock* do processador (SYST_CSR[2] = 1) ou uma fonte externa arbitrária (SYST_CSR[2] = 0). Essa fonte externa pode ser um *clock* derivado do *clock* do processador dividido por 8. Nos microcontroladores STM32H7A3/B3, conforme ilustrado na Figura 54 do Manual de Referência, a fonte externa é o *clock* da CPU dividido por 8.

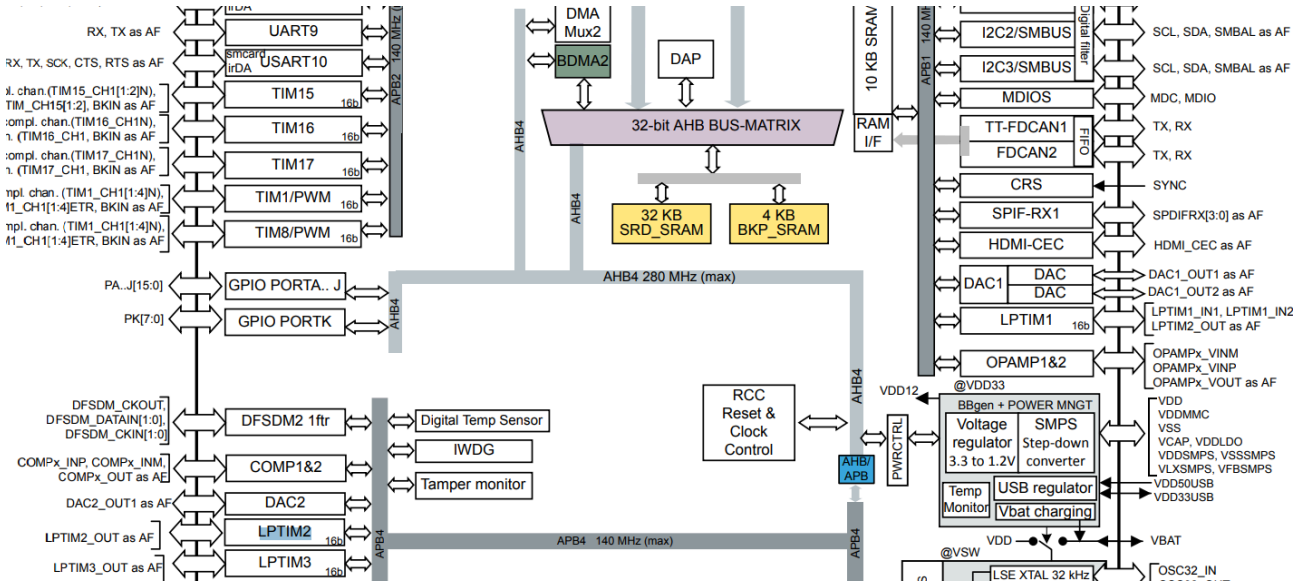
Integrado ao núcleo ARM Cortex-M, o SysTick gera interrupções sincronizadas diretamente com o clock do processador. Embora tratado como uma exceção do sistema, sua prioridade é gerenciada pelo controlador de interrupções e exceções NVIC, permitindo o ajuste de sua prioridade em relação a outras exceções e interrupções. Ao habilitar a interrupção do SysTick (SYST_CSR[1] = 1), o NVIC é configurado automaticamente para lidar com as interrupções geradas. O *bit* de estado SYST_CSR[16] é gerenciado pelo *hardware*, sendo limpo automaticamente após a leitura do SysTick ou a ocorrência da interrupção, dispensando a necessidade de limpeza manual. Além disso, o SysTick faz parte da arquitetura ISA ARM (*Instruction Set Architecture*), e sua programação é detalhada no Manual de Referência correspondente à versão específica da arquitetura ARM Cortex-M utilizada no microcontrolador. No nosso caso, devemos consultar o Manual de Referência de ARMv7-M.

Temporizador de Baixo Consumo (LPTIM)

O temporizador de baixo consumo (em inglês, *Low Power Timer* – LPTIM*)* é um temporizador de 16 *bits* projetado para operar em modos de baixo consumo de energia, mantendo-se funcional em diversos modos de energia, exceto no modo Standby. Ele pode ser usado como um contador de pulsos ou para funções de *timeout*, com baixo consumo energético. No microcontrolador STM32H7A3 está disponível em 3 instâncias: LPTIM1, LPTIM2 e LPTIM3. Apenas o LPTIM3

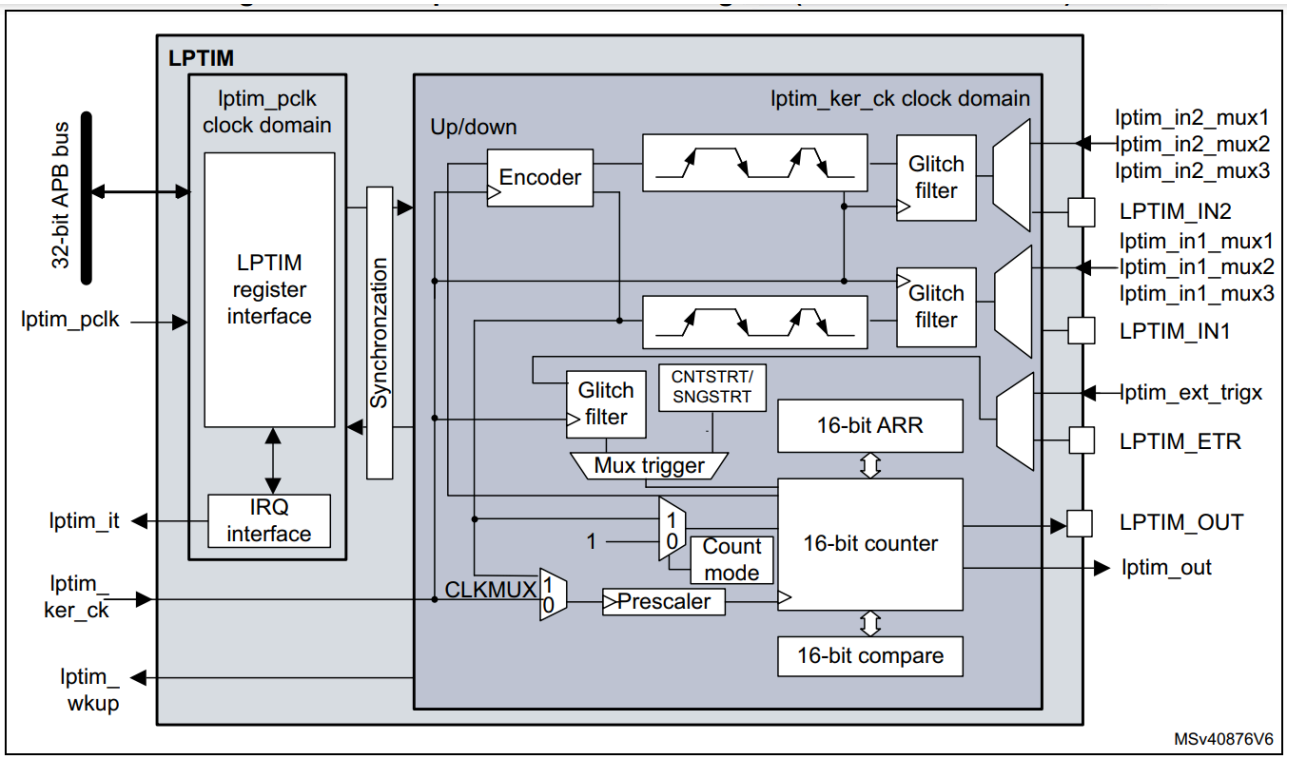
não possui o modo encoder para interagir com sensores de encoder incremental de quadratura e modificar s direção de sua contagem.

O diagrama de bloco do *Datasheet* mostra que a instância LPTIM1 está conectada ao barramento APB1, enquanto as instâncias LPTIM2 e LPTIM3 ao barramento APB4. Portanto, para habilitar o sinal de *clock* das instâncias LPTIM1 e LPTIM2/LPTIM3, é necessário setar os *bits* correspondentes nos registradores RCC_APB1LENR e RCC_APB4ENR, respectivamente. E, para resetar e setar essas instâncias por *software*, devemos usar, respectivamente, os registradores RCC_APB1LRSTR e RCC_APB4RSTR.



Porém, o LPTIM oferece flexibilidade na escolha da fonte de *clock*. Além do *clock* interno, configurável através do RCC, o periférico pode ser alimentado por um *clock* externo aplicado à entrada LPTIMn_IN1, operando como um contador de pulsos. A seleção entre as fontes interna e externa é controlada pelos *bits* `CKSEL` e `COUNTMODE`, localizados no registrador LPTIMn_CFGR. É importante observar que a alteração dos *bits* desse registrador só é permitida quando o periférico LPTIMn está desabilitado. Para isso, o *bit* ENABLE, presente no registrador LPTIMn_CR, deve ser colocado em nível lógico baixo.

O LPTIM possui um *Prescaler* que permite reduzir a frequência do sinal de *clock* que alimenta o contador interno do temporizador. Essa redução é configurada através dos 3 *bits* `PRESC` localizados no registrador LPTIMn_CFGR.



O LPTIM integra filtros de falhas (em inglês, *glitch filters*) que protegem as entradas do temporizador contra pulsos de ruído ou sinais transitórios indesejados, conhecidos como "*glitches*". Esses filtros são configuráveis pelos *bits* `CKFLT` (para entradas externas) e `TRGFLT` (para *triggers* internos) presentes no registrador LPTIMn_CFGR. Somente com o sinal de *clock* interno ativado, pode-se ativar os recursos de filtro de falhas.

A contagem do LPTIM pode ser iniciado por *software* ou por um *trigger* externo, configurável pelos *bits* `TRIGEN` presentes em LPTIMn_CFGR. A fonte do *trigger* externo é, por sua vez, selecionável através dos *bits* `TRIGSEL` de mesmo registrador. Três modos de contagem são suportados:

- **Modo Contínuo:** O contador opera continuamente até ser desabilitado, iniciado com o *bit* CNTSTRT do registrador LPTIMn_CR.
- **Modo *One-shot*:** O contador opera até atingir o valor máximo a ser contado LPTIMn_ARR, iniciado com o *bit* SNGSTRT do registrador LPTIMn_CR.
- **Modo *Set-once*:** O *bit* WAVE do LPTIMn_CFGR modifica o comportamento do modo *One-Shot*, descartando *triggers* subsequentes após o primeiro.

Os dois registradores de 16 *bits*, o de módulo LPTIMn_ARR e o de comparação LPTIMn_CMP, são responsáveis pela geração de formas de onda na saída do LPTIM, `lptim_out`, para cada modo de contagem (veja a figura):

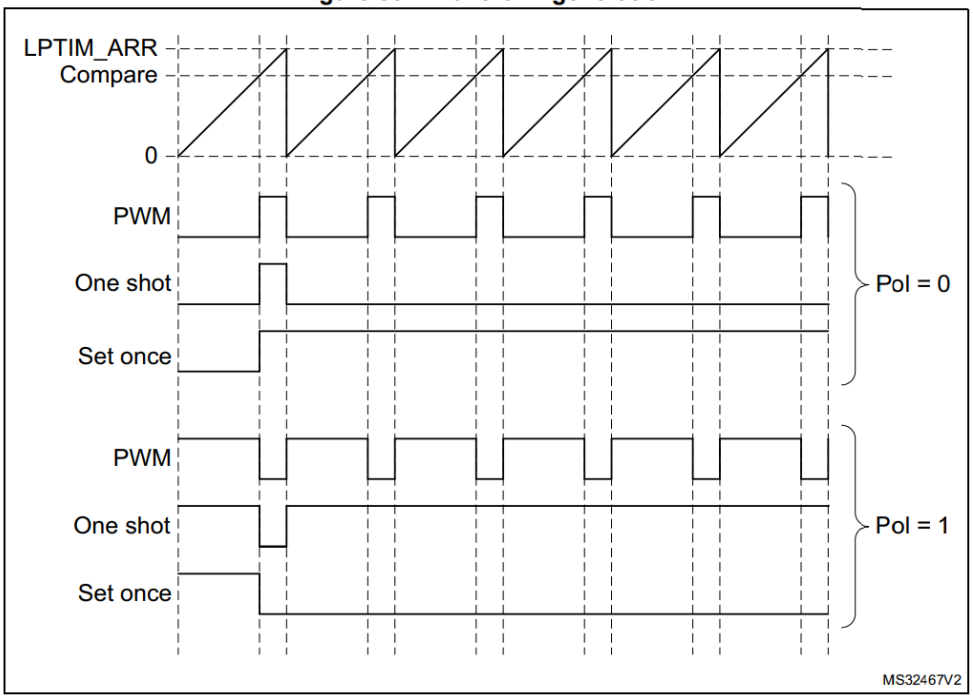
- **Modo Contínuo:** a saída do LPTIM é ativada assim que o valor do contador em LPTIMn_CNT excede o valor de comparação em LPTIM_CMP. A saída do LPTIM é

invertida assim que ocorre uma correspondência entre os registradores LPTIM_ARR e LPTIM_CNT, gerando um trem de pulsos de largura configurável. Porisso, este modo é também chamado modo PWM (do inglês *Pulse Modulation Width*).

- **Modo *One-shot*:** a forma de onda de saída é semelhante à do modo PWM para o primeiro pulso; em seguida, a saída é permanentemente resetado.
- **Modo *Set-once*:** a forma de onda de saída é semelhante à do modo *one-shot*, exceto que a saída é mantida no último nível de sinal.

A polaridade (Pol) do sinal de saída é controlada pelo *bit* WAVPOL localizado em LPTIMn_CFGR.

Figure 504. Waveform generation



Para alterar os valores dos registradores LPTIMn_ARR e LPTIMn_CMP, o periférico LPTIM deve estar abilitado. Os valores podem ser atualizados imediatamente ou ao final do período atual, de acordo com a configuração do *bit* PRELOAD em LPTIMn_CFGR. Os *flags* `ARROK` e `CMPOK` no registrador LPTIMn_ISR indicam quando a escrita nos registradores foi completada. É importante evitar escritas sucessivas antes que esses *flags* sejam setados.

O LPTIM pode ser ainda configurado para resetar o contador a cada *trigger*, implementando uma função de *timeout*, se sentarmos em '1' o *bit* TIMEOUT presente em LPTIMn_CFGR. Se nenhum *trigger* ocorrer dentro do período configurado em LPTIMn_CMP, o evento de "compare match", indicado pelo *bit* `CMPM` presente no registrador de estado LPTIMn_ISR, pode acordar o MCU.

Se habilitadas as interrupções no registrador LPTIMn_IER, os seguintes eventos podem gerar interrupções: *Compare Match*, *Autoreload Match*, Evento de *Trigger* Externo, *Autoreload Register Write Completed*, *Compare Register Write Completed* e Direção *UP/DOWN* (modo *encoder*). As interrupções de LPTIM1, LPTIM2 e LPTIM3 são diretamente roteadas para as linhas IRQ 93,

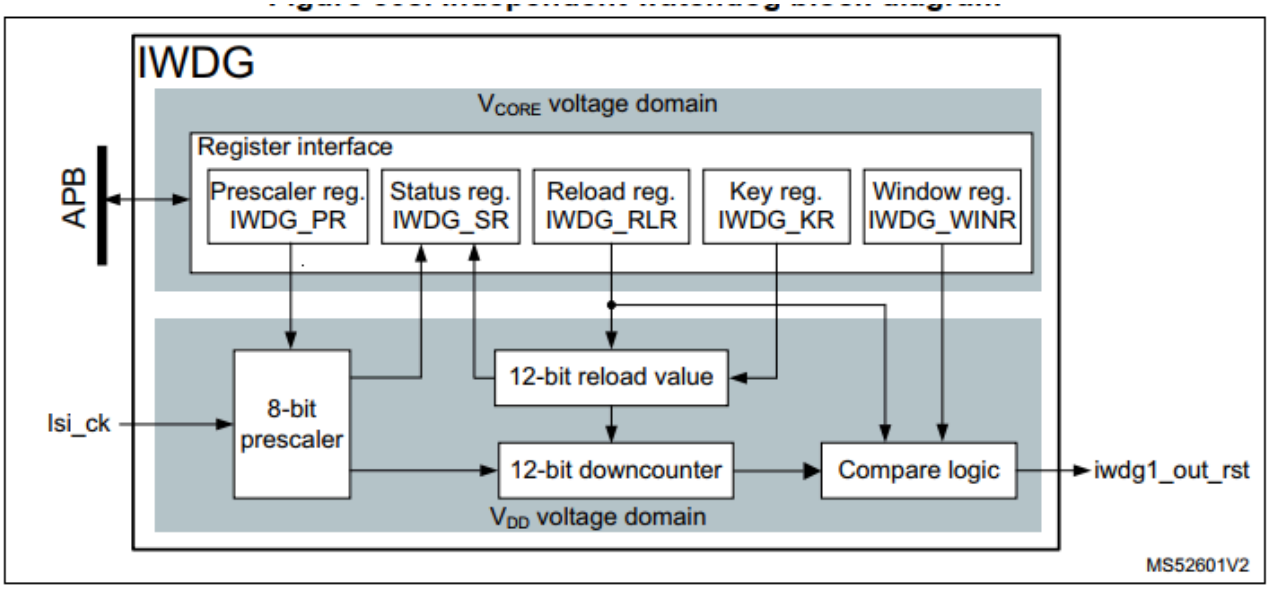
IRQ138 e IRQ 139 do NVIC, respectivamente. Neste caso, cada linha de interrupção é compartilhada por dois sinais, `lptim_it` e `exti_lptim_wkup`, que são combinados por uma operação OR lógica. Ou seja, tanto o sinal de interrupção, `lptim`, gerado pelo próprio periférico LPTIMn quanto o evento de *wakeup* do LPTIMn roteado através do EXTI, `exti_lptim_wkup`, são conectadas à mesma linha de interrupção do NVIC. Quando ocorre uma interrupção do temporizador LPTIMn ou um evento de *wakeup* do LPTIMn é detectado pelo EXTI, a mesma linha de interrupção no NVIC será ativada. O *software* então precisará verificar as *flags* de estado apropriadas (no LPTIMn e/ou no EXTI) para determinar a causa exata da interrupção.

lptim1_it exti_lptim_wkup	100	93	LPTIM1	LPTIM1 global interrupt	0x0000 01B4
lptim2_it exti_lptim2_wkup	145	138	LPTIM2	LPTIM2 timer interrupt	0x0000 0268
lptim3_it exti_lptim3_wkup	146	139	LPTIM3	LPTIM2 timer interrupt	0x0000 026C

Watchdogs

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. São integrados no STM32H7A3 dois tipos de temporizadores de *watchdog*: *System Window Watchdog* (WWDG) e *Independent Watchdog* (IWDG)**.** O **WWDG** detecta falhas de *software*, gerando um *reset* se o programa não atualizar um contador dentro de uma janela de tempo específica. O **IWDG**, por outro lado, é alimentado por um *clock* independente, o LSI (do inglês Low Speed Internal Oscillator) de 32kHz, sendo adequado para aplicações onde a precisão do tempo não é crítica. No entanto, é integrado ao IWDG o mecanismo de chave para proteger o acesso de escrita dos seus registradores IWDG_PR, IWDG_RLR e IWDG_WINR. Ambos os *watchdogs* funcionam como mecanismos de segurança, prevenindo que o sistema trave em estados indesejados, mas o IWDG continua a operar em modos de baixo consumo de energia, oferecendo maior robustez, e tem o mecanismo de proteção de acesso aos seus registradores.

A frequência do base de tempo do contador decrescente de 12 *bits* de IWDG é configurável através do *prescaler* IWDG_PR. Todas as atualizações no IWDG_PR e no reigistrador de módulo IWDG_RLR são indicadas nos *bits* PVU e RUV do registrador de estado IWDG_SR. IWDG pode operar com ou sem uma janela de tempo configurada.



O procedimento de configuração de IWDG sem janelamento de tempo recomendado no Manual de Referência compreende os seguintes passos:

1. Habilitar IWDG escrevendo 0x0000 CCCC no registrador de chave IWDG_KR.
2. Habilitar acessos de escrita dos registradores escrevendo 0x0000 5555 em IWDG_KR.
3. Configurar o divisor de frequência, programando 0 até 7 no registrador IWDG_PR.
4. Configurar o valor máximo de contagem no registrador de módulo IWDG_RLR.
5. Aguardar que IWDG_SR = 0x0000 0000, indicando que os registradores foram devidamente atualizados.
6. Carregar o contador com o valor setado no IWDG_RLR escrevendo 0x0000 AAAA em IWDG_KR.

Assim que o contador for carregado com o novo valor, ele inicia a contagem regressiva e um *reset* é gerado 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 recarreguem a contagem do *watchdog* antes que ele chegue a zero, escrevendo 0x0000 AAAA em IWDG_KR (*feed the dog*).

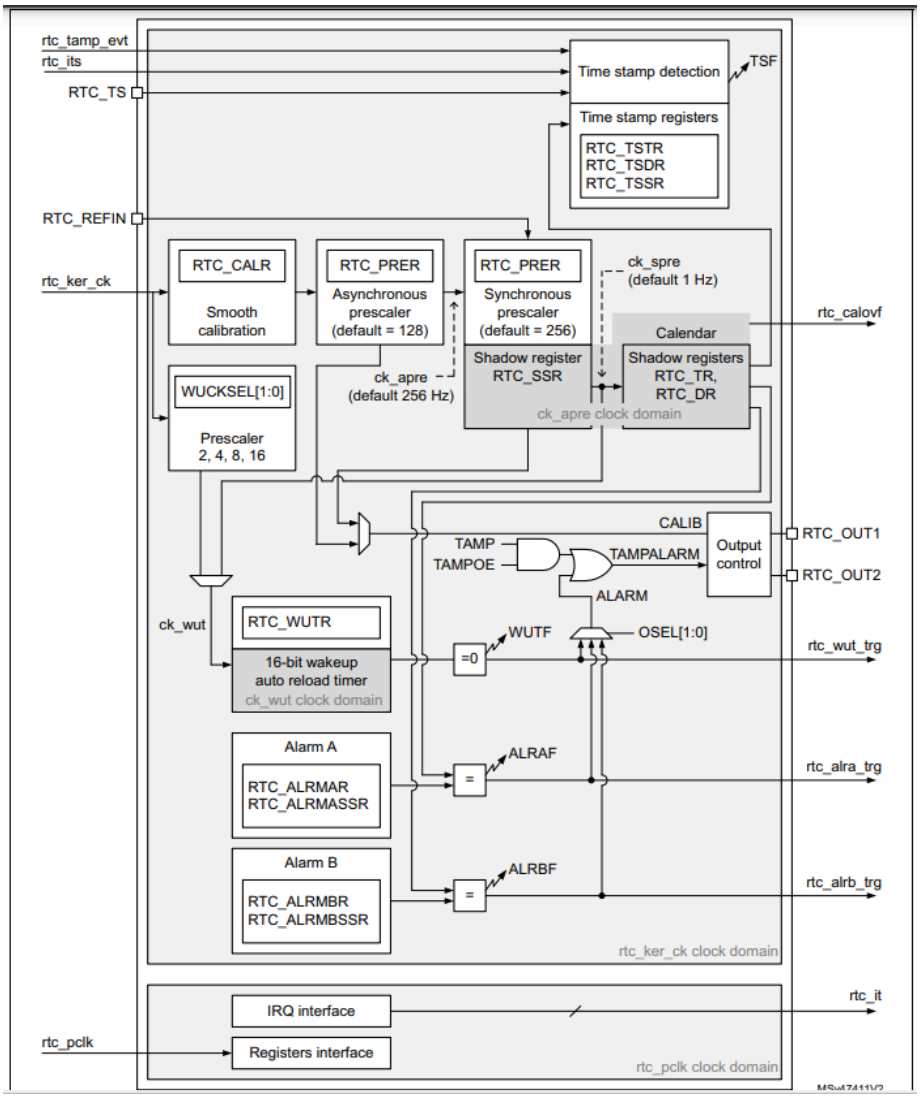
Para operar com uma janela de tempo, deve-se substituir a instrução no item 6 por "carregar o contador com o valor setado no IWDG_RLR, escrevendo o tamanho da janela no registrador IWDG_WINR". Assim, o sistema é reiniciado não apenas se o *watchdog* não receber o *feed*, mas também se receber o *feed* 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*.

Real Time Clock (RTC)

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.

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. No entanto, 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”.

Tipicamente, usa-se o oscilador LSE de 32,768 kHz, como a fonte de *clock*. 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.



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). 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 da frequência 32,768 kHz gerada por LSE. 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.

Assim como o IWGD que possui um registrador de chave que impede alterações não autorizadas em sua configuração, o RTC protege seus registradores críticos com o registrador de chave RTC_WPR. Para “desbloquear” os acessos, deve-se escrever 0xCA seguido de 0x53 nesse registrador. Além disso, as alterações nos registradores de configuração do RTC são apenas permitidas no modo de inicialização. Durante esse modo, o *bit* RTC_ICSR_INIFT deve estar em “1”. 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 os *bits* RTC_CR_ALRAIE e/ou RTC_CR_ALRAIE (IRQ41).

exti_rtc_al	48	41	RTC_ALARM	RTC alarms (A and B) through EXTI Line interrupts	0x0000 00E4

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.

exti_wkup_rtc_wkup	10	3	RTC_WKUP	RTC Wakeup interrupt through the EXTI line	0x0000 004C

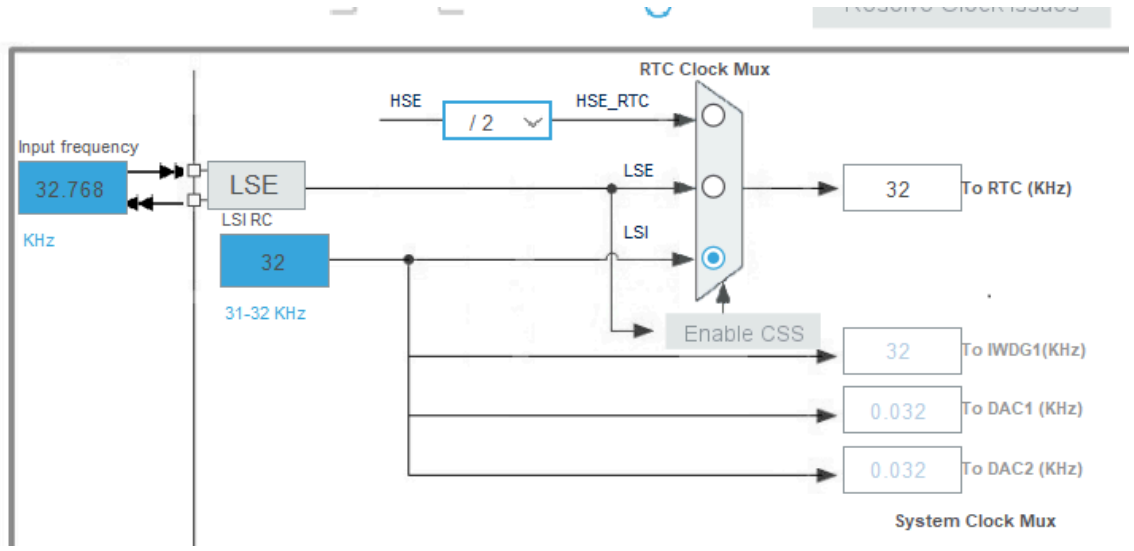
Os alarmes do RTC (*Real-Time Clock*), Alarm A e Alarm B, são fontes do evento EXTI17, que está conectado à linha IRQ41 do NVIC para tratamento como interrupção. No entanto, o EXTI17 também permite que os alarmes acionem periféricos como temporizadores e ADCs, iniciem transferências via DMA ou gerem eventos de software sem ativar a CPU. Da mesma forma, o

Wakeup Timer do RTC usa o evento EXTI19 para gerar interrupções ou ativar o microcontrolador nos modos de baixo consumo, garantindo que o sistema saia do modo *Standby* ou *Stop* apenas quando necessário, reduzindo o consumo de energia. O uso do EXTI como intermediário entre RTC e o NVIC melhora a modularidade, reduz o consumo e otimiza a distribuição de eventos.

Table 126. EXTI Event input mapping

Event input	Source	Event input type	Wakeup target(s)	Connection to NVIC
0 - 15	EXTI[15:0]	Configurable	Any	Yes
16	PVD and AVD(1)	Configurable	CPU only	Yes
17	RTC alarms	Configurable	CPU only	Yes
18	RTC tamper, RTC timestamp, RCC LSECSS(2)	Configurable	CPU only	Yes
19	RTC wakeup timer	Configurable	Any	Yes

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.



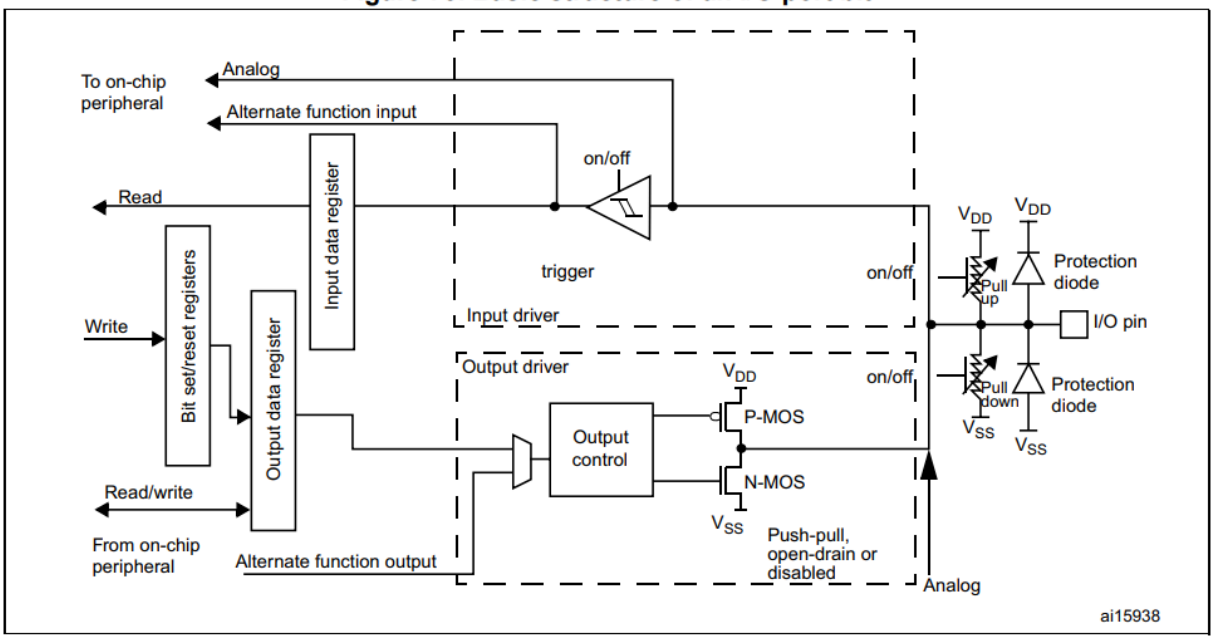
O RTC integrado no STM32H7A3Z oferece várias funcionalidades adicionais que ampliam suas capacidades além da simples contagem de tempo. Entre essas funcionalidades, o RTC inclui um mecanismo de calibração para ajustar sua precisão, garantindo uma medição de tempo mais exata ao longo do tempo, mesmo com 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.

GPIO: Pinos Multiplexáveis

O microcontrolador STM32H7A3 possui pinos de propósito geral (em inglês, *General Purpose Input/Output* – GPIO) que podem ser configurados para diferentes funções, tornando-os **multiplexáveis**. Cada pino tem uma **função primária**, geralmente como entrada ou saída digital, e pode suportar até **16 funções alternativas**, permitindo sua utilização por diversos periféricos internos. Essas funções podem ser **digitais (entrada/saída)** ou **analógicas**, dependendo da configuração.

Os pinos do microcontrolador são **bidirecionais** quando configurados como `GPIO` e incluem circuitos internos que permitem ajustes como **resistores de *pull-up* ou** pull-down e **controle do modo de saída** (*push-pull* ou *open-drain*). No entanto, alguns pinos podem ter direção fixa dependendo da função alternativa utilizada. A configuração dos pinos é feita pelo módulo `GPIOx`, onde "x" representa a porta correspondente (A, B, C, etc.). Esse módulo permite selecionar o **modo de operação** de cada pino, definindo se ele funcionará como `GPIO` comum ou estará vinculado a um periférico por meio dos registradores de função alternativa (em inglês, *Alternate Function Registers* – AFR). Dessa forma, o sistema pode otimizar o uso dos pinos e integrar diferentes periféricos de maneira eficiente, garantindo maior flexibilidade na implementação de projetos embarcados.

Figure 73. Basic structure of an I/O port bit



Após a energização ou *reset*, os pinos iniciam no modo analógico. O registrador `GPIOx_MODER` configura a função de um pino `p` através dos *bits* [2p+1:2p], que definem quatro modos de operação: 0b00 (entrada digital), 0b01 (saída digital de propósito geral), 0b10 (função digital alternativa) e 0b11 (modo analógico). Portanto, para habilitar uma função alternativa em um pino, basta configurar seus *bits* localizados em GPIOx_MODER para 0b10.

11.4.1 GPIO port mode register (GPIOx_MODER) (x =A to K)

Address offset:0x00

Reset value: 0xABFF FFFF for port A

Reset value: 0xFFFF FEBF for port B

Reset value: 0xFFFF FFFF for other ports

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

Existem dois registradores *Alternate Function Register* (AFR) em cada módulo GPIO: `GPIOx_AFRL` (*Low*, configurando pinos 0 a 7) e `GPIOx_AFRH` (*High*, configurando pinos 8 a 15). Os quatro *bits* AFRx[3:0] correspondentes a um pino específico definem a função alternativa do pino, pois a função alternativa é identificada por um código binário que vai de 0b0000 a 0b1111, correspondendo a um total de 16 funções possíveis, numeradas de AF0 a AF15. Note que, apesar de existirem 16 possibilidades para cada pino, nem todos os pinos tem 16 funções alternativas associadas.

11.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A to K)

Address offset: 0x20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFR7[3:0]				AFR6[3:0]				AFR5[3:0]				AFR4[3:0]			
rw	rw	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
AFR3[3:0]				AFR2[3:0]				AFR1[3:0]				AFR0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

11.4.10 GPIO alternate function high register (GPIOx_AFRH) (x = A to J)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFR15[3:0]				AFR14[3:0]				AFR13[3:0]				AFR12[3:0]			
rw	rw	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
AFR11[3:0]				AFR10[3:0]				AFR9[3:0]				AFR8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

As funções alternativas de cada pino são detalhadas no *Datasheet* do microcontrolador, conforme a seguinte informação fornecida no Manual de Referência.

Each I/O pin has a multiplexer with up to sixteen alternate function inputs (AF0 to AF15) that can be configured through the GPIOx_AFRL (for pin 0 to 7) and GPIOx_AFRH (for pin 8 to 15) registers:

- After reset the multiplexer selection is alternate function 0 (AF0). The I/Os are configured in alternate function mode through GPIOx_MODER register.
- The specific alternate function assignments for each pin are detailed in the device datasheet.
- Cortex-M7 with FPU EVENTOUT is mapped on AF15

As tabelas de funções alternativas para cada pino de cada porta são inclusas no *Datasheet* do microcontrolador. Estas tabelas especificam as funções alternativas disponíveis para os pinos das portas PA, PB, PC, PD, PE, PF, PG, PH, PI, PJ, PK, fornecendo uma visão clara das opções de configuração para cada pino. Para associar um pino específico a uma função alternativa, deve-se consultar a tabela correspondente à porta (por exemplo, porta PA) e identificar a função alternativa desejada. Cada pino está listado em uma linha, enquanto as funções alternativas são indicadas nas colunas, numeradas de 0 a 15.

Por exemplo, para configurar o pino PA8 para a função alternativa do canal TIM1_CH1 do temporizador TIM1, devemos consultar a tabela da porta A e localizar a função alternativa 1 (AF1). Isso implica configurar o campo `GPIOA_MODER_MODER8` como “01” para selecionar o modo de função alternativa e definir o campo `GPIOA_AFRH_AFR8` como “0001” para atribuir a função TIM1_CH1 ao pino PA8.

Table 8. Port A alternate functions

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	LPTIM1/ TIM1/2/16/17	PDM_SAI1/ TIM3/4/5/12/15	DFSDM1/ LPTIM2/3/ LPUART1/ OCTOSPIM_P1/2/ TIM8	CEC/DCMI/ PSSI/ DFSDM1/2/ I2C1/2/3/4/ LPTIM2/ TIM15/ USART1	CEC/SPI1/ I2S1/SPI2/ I2S2/SPI3/ I2S3/ SPI4/5/ SPI6/I2S6	DFSDM1/2/ I2C4/ OCTOSPIM_P1/ SAI1/SPI3/I2S3/ UART4	SDMMC1/ SPI2/I2S2/ SPI3/I2S3/ SPI6/I2S6/ UART7/ USART1/2/3/6	LPUART1/ SAI2/ SDMMC1/ SPDIFRX1/ SPI6/I2S6/ UART4/5/8	FDCAN1/2/FMC/ LCD/ OCTOSPIM_P1/2/ SDMMC2/ SPDIFRX1/ TIM13/14	CRS/FMC/LCD/ OCTOSPIM_P1/ OTG1_FS/OTG1_HS/ SAI2/SDMMC2/TIM8	DFSDM1/2/ I2C4/LCD/ MDIOS/ OCTOSPIM_P1/ SDMMC2/ SWPMI1/ TIM1/8/ UART7/9/ USART10	FMC/LCD/ MDIOS/ SDMMC1/ TIM1/8	COMP/DCMI/ PSSI/LCD/ TIM1	LCD/UART5	SYS
Port A	PA0	-	TIM2_CH1/ TIM2_ETR	TIM5_CH1	TIM8_ETR	TIM15_BKIN	SPI6_SS/ I2S6_WS	-	USART2_ CTS/ USART2_ NSS	UART4_TX	SDMMC2_CMD	SAI2_SD_B	-	-	-	-	EVENTOUT
	PA1	-	TIM2_CH2	TIM5_CH2	LPTIM3_OUT	TIM15_CH1N	-	-	USART2_ RTS	UART4_RX	OCTOSPIM_ P1_IO3	SAI2_MCK_B	OCTOSPIM_ P1_DQS	-	-	LCD_R2	EVENTOUT
	PA2	-	TIM2_CH3	TIM5_CH3	-	TIM15_CH1	-	DFSDM2_ CKIN1	USART2_ TX	SAI2_SCK_B	-	-	-	MDIOS_MDIO	-	LCD_R1	EVENTOUT
	PA3	-	TIM2_CH4	TIM5_CH4	OCTOSPIM_ P1_CLK	TIM15_CH2	I2S6_MCK	-	USART2_ RX	-	LCD_B2	OTG_HS_ ULPI_D0	-	-	-	LCD_B5	EVENTOUT
	PA4	-	-	TIM5_ETR	-	-	SPI1_SS/ I2S1_WS	SPI3_SS/ I2S3_WS	USART2_ CK	SPI6_SS/ I2S6_WS	-	-	-	-	DCMI_HSYNC/ PSSI_DE	LCD_ VSYNC	EVENTOUT
	PA5	PWR_NDSTOP2	TIM2_CH1/ TIM2_ETR	-	TIM8_CH1N	-	SPI1_SCK/ I2S1_CK	-	-	SPI6_SCK/ I2S6_CK	-	OTG_HS_ ULPI_CK	-	-	PSSI_D14	LCD_R4	EVENTOUT
	PA6	-	TIM1_BKIN	TIM3_CH1	TIM8_BKIN	-	SPI1_MISO/ I2S1_SDI	OCTOSPIM_ P1_IO3	-	SPI6_MISO/ I2S6_SDI	TIM13_CH1	TIM8_BKIN_COMP12	MDIOS_MDC	TIM1_BKIN_ COMP12	DCMI_PIXCLK/ PSSI_PDCK	LCD_G2	EVENTOUT
	PA7	-	TIM1_CH1N	TIM3_CH2	TIM8_CH1N	DFSDM2_ DATIN1	SPI1_MOSI/ I2S1_SDO	-	-	SPI6_MOSI/ I2S6_SDO	TIM14_CH1	OCTOSPIM_P1_IO2	-	FMC_SDNWE	-	LCD_VSYNC	EVENTOUT
	PA8	MCO1	TIM1_CH1	-	TIM8_BKIN2	I2C3_SCL	-	-	USART1_ CK	-	-	OTG_HS_ SOF	UART7_RX	TIM8_BKIN2_ COMP12	LCD_B3	LCD_R6	EVENTOUT
	PA9	-	TIM1_CH2	-	LPUART1_TX	I2C3_SMBA	SPI2_SCK/ I2S2_CK	-	USART1_ TX	-	-	-	-	-	DCMI_D0/ PSSI_D0	LCD_R5	EVENTOUT
	PA10	-	TIM1_CH3	-	LPUART1_RX	-	-	-	USART1_ RX	-	-	OTG_HS_ ID	MDIOS_MDIO	LCD_B4	DCMI_D1/ PSSI_D1	LCD_B1	EVENTOUT
	PA11	-	TIM1_CH4	-	LPUART1_CTS	-	SPI2_SS/ I2S2_WS	UART4_RX	USART1_CTS/ USART1_NSS	-	FDCAN1_ RX	-	-	-	-	LCD_R4	EVENTOUT
	PA12	-	TIM1_ETR	-	LPUART1_RTS	-	SPI2_SCK/ I2S2_CK	UART4_TX	USART1_ RTS	SAI2_FS_B	FDCAN1_ TX	-	-	-	-	LCD_R5	EVENTOUT
	PA13	JTMS/ SWDIO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA14	JTCK/ SWCLK	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA15	JTDI	TIM2_CH1/ TIM2_ETR	-	-	HDMI_CEC	SPI1_SS/ I2S1_WS	SPI3_SS/ I2S3_WS	SPI6_SS/ I2S6_WS	UART4_ RTS	LCD_R3	-	UART7_TX	-	-	LCD_B6	EVENTOUT

Outro exemplo é configurar o pino PB13 para a função alternativa de saida da instância LPTIM2 do temporizador LPTIM. Neste caso, devemos consultar a tabela da porta B e localizar a função alternativa 3 (AF3). Isso implica configurar o campo `GPIOB_MODER_MODER13` como “01” para selecionar o modo de função alternativa e definir o campo `GPIOB_AFRH_AFR13` como “0011” para atribuir a função LPTIM2_OUT ao pino PB13.

Table 9. Port B alternate functions

Port	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
	SYS	LPTIM1/ TIM1/2/16/17	PDM_SAI1/ TIM3/4/5/12/15	DFSDM1/ LPTIM2/3/ LPUART1/ OCTOSPIM_P1/2/ TIM8	CEC/DCMI/PSSI/ DFSDM1/2/ I2C1/2/3/4/ LPTIM2/TIM15/ USART1	CEC/SPI1/ I2S1/SPI2/ I2S2/SPI3/ I2S3/ SPI4/5/ SPI6/I2S6	DFSDM1/2/I2C4/ OCTOSPIM_P1/ SAI1/SPI3/I2S3/ UART4	SDMMC1/SPI2/ I2S2/SPI3/I2S3/ SPI6/I2S6/ UART7/ USART1/2/3/6	LPUART1/ SAI2/ SDMMC1/ SPDIFRX1/ SPI6/I2S6/ UART4/5/8	FDCAN1/2/FMC/LC D/OCTOSPIM_P1/2/ SDMMC2/ SPDIFRX1/TIM13/14	CRS/FMC/LCD/ OCTOSPIM_P1/ OTG1_FS/ OTG1_HS/ SAI2/SDMMC2/ TIM8	DFSDM1/2/ I2C4/LCD/MDIOS/ OCTOSPIM_P1/ SDMMC2/SWPMI1/ TIM1/8/UART7/9/ USART10	FMC/LCD/ MDIOS/ SDMMC1/ TIM1/8	COMP/DCMI/ PSSI/LCD/TIM1	LCD/ UART5	SYS
PB0	-	TIM1_CH2N	TIM3_CH3	TIM8_CH2N	DFSDM2_CKOUT	-	DFSDM1_CKOUT	-	UART4_CTS	LCD_R3	OTG_HS_ ULPI_D1	OCTOSPIM_P1_IO1	-	-	LCD_G1	EVENTOUT
PB1	-	TIM1_CH3N	TIM3_CH4	TIM8_CH3N	-	-	DFSDM1_DATIN1	-	-	LCD_R6	OTG_HS_ ULPI_D2	OCTOSPIM_ P1_IO0	-	-	LCD_G0	EVENTOUT
PB2	RTC_OUT2	-	SAI1_D1	-	DFSDM1_CKIN1	-	SAI1_SD_A	SPI3_MOSI/ I2S3_SDO	-	OCTOSPIM_P1_CLK	OCTOSPIM_ P1_DQS	-	-	-	-	EVENTOUT
PB3	JTDO/ TRACESWO	TIM2_CH2	-	-	-	SPI1_SCK/ I2S1_CK	SPI3_SCK/ I2S3_CK	-	SPI6_SCK/ I2S6_CK	SDMMC2_D2	CRS_SYNC	UART7_RX	-	-	-	EVENTOUT
PB4	NJTRST	TIM16_BKIN	TIM3_CH1	-	-	SPI1_MISO/ I2S1_SDI	SPI3_MISO/ I2S3_SDI	SPI2_SS/ I2S2_WS	SPI6_MISO/ I2S6_SDI	SDMMC2_D3	-	UART7_TX	-	-	-	EVENTOUT
PB5	-	TIM17_BKIN	TIM3_CH2	-	I2C1_SMBA	SPI1_MOSI/ I2S1_SDO	I2C4_SMBA	SPI3_MOSI/ I2S3_SDO	SPI6_MOSI/ I2S6_SDO	FDCAN2_RX	OTG_HS_ ULPI_D7	LCD_B5	FMC_SDCKE1	DCMI_D10/ PSSI_D10	UART5_RX	EVENTOUT
PB6	-	TIM16_CH1N	TIM4_CH1	-	I2C1_SCL	HDMI_CEC	I2C4_SCL	USART1_TX	LPUART1_TX	FDCAN2_TX	OCTOSPIM_ P1_NCS	DFSDM1_DATIN5	FMC_SDNE1	DCMI_D5/PSSI_D5	UART5_TX	EVENTOUT
PB7	-	TIM17_CH1N	TIM4_CH2	-	I2C1_SDA	-	I2C4_SDA	USART1_RX	LPUART1_RX	-	-	DFSDM1_CKIN5	FMC_NL	DCMI_VSYNC/ PSSI_RDY	-	EVENTOUT
PB8	-	TIM16_CH1	TIM4_CH3	DFSDM1_CKIN7	I2C1_SCL	-	I2C4_SCL	SDMMC1_CKIN	UART4_RX	FDCAN1_RX	SDMMC2_D4	-	SDMMC1_D4	DCMI_D6/PSSI_D6	LCD_B6	EVENTOUT
PB9	-	TIM17_CH1	TIM4_CH4	DFSDM1_DATIN7	I2C1_SDA	SPI2_SS/ I2S2_WS	I2C4_SDA	SDMMC1_CDIR	UART4_TX	FDCAN1_TX	SDMMC2_D5	I2C4_SMBA	SDMMC1_D5	DCMI_D7/PSSI_D7	LCD_B7	EVENTOUT
PB10	-	TIM2_CH3	-	LPTIM2_IN1	I2C2_SCL	SPI2_SCK/ I2S2_CK	DFSDM1_DATIN7	USART3_TX	-	OCTOSPIM_ P1_NCS	OTG_HS_ ULPI_D3	-	-	-	LCD_G4	EVENTOUT
PB11	-	TIM2_CH4	-	LPTIM2_ETR	I2C2_SDA	-	DFSDM1_CKIN7	USART3_RX	-	-	OTG_HS_ ULPI_D4	-	-	-	LCD_G5	EVENTOUT
PB12	-	TIM1_BKIN	-	OCTOSPIM_ P1_NCLK	I2C2_SMBA	SPI2_SS/ I2S2_WS	DFSDM1_DATIN1	USART3_CK	-	FDCAN2_RX	OTG_HS_ ULPI_D5	DFSDM2_DATIN1	-	TIM1_BKIN_COMP12	UART5_RX	EVENTOUT
PB13	-	TIM1_CH1N	-	LPTIM2_OUT	DFSDM2_CKIN1	SPI2_SCK/ I2S2_CK	DFSDM1_CKIN1	USART3_CTS/ USART3_NSS	-	FDCAN2_TX	OTG_HS_ ULPI_D6	-	SDMMC1_D0	DCMI_D2/PSSI_D2	UART5_TX	EVENTOUT
PB14	-	TIM1_CH2N	TIM12_CH1	TIM8_CH2N	USART1_TX	SPI2_MISO/ I2S2_SDI	DFSDM1_DATIN2	USART3_RTS	UART4_RTS	SDMMC2_D0	-	-	-	-	LCD_CLK	EVENTOUT
PB15	RTC_REFIN	TIM1_CH3N	TIM12_CH2	TIM8_CH3N	USART1_RX	SPI2_MOSI/ I2S2_SDO	DFSDM1_CKIN2	-	UART4_CTS	SDMMC2_D1	-	-	-	-	LCD_G7	EVENTOUT

Seguem-se as tabelas de definição de funções alternativas para os pinos PC, PD e PE.

Table 10. Port C alternate functions

Port	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
	SYS	LPTIM1/ TIM1/2/16/17	PDM_SAI1/ TIM3/4/5/12/15	DFSDM1/ LPTIM2/3/ LPUART1/ OCTOSPIM_P1/2/ TIM8	CEC/DCMI/PSSI/ DFSDM1/2/ I2C1/2/3/4/ LPTIM2/TIM15/ USART1	CEC/SPI1/ I2S1/SPI2/ I2S2/SPI3/ I2S3/ SPI4/5/ SPI6/I2S6	DFSDM1/2/I2C4/ OCTOSPIM_P1/ SAI1/SPI3/I2S3/ UART4	SDMMC1/ SPI2/I2S2/ SPI3/I2S3/ SPI6/I2S6/ UART7/ USART1/2/3/6	LPUART1/ SAI2/SDMMC1/ SPDIFRX1/SPI6/ I2S6/UART4/5/8	FDCAN1/2/FMC/LCD /OCTOSPIM_P1/2/ SDMMC2/ SPDIFRX1/TIM13/14	CRS/FMC/LCD/ OCTOSPIM_P1/ OTG1_FS/ OTG1_HS/ SAI2/SDMMC2/ TIM8	DFSDM1/2/ I2C4/LCD/ MDIOS/ OCTOSPIM_P1/ SDMMC2/ SWPMI1/ TIM1/8/ UART7/9/ USART10	FMC/LCD/ MDIOS/ SDMMC1/ TIM1/8	COMP/DCMI/ PSSI/LCD/ TIM1	LCD/UART5	SYS
PC0	-	-	-	DFSDM1_CKIN0	-	-	DFSDM1_DATIN4	-	SAI2_FS_B	FMC_A25	OTG_HS_ ULPI_STP	LCD_G2	FMC_SDNWE	-	LCD_R5	EVENTOUT
PC1	TRACED0	-	SAI1_D1	DFSDM1_DATIN0	DFSDM1_CKIN4	SPI2_MOSI/ I2S2_SDO	SAI1_SD_A	-	-	SDMMC2_CK	OCTOSPIM_ P1_IO4	-	MDIOS_MDC	-	LCD_G5	EVENTOUT
PC2	PWR_CSTOP	-	-	DFSDM1_CKIN1	-	SPI2_MISO/ I2S2_SDI	DFSDM1_CKOUT	-	-	OCTOSPIM_P1_IO2	OTG_HS_ ULPI_DIR	OCTOSPIM_ P1_IO5	FMC_SDNE0	-	-	EVENTOUT
PC3	PWR_CSLEEP	-	-	DFSDM1_DATIN1	-	SPI2_MOSI/ I2S2_SDO	-	-	-	OCTOSPIM_P1_IO0	OTG_HS_ ULPI_NXT	OCTOSPIM_ P1_IO6	FMC_SDCKE0	-	-	EVENTOUT
PC4	-	-	-	DFSDM1_CKIN2	-	I2S1_MCK	-	-	-	SPDIFRX1_IN2	-	-	FMC_SDNE0	-	LCD_R7	EVENTOUT
PC5	-	-	SAI1_D3	DFSDM1_DATIN2	PSSI_D15	-	-	-	-	SPDIFRX1_IN3	OCTOSPIM_ P1_DQS	-	FMC_SDCKE0	COMP1_OUT	LCD_DE	EVENTOUT
PC6	-	-	TIM3_CH1	TIM8_CH1	DFSDM1_CKIN3	I2S2_MCK	-	USART6_TX	SDMMC1_D0DIR	FMC_NWAIT	SDMMC2_D6	-	SDMMC1_D6	DCMI_D0/ PSSI_D0	LCD_HSYNC	EVENTOUT
PC7	TRGIO	-	TIM3_CH2	TIM8_CH2	DFSDM1_DATIN3	-	I2S3_MCK	USART6_RX	SDMMC1_D123DIR	FMC_NE1	SDMMC2_D7	SWPMI_TX	SDMMC1_D7	DCMI_D1/ PSSI_D1	LCD_G6	EVENTOUT
PC8	TRACED1	-	TIM3_CH3	TIM8_CH3	-	-	-	USART6_CK	UART5_RTS	FMC_NE2/ FMC_NCE	FMC_INT	SWPMI_RX	SDMMC1_D0	DCMI_D2/ PSSI_D2	-	EVENTOUT
PC9	MCO2	-	TIM3_CH4	TIM8_CH4	I2C3_SDA	I2S_CKIN	-	-	UART5_CTS	OCTOSPIM_P1_IO0	LCD_G3	SWPMI_ SUSPEND	SDMMC1_D1	DCMI_D3/ PSSI_D3	LCD_B2	EVENTOUT
PC10	-	-	-	DFSDM1_CKIN5	DFSDM2_CKIN0	-	SPI3_SCK/ I2S3_CK	USART3_TX	UART4_TX	OCTOSPIM_P1_IO1	LCD_B1	SWPMI_RX	SDMMC1_D2	DCMI_D8/ PSSI_D8	LCD_R2	EVENTOUT
PC11	-	-	-	DFSDM1_DATIN5	DFSDM2_DATIN0	-	SPI3_MISO/ I2S3_SDI	USART3_RX	UART4_RX	OCTOSPIM_P1_NCS	-	-	SDMMC1_D3	DCMI_D4/ PSSI_D4	LCD_B4	EVENTOUT
PC12	TRACED3	-	TIM15_CH1	-	DFSDM2_CKOUT	SPI6_SCK/ I2S6_CK	SPI3_MOSI/ I2S3_SDO	USART3_CK	UART5_TX	-	-	-	SDMMC1_CK	DCMI_D9/ PSSI_D9	LCD_R6	EVENTOUT
PC13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
PC14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
PC15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT

Table 11. Port D alternate functions

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	LPTIM1/ TIM1/2/16/17	PDM_SAI1/ TIM3/4/5/12/15	DFSDM1/ LPTIM2/3/ LPUART1/ OCTOSPIM_P1/2/ TIM8	CEC/DCMI/PSSI/ DFSDM1/2/ I2C1/2/3/4/ LPTIM2/TIM15/ USART1	CEC/SPI1/ I2S1/SPI2/ I2S2/SPI3/ I2S3/ SPI4/5/ SPI6/I2S6	DFSDM1/2/I2C4/ OCTOSPIM_P1/ SAI1/SPI3/I2S3/ UART4	SDMMC1/ SPI2/I2S2/ SPI3/I2S3/ SPI6/I2S6/ UART7/ USART1/2/3/6	LPUART1/ SAI2/ SDMMC1/ SPDIFRX1/ SPI6/I2S6/ UART4/5/8	FDCAN1/2/FMC/LCD/OCTOSPIM_P1/2/ SDMMC2/ SPDIFRX1/ TIM13/14	CRS/FMC/LCD/ OCTOSPIM_P1/ OTG1_FS/ OTG1_HS/SAI2/ SDMMC2/TIM8	DFSDM1/2/ I2C4/LCD/ MDIOS/ OCTOSPIM_P1/ SDMMC2/ SWPMI1/ TIM1/8/ UART7/9/ USART10	FMC/LCD/ MDIOS/ SDMMC1/ TIM1/8	COMP/ DCMI/ PSSI/LCD/ TIM1	LCD/UART5	SYS
Port D	PD0	-	-	-	DFSDM1_CKIN6	-	-	-	-	UART4_RX	FDCAN1_RX	-	UART9_CTS	FMC_D2/ FMC_DA2	-	LCD_B1	EVENTOUT
	PD1	-	-	-	DFSDM1_DATIN6	-	-	-	-	UART4_TX	FDCAN1_TX	-	-	FMC_D3/ FMC_DA3	-	-	EVENTOUT
	PD2	TRACED2	-	TIM3_ETR	-	TIM15_BKIN	-	-	-	UART5_RX	LCD_B7	-	-	SDMMC1_CMD	DCMI_D11/ PSSI_D11	LCD_B2	EVENTOUT
	PD3	-	-	-	DFSDM1_CKOUT	-	SPI2_SCK/ I2S2_CK	-	USART2_CTS/ USART2_NSS	-	-	-	-	FMC_CLK	DCMI_D5/ PSSI_D5	LCD_G7	EVENTOUT
	PD4	-	-	-	-	-	-	-	USART2_RTS	-	-	OCTOSPIM_P1_IO4	-	FMC_NOE	-	-	EVENTOUT
	PD5	-	-	-	-	-	-	-	USART2_TX	-	-	OCTOSPIM_P1_IO5	-	FMC_NWE	-	-	EVENTOUT
	PD6	-	-	SAI1_D1	DFSDM1_CKIN4	DFSDM1_DATIN1	SPI3_MOSI/ I2S3_SDO	SAI1_SD_A	USART2_RX	-	-	OCTOSPIM_P1_IO6	SDMMC2_CK	FMC_NWAIT	DCMI_D10/ PSSI_D10	LCD_B2	EVENTOUT
	PD7	-	-	-	DFSDM1_DATIN4	-	SPI1_MOSI/ I2S1_SDO	DFSDM1_CKIN1	USART2_CK	-	SPDIFRX1_IN0	OCTOSPIM_P1_IO7	SDMMC2_CMD	FMC_NE1	-	-	EVENTOUT
	PD8	-	-	-	DFSDM1_CKIN3	-	-	-	USART3_TX	-	SPDIFRX1_IN1	-	-	FMC_D13/ FMC_DA13	-	-	EVENTOUT
	PD9	-	-	-	DFSDM1_DATIN3	-	-	-	USART3_RX	-	-	-	-	FMC_D14/ FMC_DA14	-	-	EVENTOUT
	PD10	-	-	-	DFSDM1_CKOUT	DFSDM2_CKOUT	-	-	USART3_CK	-	-	-	-	FMC_D15/ FMC_DA15	-	LCD_B3	EVENTOUT
	PD11	-	-	-	LPTIM2_IN2	I2C4_SMBA	-	-	USART3_CTS/ USART3_NSS	-	OCTOSPIM_P1_IO0	SAI2_SD_A	-	FMC_A16/ FMC_CLE	-	-	EVENTOUT
	PD12	-	LPTIM1_IN1	TIM4_CH1	LPTIM2_IN1	I2C4_SCL	-	-	USART3_RTS	-	OCTOSPIM_P1_IO1	SAI2_FS_A	-	FMC_A17/ FMC_ALE	DCMI_D12/ PSSI_D12	-	EVENTOUT
	PD13	-	LPTIM1_OUT	TIM4_CH2	-	I2C4_SDA	-	-	-	-	OCTOSPIM_P1_IO3	SAI2_SCK_A	UART9_RTS	FMC_A18	DCMI_D13/ PSSI_D13	-	EVENTOUT
	PD14	-	-	TIM4_CH3	-	-	-	-	-	UART8_CTS	-	-	UART9_RX	FMC_D0/ FMC_DA0	-	-	EVENTOUT
	PD15	-	-	TIM4_CH4	-	-	-	-	-	UART8_RTS	-	-	UART9_TX	FMC_D1/ FMC_DA1	-	-	EVENTOUT

Table 12. Port E alternate functions

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	LPTIM1/ TIM1/2/16/17	PDM_SAI1/ TIM3/4/5/12/15	DFSDM1/ LPTIM2/3/ LPUART1/ OCTOSPIM_P1/2/ TIM8	CEC/DCMI/ PSSI/ DFSDM1/2/ I2C1/2/3/4/ LPTIM2/ TIM15/ USART1	CEC/SPI1/ I2S1/SPI2/ I2S2/SPI3/ I2S3/ SPI4/5/ SPI6/I2S6	DFSDM1/2/ I2C4/ OCTOSPIM_P1/ SAI1/SPI3/I2S3/ UART4	SDMMC1/ SPI2/I2S2/ SPI3/I2S3/ SPI6/I2S6/ UART7/ USART1/2/3/6	LPUART1/ SAI2/ SDMMC1/ SPDIFRX1/ SPI6/I2S6/ UART4/5/8	FDCAN1/2/FMC/LCD/OCTOSPIM_P1/2/ SDMMC2/ SPDIFRX1/ TIM13/14	CRS/FMC/LCD/ OCTOSPIM_P1/ OTG1_FS/ OTG1_HS/SAI2/ SDMMC2/TIM8	DFSDM1/2/ I2C4/LCD/MDIOS/ OCTOSPIM_P1/ SDMMC2/SWPMI1/ TIM1/8/UART7/9/ USART10	FMC/LCD/ MDIOS/ SDMMC1/ TIM1/8	COMP/DCMI/ PSSI/LCD/TIM1	LCD/ UART5	SYS
Port E	PE0	-	LPTIM1_ETR	TIM4_ETR	-	LPTIM2_ETR	-	-	-	UART8_Rx	-	SAI2_MCK_A	-	FMC_NBL0	DCMI_D2/ PSSI_D2	LCD_R0	EVENTOUT
	PE1	-	LPTIM1_IN2	-	-	-	-	-	-	UART8_Tx	-	-	-	FMC_NBL1	DCMI_D3/ PSSI_D3	LCD_R6	EVENTOUT
	PE2	TRACECLK	-	SAI1_CK1	-	-	SPI4_SCK	SAI1_MCLK_A	-	-	OCTOSPIM_P1_IO2	-	USART10_RX	FMC_A23	-	-	EVENTOUT
	PE3	TRACED0	-	-	-	TIM15_BKIN	-	SAI1_SD_B	-	-	-	-	USART10_TX	FMC_A19	-	-	EVENTOUT
	PE4	TRACED1	-	SAI1_D2	DFSDM1_DATIN3	TIM15_CH1N	SPI4_SS	SAI1_FS_A	-	-	-	-	-	FMC_A20	DCMI_D4/ PSSI_D4	LCD_B0	EVENTOUT
	PE5	TRACED2	-	SAI1_CK2	DFSDM1_CKIN3	TIM15_CH1	SPI4_MISO	SAI1_SCK_A	-	-	-	-	-	FMC_A21	DCMI_D6/ PSSI_D6	LCD_G0	EVENTOUT
	PE6	TRACED3	TIM1_BKIN2	SAI1_D1	-	TIM15_CH2	SPI4_MOSI	SAI1_SD_A	-	-	-	SAI2_MCK_B	TIM1_BKIN2_ COMP12	FMC_A22	DCMI_D7/ PSSI_D7	LCD_G1	EVENTOUT
	PE7	-	TIM1_ETR	-	DFSDM1_DATIN2	-	-	-	UART7_RX	-	-	OCTOSPIM_P1_IO4	-	FMC_D4/ FMC_DA4	-	-	EVENTOUT
	PE8	-	TIM1_CH1N	-	DFSDM1_CKIN2	-	-	-	UART7_TX	-	-	OCTOSPIM_P1_IO5	-	FMC_D5/ FMC_DA5	COMP2_OUT	-	EVENTOUT
	PE9	-	TIM1_CH1	-	DFSDM1_CKOUT	-	-	-	UART7_RTS	-	-	OCTOSPIM_P1_IO6	-	FMC_D6/ FMC_DA6	-	-	EVENTOUT
	PE10	-	TIM1_CH2N	-	DFSDM1_DATIN4	-	-	-	UART7_CTS	-	-	OCTOSPIM_P1_IO7	-	FMC_D7/ FMC_DA7	-	-	EVENTOUT
	PE11	-	TIM1_CH2	-	DFSDM1_CKIN4	-	SPI4_SS	-	-	-	-	SAI2_SD_B	OCTOSPIM_P1_NCS	FMC_D8/ FMC_DA8	-	LCD_G3	EVENTOUT
	PE12	-	TIM1_CH3N	-	DFSDM1_DATIN5	-	SPI4_SCK	-	-	-	-	SAI2_SCK_B	-	FMC_D9/ FMC_DA9	COMP1_ OUT	LCD_B4	EVENTOUT
	PE13	-	TIM1_CH3	-	DFSDM1_CKIN5	-	SPI4_MISO	-	-	-	-	SAI2_FS_B	-	FMC_D10/ FMC_DA10	COMP2_OUT	LCD_DE	EVENTOUT
	PE14	-	TIM1_CH4	-		-	SPI4_MOSI	-	-	-	-	SAI2_MCK_B	-	FMC_D11/ FMC_DA11	-	LCD_CLK	EVENTOUT
	PE15	-	TIM1_BKIN	-		-	-	-	-	-	-	-	USART10_CK	FMC_D12/ FMC_DA12	TIM1_BKIN_COMP12	LCD_R7	EVENTOUT

Além das funções primárias e alternativas gerenciadas pelos módulos GPIOx, os pinos podem assumir **funções adicionais**, configuradas diretamente pelos registradores dos periféricos associados. Esse mecanismo simplifica o processo de configuração e gerenciamento dos pinos, pois a atribuição da função ocorre automaticamente quando o periférico correspondente é habilitado e o pino é alocado para seu uso. O *Datasheet* do microcontrolador fornece informações detalhadas

sobre essas funções adicionais, incluindo tabelas que descrevem as funções alternativas e adicionais para cada pino, considerando os diferentes encapsulamentos disponíveis.

Os periféricos GPIOx oferecem apenas funções de multiplexação e operações básicas de entrada e saída digital para pinos configurados no modo de **função primária**. Conforme visto no Roteiro 3, para que um pino GPIO configurado como entrada digital possa reagir de maneira eficiente a eventos externos por meio de interrupções e sinais de evento, ele deve ser associado a um evento EXTI correspondente no registrador SYSCFG_EXTICRn. Dessa forma, o EXTI adiciona uma camada extra de funcionalidade e flexibilidade, estendendo as capacidades convencionais dos GPIOs. No entanto, para utilizar essas funcionalidades avançadas, são necessárias configurações adicionais, conforme detalhado no Roteiro 3.

Já no caso de pinos configurados no modo de função alternativa, o tratamento de eventos externos é realizado diretamente pelo periférico associado, que gerencia a captura e processamento dos sinais sem depender do EXTI na maioria dos casos. No entanto, o desenvolvedor ainda pode precisar configurar e tratar interrupções dentro do periférico para lidar com eventos específicos, como captura de borda em temporizadores ou recepção de dados em interfaces seriais..

Pin/ball name[1] [2]															Pin name (function after reset)	Pin type	I/O structure	Alternate functions	Additional functions
LQFP100 with SMPS	TFBGA100 with SMPS	LQFP144 with SMPS	WLCSP132 with SMPS	UFBGA169 with SMPS	UFBGA176+25 with SMPS	LQFP176 with SMPS	TFBGA225 with SMPS	LQFP64	TFBGA100	LQFP100	LQFP144	UFBGA176+25	LQFP176	TFBGA216					
-	-	-	-	-	-	-	C1	-	-	-	-	D2	7	C2	PI8	I/O	FT	EVENTOUT	TAMP_IN2/ TAMP_OUT3, RTC_OUT2, WKUP4
6	E4	9	E10	D3	C1	9	F4	2	A2	7	7	D1	8	D1	PC13	I/O	FT	EVENTOUT	TAMP_IN1/ TAMP_OUT2/ TAMP_OUT3, RTC_OUT1/ RTC_TS, WKUP3

Há um mecanismo de proteção nos GPIOs do microcontrolador STM32H7A3. É possível “congelar" a configuração de alguns registradores de controle dos GPIOs, impedindo modificações acidentais ou não autorizadas pelo registrador:

- **GPIOx_LCKR (Lock Control Register):** Este é o registrador de chave do GPIO. Escrevendo uma sequência específica de *bits* nele, é possível travar (congelar) outros registradores.

Os registradores listados abaixo são afetados pelo travamento. Eles não poderão mais ser alterados até que o sistema seja reiniciado ou uma outra ação específica seja realizada (normalmente um *reset*). As configurações atuais desses registradores são mantidas.

- **GPIOx_MODER (Mode Register):** Define o modo do pino (entrada, saída, função alternativa, analógico).
- **GPIOx_OTYPER (Output Type Register):** Define o tipo de saída (push-pull ou open-drain).
- **GPIOx_OSPEEDR (Output Speed Register):** Define a velocidade de comutação da saída.
- **GPIOx_PUPDR (Pull-up/Pull-down Register):** Define se há resistores pull-up ou pull-down internos habilitados nos pinos.

- **GPIOx_AFRL (Alternate Function Low Register):** Define as funções alternativas para os pinos (parte baixa).
- **GPIOx_AFRH (Alternate Function High Register):** Define as funções alternativas para os pinos (parte alta).