DISCIPLINA EA701

Introdução aos Sistemas Embarcados

ROTEIRO 3: Varredura (*Polling*) e Interrupção

VARREDURA, PROCESSAMENTO DE INTERRUPÇÕES, PILHAS, CONTROLADOR DE INTERRUPÇÕES VETORIZADAS ANINHADAS (NVIC), CONTROLADOR DE INTERRUPÇÕES EXTERNAS (EXTI), CONTROLADOR DE CONFIGURAÇÃO DO SISTEMA (SYSCFG).

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

FEEC / UNICAMP

Revisado em agosto de 2024

INTRODUÇÃO

A interface mais simples de um microcontrolador com o mundo exterior é através de sinais digitais básicos (níveis alto e baixo) em seus pinos, seja com o microcontrolador produzindo estes sinais (Saídas) ou lendo os sinais digitais provindos de outros elementos do sistema digital (Entradas), sempre através de seus pinos físicos. O módulo interno que gerencia esta comunicação digital básica é denominado **GPIO** (do inglês *General Purpose Input/Output*, ou Entrada/Saída de Uso Geral). Nos primeiros testes, configuramos o pino PB0 como uma saída digital, ativando e configurando o módulo GPIOB correspondente. Controlamos o estado do pino utilizando o registrador de controle GPIOB_BSRR para definir os níveis lógicos ou o registrador de dados GPIOB_ODR para escrever o estado do pino.

Neste roteiro, expandiremos o uso do GPIO para incluir entradas digitais e exploraremos dois métodos amplamente utilizados para monitorar o estado de sinais ou módulos externos ao núcleo do processador: ***polling*** e **interrupção**. O *polling* envolve a verificação contínua de um evento de interesse por meio de comandos, para determinar se o fluxo de execução deve ser interrompido. Por outro lado, a interrupção permite que o processador desvie automaticamente o fluxo de execução assim que um evento relevante ocorre. Além disso, abordaremos a programação dos periféricos do microcontrolador para implementar ambos os

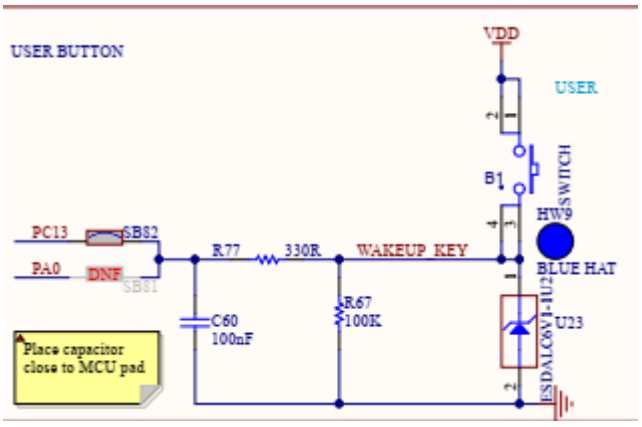
métodos. Também discutiremos a importância da estrutura de dados chamada **pilha** no gerenciamento do tratamento de interrupções.

PROJETOS-EXEMPLO

Nos dois primeiros Roteiros, exploramos como configurar microcontroladores para enviar sinais a dispositivos externos, com foco específico no controle de LEDs. Para que um microcontrolador seja realmente útil, ele precisa interagir com o mundo externo através de seus pinos, tanto recebendo quanto respondendo a sinais gerados por dispositivos externos. Ao longo deste curso, veremos que os microcontroladores possuem pinos com as mais diversas funções, permitindo **comunicação bidirecional** com o ambiente externo. Neste Roteiro, daremos início às interações digitais básicas, abordando a entrada de sinais digitais gerados por um botão de acionamento. Um dos principais desafios ao lidar com esses sinais é sua **natureza assíncrona** em relação ao sinal de relógio do microcontrolador, além da imprevisibilidade de quando esses sinais ocorrem. Para superar esse desafio, apresentamos duas soluções práticas amplamente utilizadas através de dois projetos-exemplo.

Um projeto de GPIO com *Polling*

Inicialmente, vamos criar um projeto semelhante ao pisca com CMSIS visto no Roteiro 2, mas adicionando uma entrada digital. Vamos usar o pino PC13 como entrada, pois ele está ligado ao botão azul na placa NUCLEO, como se pode ver no trecho do esquemático oficial da placa, reproduzido abaixo:



À esquerda, existem *jumpers* que determinam se o botão é ligado no pino PC13 ou no PA0, sendo que o padrão da placa é o PC13. O resistor de 100k mantém o pino PC13 em nível lógico baixo, até que o botão seja pressionado. O resistor de 330R e o capacitor de 100nF evitam o fenômeno de *bounce*, ou "repique", que faz com que o repique nos contatos mecânicos do botão seja percebido pela entrada digital como múltiplas transições entre os níveis alto e baixo. Este fenômeno será melhor estudado quando trabalharmos com entrada de

teclado matricial.. Assim, ao se pressionar o botão, o pino PC13 vai ao nível lógico alto, e ao se soltar o botão o pino retorna ao nível lógico baixo.

1. Vamos escrever um programa que troque o estado do LED verde quando o botão é pressionado. Inicialmente, crie o projeto "Toggle_Polling" da mesma forma que o projeto com CMSIS do Roteiro 2. Lembre-se de selecionar "Empty" na opção "Targeted Project Type" e de adicionar as pastas com os arquivos de suporte, além de colocar o "include" para o arquivo de cabeçalho. Adicione os caminhos que contêm os arquivos de cabeçalho "stm32h7a3xxq.h" e "core_cm7.h"

2. Vamos repetir a configuração do pino PB0 para o LED verde, e vamos incluir a configuração do PC13 como entrada. No Manual de Referência, vemos que o *bit* 2 do registrador RCC_AHB4ENR habilita o GPIOC. Assim, podemos setar (colocar em "1") os *bits* 1 e 2 deste registrador simultaneamente para ativar GPIOB e GPIOC usando códigos binários:
RCC_AHB4ENR |= 0x00000006;
ou usando a interface CMSIS:
RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk |
RCC_AHB4ENR_GPIOCEN_Msk;

Na sequência, os registradores de GPIOB são configurados como no projeto do Roteiro 2, para que se possa ligar e desligar o LED verde. Para a entrada do *push-button*, vamos configurar o pino 13 de GPIOC como entrada digital. Os registradores de GPIOC são idênticos aos de GPIOB. Assim, só é necessário configurar o registrador GPIOC_MODER para que o pino 13 funcione como entrada. No Manual de Referência, pode-se ver que os dois *bits* de configuração de um pino precisam estar em "0" para que o pino seja configurado como entrada. Para o pino 13, os *bits* correspondentes são o 27 e o 26. Assim, a instrução de configuração fica:
GPIOC_MODER &= 0xF3FFFFFF;
Se usarmos a interface CMSIS, basta saber o número do pino, que é 13
GPIOC->MODER &= ~(GPIO_MODER_MODE13_Msk);

Por fim, para ler o nível lógico de um pino de entrada, basta ler o registrador GPIOC_IDR (*Input Data Register*). O *bit* 13 será "0" quando o pino PC13 está em nível lógico baixo, e "1" quando o pino está no nível lógico alto. Para que possamos monitorar o estado deste pino, é necessário realizar repetidas leituras do valor do IDR dentro de um *loop*. Este processo é denominado ***Polling***.

3. Dentro da função *main*, escreva o seguinte código:

```
int main(void){
	uint8_t status = 0;

	RCC->AHB4ENR |= 0x00000006; // Habilita GPIOB eGPIOC
```

```
	GPIOB->MODER |= 0x00000001; //PB0 como saida digital
	GPIOB->MODER &= 0xFFFFFFFD;
	GPIOB->OTYPER &= 0xFFFFFFFE; // PB0 como push-pull
	GPIOC->MODER &= 0xF3FFFFFF; // PC13 como entrada digital
	GPIOB->ODR &= 0xFFFFFFFE; //PB0 = 0
	/* Loop forever */
	for(;;) { // Fazendo Polling
		if(GPIOC->IDR & (1 << 13)) { // PC13 = 1
			if(status) { // LED esta On
				GPIOB->ODR &= 0xFFFFFFFE; // LED Off
				status = 0;
			} else { // LED esta Off
				GPIOB->ODR |= 0x00000001; // LED On
				status = 1;
			}
			while(GPIOC->IDR & (1 << 13));
		}
	}
}
```

Note o teste feito no início do *loop for* infinito. O valor de IDR é lido e é feita uma máscara AND com todos os *bits* em "0", exceto o *bit* 13. A expressão "**(1 << 13)**" significa o valor "1" sofrendo um deslocamento de 13 *bits* para a esquerda (em binário, 0010 0000 0000 0000). O valor resultante será zero se o *bit* 13 do IDR estiver em "0", ou o valor 0x2FFFFFFF se o *bit* 13 estiver em "1". Ou seja, o código dentro do *if* só será executado caso o botão esteja pressionado, pois valores zero são interpretados no *if* como "falso" e qualquer valor diferente de zero é interpretado como "verdadeiro".

Alternativamente, o CMSIS disponibiliza macros para as máscaras de *bits* que podem ser usadas para setar ou limpar *bits* dos registradores, geralmente denominadas **campos de bits**. Estas máscaras estão definidas também no arquivo "stm32h7a3xxq.h", e usam sempre o sufixo "**_Msk**". Para o *bit* 13 do GPIOx_IDR, existe a máscara **GPIO_IDR_ID13_Msk**. Faça uma busca pelo nome da máscara no arquivo (Ctrl+F) de definições. A definição para esta máscara é "(0x1UL << GPIO_IDR_ID13_Pos)". Ou seja, o valor "1" expresso como "Unsigned Long" (UL, para ter os 32 *bits*), com um deslocamento de *bits* à esquerda de um valor definido por "GPIO_IDR_ID13_Pos". Na linha imediatamente acima, encontramos sua definição, que é "(13U)", ou seja, o valor 13 sem sinal. Assim, a máscara corresponde à expressão "(1 << 13)" usada originalmente. Se desejar, pode mudar a linha " while(GPIOC->IDR & (1 << 13));" para "while(GPIOC->IDR & GPIO_IDR_ID13_Msk);", pois o resultado será o mesmo.

Dentro do *if*, a variável *status* é verificada. Esta variável segue o estado do LED: valor "1" se o LED está aceso e valor "0" se o LED está apagado. Assim, o estado do LED é invertido. Após ao inversão do estado do LED, existe um *loop* do tipo ***while***. Este *loop* permanece em

execução enquanto a condição lógica entre parênteses for verdadeira. No programa acima, o *loop while* atualiza a leitura do botão, só permitindo a continuidade da execução do programa quando o mesmo for solto (entrada em nível baixo). Veja que não há o "abre-e-fecha colchetes" neste *loop*, apenas um ponto-e-vírgula após a declaração do *while*. Isto indica ue nenhuma instrução será executada enquanto o programa não sair do *loop*, exceto as existentes dentro da condição de teste (no caso, a leitura do registrador e a máscara AND). Este laço de espera é crucial para evitar que o LED alterne repetidamente entre os estados enquanto o botão estiver pressionado. Sem esse laço, o LED poderia passar por várias transições rápidas, em vez de realizar apenas uma mudança de estado quando o botão é pressionado e solto.

4. Realize o *Build* e o D*ebug*. Execute o programa e veja o estado do LED se invertendo a cada pressão do botão.

5. Adicione um *breakpoint* na linha "if(status)". Veja que o programa apenas atinge o *breakpoint* quando o botão é pressionado. A partir daí, execute o programa passo a passo (tecla F6 ou botão "Step Over") e veja o comportamento do programa.

6. Remova o *breakpoint* do item anterior, e acrescente as seguintes linhas:

```
uint32_t d;
```
(logo abaixo a declaração da variável "status")

```
for(d = 0; d < 16000000; d++);
```
(logo após a linha "`for(;;) {`")

Faça o *Build* e o *Debug* e execute o programa. Veja o que acontece ao se pressionar o botão. Veja o que acontece ao se manter o botão pressionado por vários segundos. O LED está respondendo às ações sobre o botão como esperado? Por quê isto acontece? Veja que este é um dos principais problemas do *polling*.

7. Substitua a linha de comando "`for(d = 0; d < 16000000; d++);`" pela chamada da função "espera (16000000)", conforme implementada no Roteiro 2. O código atualizado, que utiliza a interface CMSIS, deve se parecer com o exemplo abaixo. Após realizar a alteração, faça o *Build* e transfira o código para o microcontrolador em modo *Debug*.

```c
int main(void)
{
    // Inicializa GPIOB (PB0) e GPIOC (PC13)
    RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk |
                    RCC_AHB4ENR_GPIOCEN_Msk;

    // PB0 como saida digital
    GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk);
    GPIOB->MODER |= GPIO_MODER_MODE0_0;

    GPIOB->OTYPER &= ~GPIO_OTYPER_OT0_Msk; // PB0 como push-pull

    // PC13 como entrada digital
    GPIOC->MODER &= ~(GPIO_MODER_MODE13_Msk);

    //Inicializa PB0 apagado
    GPIOB->ODR &= ~GPIO_ODR_OD0_Msk;

    uint8_t status=0;

    /* Loop forever */
    for(;;) {
        espera (16000000);
        if (GPIOC->IDR & GPIO_IDR_ID13_Msk) {
            if (!status) {
                GPIOB->ODR |= GPIO_ODR_OD0_Msk; // PB0 = 1
                status = 1;
            } else {
                GPIOB->ODR &= ~GPIO_ODR_OD0_Msk; // PB0 = 0
                status = 0;
            }
            while (GPIOC->IDR & GPIO_IDR_ID13_Msk);
        }
    }
}
```

Compare as instruções usando as macros da interface CMSIS
GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk);
GPIOB->MODER |= GPIO_MODER_MODE0_0;

com as instruções apresentadas no Roteiro 2 usando endereços dos registradores:
GPIOB_MODER&=0xFFFFFFFC;
GPIOB_MODER|=0x00000001;

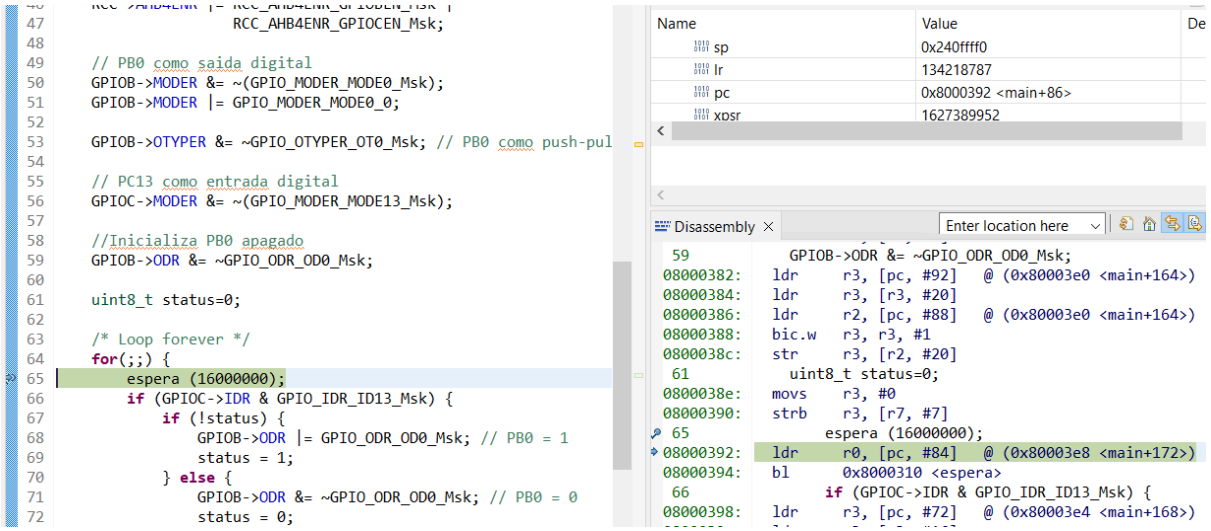
Ambos os conjuntos de instruções são equivalentes. Para verificar isso, você pode rastrear as macros definidas no arquivo de cabeçalho “stm32h7a3xxq.h”. No editor da perspectiva C/C++, você pode consultar essas macros passando o *mouse* sobre elas.

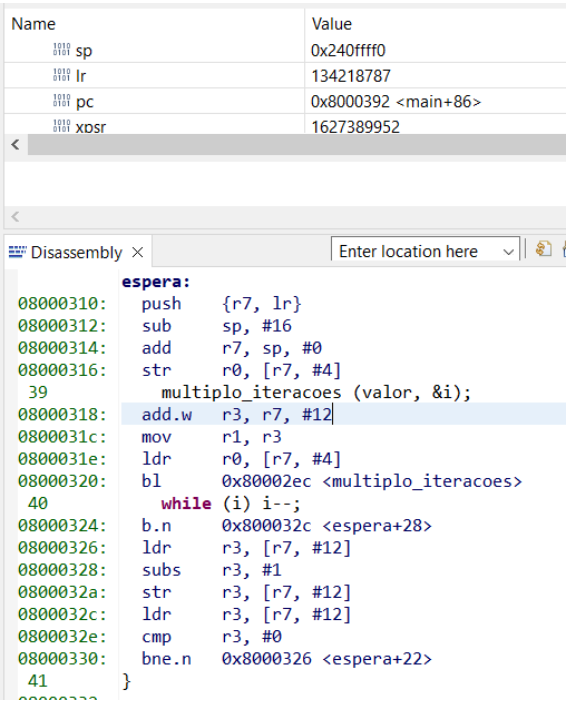
```
#define GPIO_MODER_MODE0_Pos          (0U)
#define GPIO_MODER_MODE0_Msk          (0x3UL << GPIO_MODER_MODE0_Pos)
                                      /*!< 0x00000003 */
#define GPIO_MODER_MODE0              GPIO_MODER_MODE0_Msk
#define GPIO_MODER_MODE0_0            (0x1UL << GPIO_MODER_MODE0_Pos)
                                      /*!< 0x00000001 */
```

Substituindo as macros pelas suas definições, recursivamente até alcançar as macros-folha, como faz o pré-processador, obtemos os valores reais que o compilador interpretará:

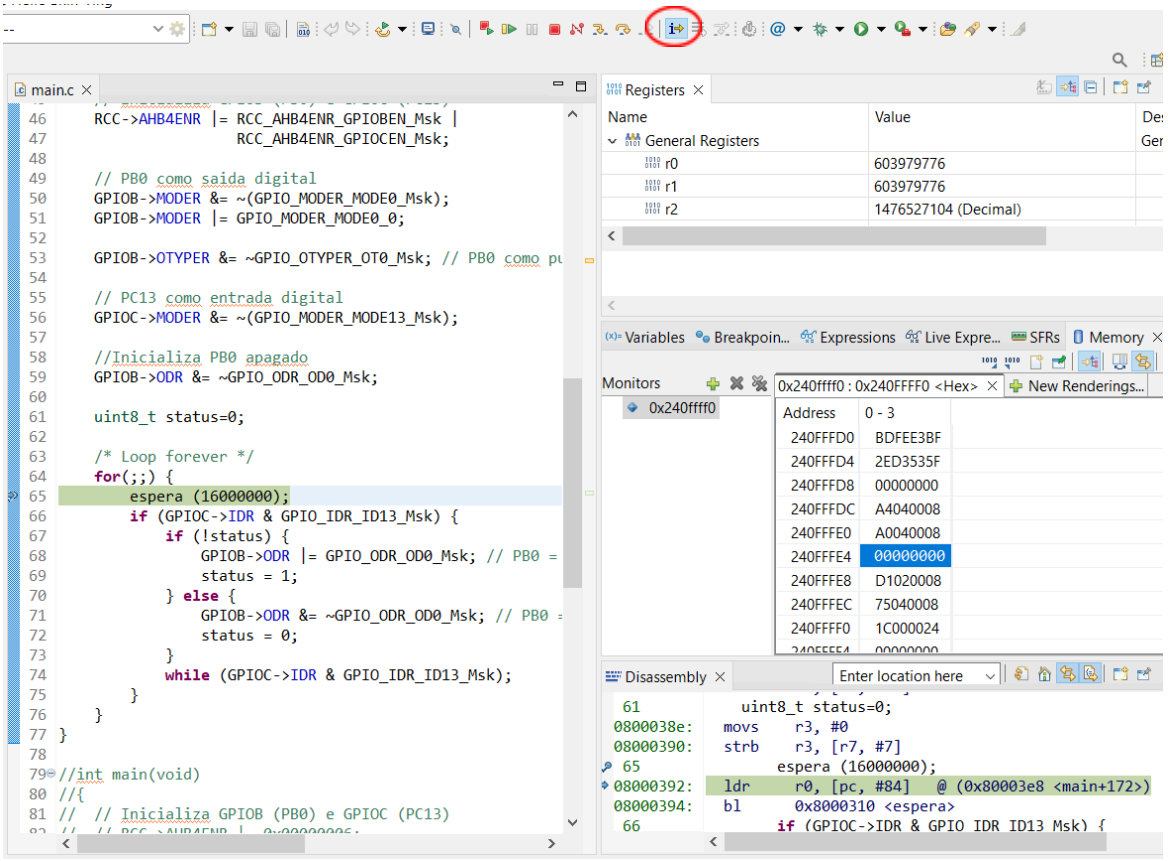
~(GPIO_MODER_MODE0_Msk) = ~(0x3UL << GPIO_MODER_MODE0_Pos)
= ~(0x3UL << 0U) = ~(0x3UL) = 0xFFFFFFFC
GPIO_MODER_MODE0_0 = (0x1UL << GPIO_MODER_MODE0_Pos)
= (0x1UL << 0) = 0x1UL

8. Abra a aba “Disassembly” na perspectiva “Debug” e coloque um *breakpoint* na linha que contém “espera (16000000)”. Inicie a execução do programa. Quando a execução parar no *breakpoint*, observe a linha correspondente em *assembly* na aba "Disassembly". Em seguida, verifique o conteúdo do endereço 0x8000310 na aba “Disassembly”.





9. Abra a aba “Memory” e insira o endereço do ponteiro da pilha (SP ou R13). Configure o formato de exibição para “Row Size” e “Column Size” igual a 4. Em seguida, abra a aba “Registers” para monitorar o conteúdo dos registradores. Ative o modo “Instruction Stepping” pelo ícone destacado para avançar a execução do programa, instrução por instrução em *assembly*.

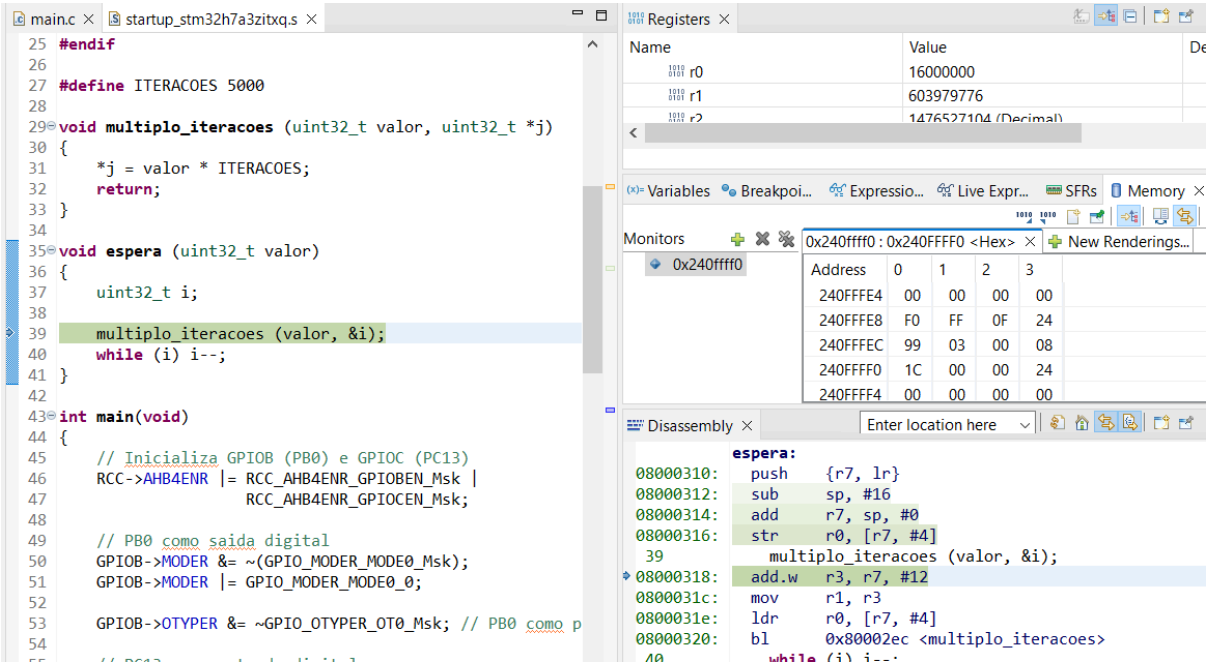


10. Avance um passo “Step Over” (tecle F6) e registre o valor carregado em r0, PC, SP.

11. Na linha da instrução “bl 0x8000310”, avance com o comando "Step Into" (tecla F5) para entrar na função “espera”. Em seguida, use o comando "Step Over" (tecla F6) para avançar pelas instruções até chegar na instrução “add.w r3, r7, #12”. Monitore as alterações no conteúdo da pilha a cada passo. Responda às seguintes perguntas:

- Qual é a função da instrução “push [r7, lr]”?
- Quais dados foram empilhados desde a entrada na função “espera” até este ponto?
- Quais deles são originados durante o chaveamento de contexto?
- Em qual endereço (da pilha) está armazenado o parâmetro de entrada “valor”?
- Em qual endereços (da pilha) está armazenado a variável local “i”?
- Qual registrador é usado como *frame pointer*?
- Quais são os endereços relativos (deslocamentos ou *offsets*) de “valor” e “i” em relação ao *frame pointer*?

Na aba "Memory", destaque o quadro de pilha (*stack frame*) da chamada da função “espera”.



12. Avance usando "Step Over" até a instrução “bl 0x80002ec”. Antes de utilizar "Step Into" para entrar na função “multiplo_iteracoes”, registre o conteúdo dos registradores r0, r1, PC e SP.

13. Ao entrar na função “multiplo_iteracoes”, avance com "Step Over" até a instrução “ldr r3, [r7,#4]”. Registre o conteúdo da pilha após a entrada na função. Identifique como os valores das variáveis “valor” e “j” são acessados. Além disso, determine qual é o quadro de pilha (*stack frame*) para esta chamada e qual registrador está sendo usado como *frame pointer* para a chamada desta função.

14. Com base na análise realizada até agora, descreva as diferenças entre a passagem de parâmetros por valor e a passagem de parâmetros por (valor de) endereço em termos de instruções de máquina na linguagem C. Considere como cada método afeta a execução e o armazenamento dos dados.

15. Continue avançando com "Step Over" até chegar à instrução “return;” no código C. Em seguida, avance até a instrução “bx lr”, enquanto monitora o conteúdo dos registradores r7 e SP. Essas instruções são responsáveis pela restauração do contexto anterior antes de retornar ao fluxo da função “espera”. Não se esqueça de registrar o conteúdo do registrador lr.

16. Dê um “Step Over” em “bx lr”. Anote o endereço da instrução depois do passo e compare-o com o valor de PC que você anotou antes de entrar na função “multiplo_interacoes”.

17. Continue avançando com "Step Over" até atingir a instrução “pop {r7,pc}”, monitorando o conteúdo dos registradores r7, PC e SP durante o processo. Após executar um "Step Over"

na instrução “pop {r7,pc}”, registre o conteúdo atualizado desses registradores. Além disso, explique a função desta instrução.

18. Analise o que foi observado durante o percurso pelo fluxo de execução do programa. Compare suas observações com os conceitos que você aprendeu. A execução do programa está alinhada com esses conceitos?

Um Projeto de GPIO com Interrupção

Para utilizar uma transição de nível lógico em um pino GPIO como uma interrupção, é necessário conectar o pino configurado como entrada ao periférico EXTI, que é responsável por detectar eventos e gerar interrupções. Além disso, deve-se habilitar o NVIC para processar a interrupção, configurando a prioridade, o tipo de evento e a função ISR correspondente. Vamos executar os passos necessários:

1. Vamos escrever um programa que troque o estado do LED verde quando o botão é pressionado, mas agora usando uma interrupção. Inicialmente, crie o projeto “Toggle_Int” da mesma forma que o projeto anterior deste roteiro. Lembre-se de selecionar “Empty” na opção “Targeted Project Type” e de adicionar as pastas com os arquivos de suporte “stm32h7a3xxq.h” e “core_cm7.h”, além de colocar o “include” para o arquivo de cabeçalho.

2. Vamos repetir a configuração do pino PB0 para o LED verde, e vamos incluir a configuração do PC13 como entrada. Repetindo a ativação simultânea de GPIOB e GPIOC:
RCC_AHB4ENR |= 0x00000006;

Na sequência, os registradores de GPIOB são configurados como no projeto anterior, para que se possa ligar e desligar o LED verde. Para a entrada do *push-button*, vamos configurar o pino 13 de GPIOC como uma requisição se interrupção sensível à borda de subida (transição de nível baixo para alto). Inicialmente, configura-se o pino como entrada:
GPIOC_MODER &= 0xF3FFFFFF;

O gerenciamento das linhas de interrupção externas ao microcontrolador é feita pelo módulo EXTI e o mapeamento dos pinos de entrada GPIO nos sinais EXTIn é feito por SYSCFG. Inicialmente, precisamos habilitar o módulo SYSCFG, que é ligado ao barramento APB4. Assim, a habilitação do módulo ocorre no registrador RCC_APB4ENR. O Manual de Referência mostra que o *bit* 1 deste registrador habilita o módulo:
RCC->APB4ENR |= 0x00000002;

8.7.46 RCC APB4 clock register (RCC_APB4ENR)

Address offset: 0x154

Reset value: 0x0001 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	DFSDM2EN	DTSEN	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RTCAPBEN
				rw	rw										rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VREFEN	COMP12EN	DAC2EN	Res.	Res.	LPTIM3EN	LPTIM2EN	Res.	I2C4EN	Res.	SPI6EN	Res.	LPUART1EN	Res.	SYSCFGEN	Res.
rw	rw	rw			rw	rw		rw		rw		rw		rw	

Agora precisamos conectar o pino PC13 à linha de interrupção EXTI13. Cada interrupção externa EXTIn (n = 0..15) pode ser conectada ao pino de mesmo número de qualquer porta. A definição da porta usada para cada número de pino é feita pelos registradores SYSCFG_EXTICR1, SYSCFG_EXTICR2, SYSCFG_EXTICR3 e SYSCFG_EXTICR4. Para a EXTI13, usa-se os *bits* 4, 5, 6 e 7 de SYSCFG_EXTICR4. A combinação de *bits* que especifica a Porta C é 0010. Assim:

```
SYSCFG->EXTICR[3] &= 0xFFFFFF0F;
SYSCFG->EXTICR[3] |= 0x00000010;
```

Note que os 4 registradores EXTICRn estão estruturados em um vetor. Na linguagem C, o índice inicial de um vetor é sempre 0. Assim, os registradores de 1 a 4 estão mapeados nas posições de vetores de 0 a 3, respectivamente.

Os detalhes da configuração da interrupção são realizados pelo módulo EXTI. Os registradores EXTI_RTSRn e EXTI_FTSRn (n = 1..3) determinam se aquela interrupção externa será chamada pela borda de subida e pela de descida respectivamente. Nós queremos apenas a borda de subida, e assim precisamos setar o *bit* correspondente em EXTI_RTSRn e limpar o *bit* correspondente em EXTI_FTSRn. Para EXTI13, usamos o *bit* 13 dos registradores de número 1:

```
EXTI->RTSR1 |= 0x00002000; // Habilita borda de subida de EXTI13
EXTI->FTSR1 &= 0xFFFFDFFF; // Desabilita borda de descida de EXTI13
```

Ainda no módulo EXTI, é necessário habilitar a **máscara de interrupção** correspondente a EXTI13. A máscara de interrupção permite que o código possa bloquear temporariamente uma imterrupção quando necessário. Os registradores de máscara de interrupção do módulo são EXTI_CPUIMRn (n = 1..3). Para EXTI13, precisamos setar o *bit* 13 de EXTI_CPUIMR1:

```
EXTI->IMR1 |= 0x00002000; // Habilita mascara de EXTI13
```

Para finalizar, precisamos configurar o NVIC para usar prioridade 1 em nossa interrupção (quanto menor o número, mais prioritária é a interrupção) e habilitar a mesma. No Manual de Referência, podemos ver a tabela de vetores de interrupção. Cada interrupção tem um

número, e as 16 primeiras (0 a 15) são as exceções geradas pelo próprio núcleo ARM. A coluna “NVIC position” indica o número de requisição de interrupção. Podemos ver que a interrupção de número 40 é a EXTI15_10, ou seja, há apenas um vetor de interrupção para tratar as interrupções externas de 10 a 15. É este vetor que devemos usar para EXTI13.

Signal	Priority	NVIC position	Acronym	Description	Address offset
i2c2_ev_it exti_i2c2_ev_wkup	40	33	I2C2_EV	I2C2 event interrupt	0x0000 00C4
i2c2_err_it	41	34	I2C2_ER	I2C2 error interrupt	0x0000 00C8
spi1_it exti_spi1_it	42	35	SPI1	SPI1 global interrupt	0x0000 00CC
spi2_it exti_spi2_it	43	36	SPI2	SPI2 global interrupt	0x0000 00D0
usart1_gbl_it exti_usart1_wkup	44	37	USART1	USART1 global interrupt	0x0000 00D4
usart2_gbl_it exti_usart2_wkup	45	38	USART2	USART2 global interrupt	0x0000 00D8
usart3_gbl_it exti_usart3_wkup	46	39	USART3	USART3 global interrupt	0x0000 00DC
exti_exti10_wkup exti_exti11_wkup exti_exti12_wkup exti_exti13_wkup exti_exti14_wkup exti_exti15_wkup	47	40	EXTI15_10	EXTI Line[15:10] interrupts	0x0000 00E0

Há um conjunto de registradores de prioridade e de habilitação das interrupções no NVIC. Os detalhes podem ser encontrados na Documentação da ARM.

Para simplificar o trabalho, o CMSIS usa macros definidas para ajustar prioridade e habilitar as interrupções no NVIC. Assim, usamos:

```
NVIC_SetPriority(EXTI15_10_IRQn, 1);  // Prioridade 1
NVIC_EnableIRQ(EXTI15_10_IRQn);  // Habilita
```

O STM32CubeIDE também simplifica o trabalho de achar o endereço de entrada da ISR e carregar no vetor de interrupção. Ele já define uma função “void EXTI15_10_IRQHandler(void)” que é chamada no atendimento da interrupção, e deve ser definida dentro do arquivo “main.c”. As definições dos nomes de funções ISR para cada vetor da tabela pode ser encontrada no arquivo “startup_stm32h7a3zitxq.s”, na pasta “Startup”.

```
.word     TIM2_IRQHandler                 /* TIM2                                 */
.word     TIM3_IRQHandler                 /* TIM3                                 */
.word     TIM4_IRQHandler                 /* TIM4                                 */
.word     I2C1_EV_IRQHandler              /* I2C1 Event                           */
.word     I2C1_ER_IRQHandler              /* I2C1 Error                           */
.word     I2C2_EV_IRQHandler              /* I2C2 Event                           */
.word     I2C2_ER_IRQHandler              /* I2C2 Error                           */
.word     SPI1_IRQHandler                 /* SPI1                                 */
.word     SPI2_IRQHandler                 /* SPI2                                 */
.word     USART1_IRQHandler               /* USART1                               */
.word     USART2_IRQHandler               /* USART2                               */
.word     USART3_IRQHandler               /* USART3                               */
.word     EXTI15_10_IRQHandler            /* External Line[15:10]s                */
.word     RTC_Alarm_IRQHandler            /* RTC Alarm (A and B) through EXTI Line */
.word     DFSDM2_IRQHandler               /* DFSDM2 Interrupt                     */
.word     TIM8_BRK_TIM12_IRQHandler       /* TIM8 Break and TIM12                 */
.word     TIM8_UP_TIM13_IRQHandler        /* TIM8 Update and TIM13                */
.word     TIM8_TRG_COM_TIM14_IRQHandler   /* TIM8 Trigger and Commutation and TIM14 *
.word     TIM8_CC_IRQHandler              /* TIM8 Capture Compare                 */
.word     DMA1_Stream7_IRQHandler         /* DMA1 Stream7                         */
.word     FMC_IRQHandler                  /* FMC                                  */
.word     SDMMC1_IRQHandler               /* SDMMC1                               */
.word     TIM5_IRQHandler                 /* TIM5                                 */
.word     SPI3_IRQHandler                 /* SPI3                                 */
.word     UART4_IRQHandler                /* UART4                                */
.word     UART5_IRQHandler                /* UART5                                */
.word     TIM6_DAC_IRQHandler             /* TIM6 and DAC1&2 underrun errors */
.word     TIM7_IRQHandler                 /* TIM7                                 */
```

Para verificar se um número de interrupção foi ativado, verifica-se os registradores de *flag*. Ao atender a interrupção, este "flag" deve ser limpa, normalmente escrevendo "1" sobre o *bit* correspondente (do inglês *write-one-to-clear*, sigla **w1tc**).

3. Dentro do arquivo main.c, **antes** da função main(), inclua a definição da rotina de serviço:

```
// ISR para linhas EXTI 15_10
void EXTI15_10_IRQHandler(void) {
    static uint8_t status=0;

    if (EXTI->PR1 & EXTI_PR1_PR13_Msk) { // Testa "flag" de EXTI13
        // Limpar o flag EXTI13
        EXTI->PR1 |= EXTI_PR1_PR13_Msk;

        // Vamos fazer o "toggle" do LED aqui
        if(status) { // LED esta On
            GPIOB->ODR &= ~GPIO_ODR_OD0_Msk; // LED Off
            status = 0;
        } else { // LED esta Off
            GPIOB->ODR |= GPIO_ODR_OD0_Msk; // LED On
            status = 1;
        }
    }
}
```

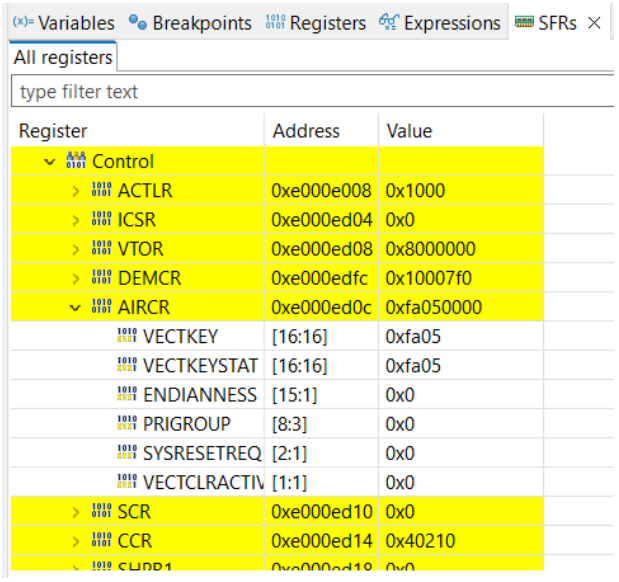
Veja que a variável "status" agora foi declarada como estática, pois suas modificações dentro da ISR deverão ser preservadas. Vemos ainda que no início da ISR testamos se o evento que a chamou foi realmente o EXTI13, e, caso positivo, limpa o *flag*. Essa prática de verificação é importante, uma vez que há mais de uma fonte de interrupção associada à IRQ40. A limpeza do *flag* remove a interrupção do estado pendente. Na sequência, é realizado o *toggle* do LED. Note também que não é necessário esperar o botão ser solto para terminar o processo, pois o que é testado é a transição do nível lógico baixo para o alto, e não o nível alto. A seguir, apresentamos uma implementação do arquivo "main.c" utilizando exclusivamente as macros da interface CMSIS. Vale ressaltar que as funções NVIC_SetPriorityGrouping,

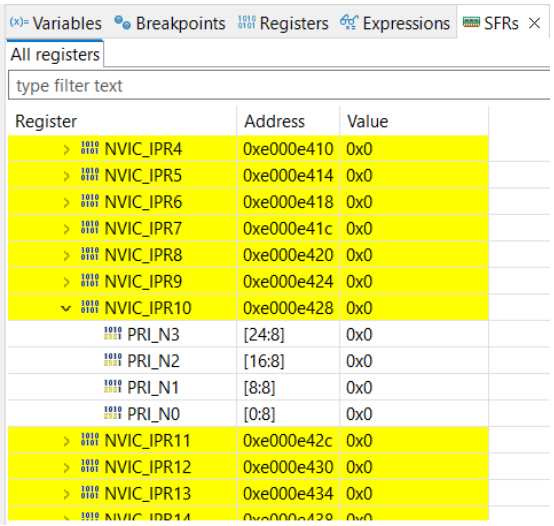
NVIC_SetPriority e **NVIC_EnableIRQ**, também fornecidas pela CMSIS, oferecem uma abstração um pouco maior como alternativas.

```
int main(void)
{
	// Ativa GPIOB (PB0) e GPIOC (PC13)
	RCC->AHB4ENR |= RCC_AHB4ENR_GPIOBEN_Msk |
	                RCC_AHB4ENR_GPIOCEN_Msk;
	// PB0 como saida digital
	GPIOB->MODER &= ~(GPIO_MODER_MODE0_Msk);
	GPIOB->MODER |= GPIO_MODER_MODE0_0;
	GPIOB->OTYPER &= ~GPIO_OTYPER_OT0_Msk; // PB0 como push-pull
	// PC13 como entrada digital
	GPIOC->MODER &= ~(GPIO_MODER_MODE13_Msk);
	// Ativar o SYSCFG
	RCC->APB4ENR |= RCC_APB4ENR_SYSCFGEN_Msk;
	// Conectar PC3 com EXTI13
	SYSCFG->EXTICR[3] &= 0xFFFFFF0F;
	SYSCFG->EXTICR[3] |= SYSCFG_EXTICR4_EXTI13_PC;
	// Ativar a borda de subida e desativar borda de descida
	EXTI->RTSR1 |= EXTI_RTSR1_TR13_Msk;
	EXTI->FTSR1 &= ~EXTI_FTSR1_TR13_Msk;
	// Desmascarar EXTI13
	EXTI->IMR1 |= EXTI_IMR1_IM13_Msk;
	// Ativar IRQ 40 do NVIC com prioridade 1 (macros definidos em core_cm7.h)
	uint8_t prioridade = 1;
	//NVIC_SetPriorityGrouping(0b11);
	//Montar a palavra para escrever os bits simultaneamente
	uint32_t reg_value;
	reg_value  =  SCB->AIRCR;
	reg_value  &=  ~((uint32_t)(SCB_AIRCR_VECTKEY_Msk  |  SCB_AIRCR_PRIGROUP_Msk));  //
limpar bits
	reg_value  =  (reg_value                          |
	        ((uint32_t)0x5FAUL << SCB_AIRCR_VECTKEY_Pos) |
	          ((0b11 & (uint32_t)0x07UL) << SCB_AIRCR_PRIGROUP_Pos)  );  // chave e formato de
agrupamento
	SCB->AIRCR =  reg_value;
	  //NVIC_SetPriority(EXTI15_10_IRQn, 1);
	//Seta a prioridade nos bits [7:5] do byte de prioridade no registrador NVIC_IPRm
	NVIC->IP[40]=(uint8_t)((prioridade << (8U-__NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
	//NVIC_EnableIRQ(EXTI15_10_IRQn);
	//Setar o bit 40%32=(40&0x01) do registrador NVIC_ISERn, onde n = 40/32 = 40<<5.
	NVIC->ISER[(((uint32_t)40) >> 5UL)] = (uint32_t)(1UL << (((uint32_t)40) & 0x1FUL));
	//Inicializa PB0 apagado
	GPIOB->ODR &= ~GPIO_ODR_OD0_Msk;
	/* Loop forever */
	uint32_t d;
	for(;;) {
		// Fazendo Polling
		for(d = 0; d < 16000000; d++);
	}
}
```

Observe que estamos mantendo o *loop* de atraso que causava problemas no *polling*. Isso está impactando a alternância como ocorria no projeto anterior? Você poderia explicar o que está acontecendo? Além disso, note que a alternância dos estados do LED não é mais realizada dentro do laço principal. Quando ocorre essa alternância agora? E por qual motivo foi feita essa mudança?

4. Realize o *Build* e transfira o executável para o microcontrolador no modo *Debug* do programa. Expanda o item "Control" na aba SFRs para visualizar o conteúdo dos registradores VTOR e AIRCR. Em seguida, expanda o item "NVIC" e consulte o valor do *byte* 40%4 do registrador NVIC_IPRn. Note que, para IRQ40, n = 40/4 = 10.





Com base nisso, responda às seguintes perguntas:

- Qual é o deslocamento em relação ao endereço-base 0x00000000 configurado para o endereço inicial da Tabela de Vetores de Interrupção? Está condizente com o endereço onde se encontra armazenado SP?
- Qual é o agrupamento de prioridades configurado?

5. Defina um *breakpoint* na linha "NVIC->IP[40]=(uint8_t)((prioridade << (8U-__NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);" e um breakpoint na linha "GPIOB->ODR &= ~GPIO_ODR_OD0_Msk;". Em seguida, continue (*Resume*) a execução do programa. Quando o código parar no primeiro *breakpoint*, verifique o formato de agrupamento setado no campo PRIGROUP. Note a instrução "SCB->AIRCR = reg_value;" de atribuição direta do valor "reg_value" no registrador SBC->AIRCR. É possível substituir pela instrução "SCB->AIRCR |= reg_value;"? Justifique com base no que vimos no Roteiro 2.

Continue (*Resume*) a execução do programa. Ao parar no segundo *breakpoint*, veja a prioridade setada para a linha de requisição de interrupção IRQ40 no *byte* menos significativo do registrador NVIC_IPR10 e o *bit* 8 do registrador NVIC_ISER1. Estão condizentes com as instruções executadas? Justifique.

Continue (*Resume*) a execução do programa. Veja que, mesmo com o *loop* de atraso, não há mais problemas operacionais.

6. Um teste simples para verificar se o mecanismo de interrupção foi configurado corretamente é definir um *breakpoint* na primeira instrução da rotina de serviço. Em seguida, produza um evento que acione a interrupção e observe se a execução pára no *breakpoint*. Ative a aba "Disassembly", reinicie o programa e execute-o novamente. Ao parar no *breakpoint*, analise as instruções anteriores e posteriores à instrução correspondente na aba *Disassembly*. Identifique o quadro de pilha (*stack frame*) e o *frame pointer* para a chamada dessa ISR.

7. Compare o processamento do desvio de fluxo para as funções "espera" e "multiplo_interacoes" do projeto anterior com o processamento de desvio de fluxo nas ISRs deste projeto. Analise como o desvio de fluxo é gerenciado em cada caso e explique por quê as ISRs geralmente não possuem argumentos.

8. Quais alterações você faria para que o LED mude de estado na borda de descida do sinal em PC13 e reduzir a prioridade para 3?

9. Problemas de **reentrância** ocorrem quando uma função pode ser interrompida durante sua execução e chamada novamente antes que a primeira execução seja concluída, levando a comportamentos inesperados e difíceis de depurar. Considere a seguinte função em um cenário onde duas rotinas de serviço de interrupção (ISRs), ISR A e ISR B, podem chamar a mesma função:

```
void AAA () {
    static uint32_t i=0;
   i++;
```

```
    if ( i%2 == 0) {
        // execute A
    } else {
        // execute B
    }
}
```

É possível que ocorram problemas de reentrância se ISR A puder interromper ISR B? Justifique sua resposta.

10. Comente sobre a seguinte afirmação: “O processamento de interrupções envolve tanto *hardware* quanto *software*. O *hardware* detecta e sinaliza a interrupção, enquanto o *software* é responsável por lidar com a interrupção após sua ocorrência. Isso inclui a execução da rotina de serviço de interrupção, o chaveamento de contexto e a gestão da concorrência de recursos para garantir a integridade dos dados e do estado do sistema durante a execução do programa. Portanto, a resolução de problemas relacionados à reentrância é, predominantemente, uma questão de *design* e implementação de *software*.”

CARACTERÍSTICAS BÁSICAS DE SINAIS DIGITAIS

Sinais digitais são fundamentais para a comunicação e operação em sistemas digitais, desempenhando um papel crucial na troca precisa de informações. Entre as características principais que definem qualquer sinal digital, quatro aspectos básicos são essenciais para a interpretação e resposta dos sistemas digitais às mudanças no sinal, frequentemente refletindo alterações em fenômenos físicos:

Borda de Subida: Corresponde à transição de um nível baixo para um nível alto. Esta transição é crítica para a detecção de eventos ou a sincronização de operações. Em muitos sistemas, a borda de subida é usada para indicar o início de uma nova operação ou a ativação de um estado.

Borda de Descida: Corresponde à transição de um nível alto para um nível baixo. Assim como a borda de subida, a borda de descida é fundamental para detectar eventos e sincronizar atividades dentro de um sistema. Muitas vezes, é utilizada para marcar a conclusão de uma operação ou para sinalizar a desativação de um estado.

Nível Alto: Representa um estado em que o sinal digital está em seu valor máximo. Este nível geralmente corresponde a uma condição lógica “1”. O nível alto é frequentemente associado à ativação de funções ou à comunicação de dados importantes, servindo como um indicativo de que um sinal ou dado está presente.

Nível Baixo: Representa um estado em que o sinal digital está em seu valor mínimo, correspondente a uma condição lógica “0”. Este nível indica a ausência de um sinal ativo ou a desativação de uma função. O nível baixo é essencial para o reconhecimento de períodos de

inatividade ou para representar a ausência de dados.

POLLING

Varredura (em inglês, *polling*) é uma técnica amplamente utilizada em sistemas embarcados para verificar periodicamente o estado de um dispositivo ou recurso, com o objetivo de determinar se ele requer atenção ou se há novos dados disponíveis. Em vez de aguardar que o dispositivo sinalize um evento, o processo de varredura envolve a consulta contínua ou periódica do estado do dispositivo. No contexto de sinais digitais, essa técnica consiste essencialmente na detecção de alterações em fenômenos físicos através de um dos quatro eventos ou estados básicos: borda de subida, borda de descida, nível alto e nível baixo.

Na prática, periféricos são projetados para detectar as mudanças nos estados físicos através da observação dos quatro eventos mencionados. A técnica de varredura simplifica a tarefa ao reduzir o problema a detectar um desses eventos específicos, amostrando periodicamente o sinal para identificá-los.

As principais vantagens da varredura incluem sua simplicidade e previsibilidade. A abordagem é fácil de implementar e não requer estruturas complexas, oferecendo um comportamento previsível no processo de verificação do estado dos dispositivos. No entanto, as desvantagens são significativas: a varredura pode ser ineficiente e introduzir latência. Isso ocorre porque o sistema gasta ciclos de processamento constantemente verificando o estado do dispositivo, mesmo quando não há mudanças, o que pode resultar em um uso ineficiente dos recursos e maior tempo de resposta para eventos importantes.

EXCEÇÕES E INTERRUPÇÕES

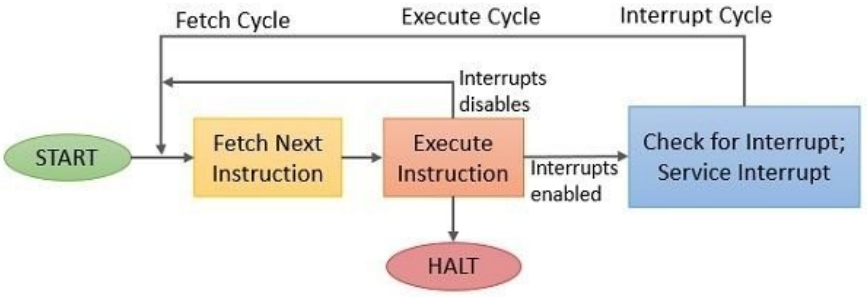
A maneira mais comum de se priorizar o tratamento de um evento (como o apertar do botão) é com o uso das **interrupções.** Exceções e interrupções são mecanismos fundamentais em sistemas de computação para lidar com eventos inesperados e garantir o processamento das tarefas dedicadas ao tratamento destes eventos. Embora ambos envolvam a interrupção do fluxo normal de execução, suas formas de geração e detalhes de processamento apresentam diferenças significativas.

Exceções são eventos que ocorrem como resultado de condições anômalas durante a execução de um programa. Elas podem ser geradas por erros de *software*, como divisão por zero, instruções inválidas ou acesso a áreas de memória proibidas. Exceções são geralmente síncronas, ou seja, ocorrem em resposta direta a uma instrução específica que está sendo executada. Elas são intrínsecas ao processo de execução do programa e são frequentemente utilizadas para implementar mecanismos de tratamento de erros e garantir a integridade do sistema.

Por outro lado, **interrupções** são eventos assíncronos causados por fontes externas ao fluxo de execução do programa, como periféricos, temporizadores ou sinais de *hardware*. Quando um dispositivo precisa da atenção do processador, ele gera uma interrupção utilizando um dos quatro possíveis estados dos sinais digitais (bordas ou níveis). Essa interrupção sinaliza que um evento específico ocorreu, forçando o processador a interromper seu trabalho atual para tratar o evento. Isso permite que o sistema responda rapidamente a eventos externos, sem a necessidade de verificar continuamente seu estado.

Dependendo de como o processador gerencia as interrupções, elas podem ser classificadas como vetorizadas ou não-vetorizadas. Em um sistema com **interrupções vetorizadas**, cada fonte de interrupção é associada a um número de vetor. Esse número é utilizado pelo processador para localizar o endereço do manipulador (em inglês, *handler*), conhecido como rotina de serviço de interrupção (em inglês, *Interrupt Service Routine* – ISR), em uma estrutura chamada tabela de vetores de interrupção. Dessa forma, quando uma interrupção ocorre, o processador pode redirecionar o fluxo de controle diretamente para a ISR associada. Em contraste, **interrupções não-vetorizadas** não utilizam uma tabela de vetores para mapear as interrupções. Em vez disso, o processador pode precisar realizar etapas adicionais para determinar qual interrupção ocorreu e qual ISR deve ser executada, o que pode envolver verificações adicionais e processamento mais complexo para identificar a origem da interrupção e seu tratamento adequado.

Quando a ISA adota uma abordagem de interrupção vetorizada, como a arquitetura ARM, exceções e interrupções, apesar de suas diferenças em origem e natureza, compartilham semelhanças no processamento e podem ser tratadas de maneira uniforme. Cada evento é associado a um **número de vetor** específico. Quando uma exceção ou interrupção ocorre, o tratamento segue um ciclo padronizado dentro do ciclo de instrução com interrupção.



Instruction Cycle with Interrupts

A arquitetura ARM explora as características básicas dos sinais digitais, distinguindo interrupções sensíveis ao nível e de pulso. As **interrupções de pulso** também são descritas como interrupções acionadas por borda. Uma interrupção sensível ao nível é mantida ativa até que o periférico desative o sinal de interrupção. Normalmente isso acontece quando a ISR acessa o periférico, fazendo com que ele remova o *flag* de solicitação de interrupção. Uma interrupção de pulso é um sinal de interrupção amostrado de forma síncrona na borda do sinal de *clock* do processador. Para garantir que o processador detecte a interrupção de pulso, o periférico deve ativar o sinal de interrupção por pelo menos um ciclo de relógio, durante o

qual o processador detecta o pulso e trava a interrupção.

Quando o processador entra na ISR, ele remove automaticamente o estado pendente da interrupção, como descreve o Manual de Programação. Para uma **interrupção sensível ao nível**, se nível do sinal não for desativado antes do processador retornar da ISR, o processador deve executar sua ISR novamente. Isso significa que o periférico pode reter o sinal de interrupção ativado até que ele não precise mais de atendimento. Para uma interrupção acionada por borda, o sinal de interrupção é continuamente monitorado mesmo estando em execução a ISR. Assim, quando um pulso é detectado, a solicitação é ativada e, ao retornar de uma ISR, o processador re-entra novamente na ISR se não houver outras solicitações prioritárias.

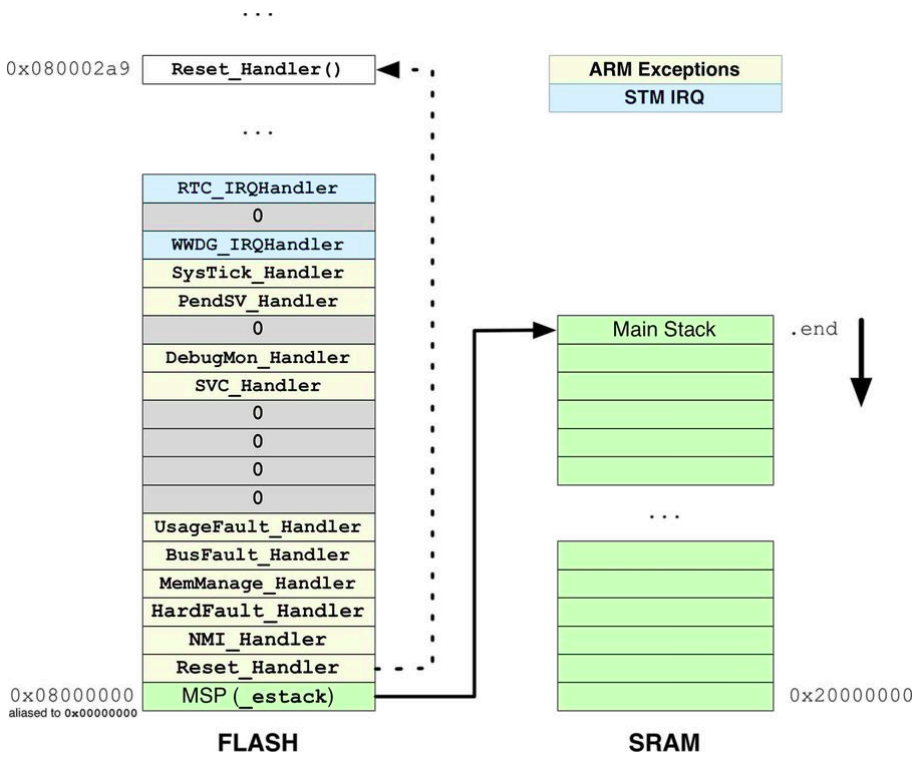
Ciclo de Interrupção

O ciclo de interrupção/exceção consiste dos seguintes passos:

Interrupção do Fluxo Normal de Execução: O processador desvia temporariamente do código em execução para tratar o evento, seja ele uma condição de erro ou uma sinalização de *hardware*.

Uso de Tabela de Vetores de Interrupção: Quando uma exceção ou interrupção ocorre, a tabela de vetores de interrupção é utilizada pelo processador para localizar o endereço da ISR correspondente e redireciona o fluxo de execução para esse endereço. A ISA especifica não apenas a presença e a organização dos endereços das ISRs na tabela, mas também a localização desta tabela na memória.

Para processadores baseados na arquitetura ARM Cortex-M, o endereço-padrão da tabela de vetores é o início da memória de instrução, que geralmente é 0x00000000. No entanto, no núcleo Cortex-M7 do microcontrolador STM32H7A3ZIT6-Q, a tabela de vetores é mapeada para o endereço 0x08000000 no espaço de endereçamento do processador através do registrador VTOR, como ilustra a figura. Assim, a referência ao endereço 0x08000000 como o local da tabela de vetores é específica para essa configuração de mapeamento de memória e reflete a forma como a memória FLASH é mapeada neste microcontrolador.. Para mais detalhes sobre a organização da tabela, consulte o Manual de Referência. Observe que as 16 primeiras entradas da tabela de vetores são reservadas para exceções, enquanto as 155 entradas subsequentes são destinadas a interrupções.



Salvamento do Contexto e Estado: Antes de transferir o controle para a ISR, o processador salva automaticamente o estado atual do programa, incluindo o contador de programa (PC) e outros registradores, em uma pilha. Isso assegura que o fluxo de execução possa ser retomado corretamente após o tratamento da interrupção. Na arquitetura ARM, é chamada essa transferência de controle de chaveamento de contexto (do inglês *context switching*) do fluxo de controle atual, seja do modo Thread ou do modo Handler, para um novo fluxo de controle sempre do modo Handler. Para garantir a integridade do fluxo de controle neste chaveamento, o compilador e o ligador geram o código necessário para realizar as seguintes tarefas conforme as convenções e requisitos da arquitetura ARM ao compilar uma função que define uma ISR:

- finalizar a execução da instrução corrente,
- salvar na pilha os registradores de AAPCS (do inglês ARM *Architecture Procedure Call Standard*): R0, R1, R2, R3, R12, LR, PC, APSR (do inglês *Application Program Status Register*), que contém as *flags* de ALU necessárias para desvios condicionaisl e operação de algumas instruções, IPSR (do inglês *Interrupt Program Status Register*), que contém o número de vetor, e os registradores de FPUs,
- setar o registrador de retorno LR (do inglês *Link Register*) ou R14, com o valor de retorno da exceção, EXEC_RETURN.
- atualizar o registrador de estado IPSR com o número de exceção do novo contexto, e
- carregar no PC (*Program Counter)* ou R15, que mantém o endereço da próxima instrução, o endereço da ISR armazenado na Tabela de Vetores de Interrupção.

Execução de Código de Tratamento: Tanto exceções quanto interrupções são tratadas por um código específico definido em uma ISR. Esse código é responsável por resolver a condição que causou a exceção ou responder ao evento de *hardware*, garantindo que o

sistema possa continuar funcionando corretamente.

As definições de ISRs e os endereços das suas instruções na memória são fortemente dependentes do aplicativo em desenvolvimento. Cada ISR deve ser implementada conforme as necessidades específicas do aplicativo, e os endereços dessas rotinas na tabela de vetores de interrupção precisam ser configurados para refletir corretamente onde cada ISR está localizada na memória. Para facilitar o desenvolvimento e reduzir a complexidade associada ao gerenciamento manual desses endereços, o STM32CubeIDE oferece uma funcionalidade prática. Em vez de exigir que o desenvolvedor configure manualmente os endereços das ISRs, o STM32CubeIDE cria o arquivo "Startup/startup_stm32h7a3zitxq.s" que contém a tabela de vetores de interrupção inicialmente preenchida com símbolos no lugar de endereços específicos das ISRs:

```
  .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                     */
  .word FLASH_IRQHandler                                  /* Flash memory global interrupt            */
  .word RCC_IRQHandler                                    /* RCC global interrupt                     */
  .word EXTI0_IRQHandler                                  /* EXTI Line 0 interrupt                    */
  .word EXTI1_IRQHandler                                  /* EXTI Line 1 interrupt                    */
  .word EXTI2_IRQHandler                                  /* EXTI Line 2 interrupt                    */
  .word EXTI3_IRQHandler                                  /* EXTI Line 3interrupt                     */
  .word EXTI4_IRQHandler                                  /* EXTI Line 4interrupt                     */
  .word DMA_STR0_IRQHandler                               /* DMA1 Stream0 global interrupt            */
  .word DMA_STR1_IRQHandler                               /* DMA1 Stream1 global interrupt            */
  .word DMA_STR2_IRQHandler                               /* DMA1 Stream2 global interrupt            */
  .word DMA_STR3_IRQHandler                               /* DMA1 Stream3 global interrupt            */
  .word DMA_STR4_IRQHandler                               /* DMA1 Stream4 global interrupt            */
  .word DMA_STR5_IRQHandler                               /* DMA1 Stream5 global interrupt            */
  .word DMA_STR6_IRQHandler                               /* DMA1 Stream6 global interrupt            */
  .word ADC1_2_IRQHandler                                 /* ADC1 and ADC2 global interrupt           */
  .word FDCAN1_IT0_IRQHandler                             /* TTCAN Interrupt 0                        */
```

O desenvolvedor pode então implementar as ISRs como funções C normais, utilizando os mesmos símbolos especificados na tabela de vetores para garantir consistência. Por exemplo, se a ISR para o módulo "SysTick" é definida como "SysTick_Handler", o código pode ser escrito da seguinte forma:

```
void SysTick_Handler(void) {
        //código de tratamento de interrpção

}
```

Durante o processo de vinculação, o ligador GNU do STM32CubeIDE localiza as definições das funções ISR no código e associa essas definições aos símbolos listados na tabela de

vetores de interrupção. Ele então resolve as referências a esses símbolos, substituindo-os pelos endereços reais das funções ISR no programa final. Isso garante que a tabela de vetores contenha os endereços corretos para que, quando uma interrupção ocorre, o processador possa redirecionar o fluxo de execução para a ISR apropriada.

O STM32CubeIDE também trata a situação em que uma definição de ISR está ausente para um evento de interrupção específico, fornecendo uma função padrão chamada "Default_Handler" no mesmo arquivo "Startup/startup_stm32h7a3zitxq.s".

```
/*******************************************************************************
*
* Provide weak aliases for each Exception handler to the Default_Handler.
* As they are weak aliases, any function with the same name will override
* this definition.
*
*******************************************************************************/

        .weak   NMI_Handler
        .thumb_set NMI_Handler,Default_Handler

        .weak   HardFault_Handler
        .thumb_set HardFault_Handler,Default_Handler

        .weak   MemManage_Handler
        .thumb_set MemManage_Handler,Default_Handler

        .weak   BusFault_Handler
        .thumb_set BusFault_Handler,Default_Handler

        .weak   UsageFault_Handler
        .thumb_set UsageFault_Handler,Default_Handler

        .weak   SVC_Handler
        .thumb_set SVC_Handler,Default_Handler

        .weak   DebugMon_Handler
        .thumb_set DebugMon_Handler,Default_Handler

        .weak   PendSV_Handler
        .thumb_set PendSV_Handler,Default_Handler

        .weak   SysTick_Handler
        .thumb_set SysTick_Handler,Default_Handler

        .weak   WWDG_IRQHandler
        .thumb_set WWDG_IRQHandler,Default_Handler
```

A função de substituição "Default_Handler" é automaticamente utilizada pelo sistema para lidar com interrupções para as quais não há um manipulador definido, garantindo que o sistema tenha uma resposta predefinida e evitando possíveis falhas ou comportamentos indefinidos.

```
/**
 * @brief  This is the code that gets called when the processor receives an
 *         unexpected interrupt.  This simply enters an infinite loop, preserving
 *         the system state for examination by a debugger.
 *
 * @param  None
 * @retval : None
*/
  .section .text.Default_Handler,"ax",%progbits
Default_Handler:
Infinite_Loop:
  b Infinite_Loop
  .size Default_Handler, .-Default_Handler
```

Retorno ao Fluxo Normal: Após o tratamento da exceção ou interrupção, o processador restaura o estado salvo do programa e retoma a execução do código no ponto em que foi interrompido usando o retorno de exceção. Como **o tratamento de exceção ocorre no modo Handler**, há três tipos de retorno, indicado pelo valor EXEC_RETURN: para modo *Handler*, para modo *Thread* privilegiado ou para modo *Thread* não-privilegiado. O valor EXEC_RETURN é carregado no LR na entrada da exceção.

EXC_RETURN	Behavior
0xFFFFFFF1	Return to Handler Mode. Exception return gets state from the Main stack. On return execution uses the Main Stack.
0xFFFFFFF9	Return to Thread Mode. Exception return gets state from the Main stack. On return execution uses the Main Stack.
0xFFFFFFFD	Return to Thread Mode. Exception return gets state from the Process stack. On return execution uses the Process Stack.
Unused	Reserved.

Prioridade de Atendimento

As interrupções/exceções permitem que o microcontrolador responda imediatamente os eventos externos/internos do processador, suspendendo temporariamente a execução do fluxo de controle principal (núcleo) para lidar com uma situação emergente. No entanto, quando múltiplas interrupções ocorrem simultaneamente ou em rápida sucessão, é necessário garantir que eventos sejam atendidos pelo processador na ordem dos seus níveis de criticidade ou de prioridade. Portanto, existem dois conceitos fundamentais em interrupções que o programador deve ter em mente: preempção e prioridade. A **preempção** se refere à capacidade do sistema de interromper a execução de uma tarefa para que uma tarefa mais importante ou urgente possa ser executada, o que é realizado através das interrupções. A **prioridade** de interrupção determina a ordem de importância das interrupções. Em um sistema onde múltiplas interrupções podem ocorrer, a cada uma é atribuída um nível de prioridade. Interrupções com prioridade mais alta podem interromper aquelas com prioridade mais baixa.

Atualmente, praticamente todos os processadores provêm níveis de prioridade de interrupção para gerenciar de forma eficiente as interrupções que ocorrem no sistema. O nível de prioridade de uma ISR em execução é setado como a **prioridade de execução** do processador. Enquanto esta ISR estiver executando, existem duas abordagens para lidar com a

ocorrência de interrupções de prioridade maior: a técnica de interrupções aninhadas e a técnida de interrupções não-aninhadas. Em sistemas com **interrupções não-aninhada**s, uma vez que uma interrupção começa a ser atendida, nenhuma outra interrupção pode interrompê-la até que o atendimento seja concluído. As interrupções são atendidas em uma ordem sequencial, conforme o nível de prioridade configurado enquanto não houver nenhuma ISR em execução. As **interrupções aninhadas** permitem, por sua vez, que uma interrupção em andamento seja interrompida por outra interrupção de prioridade maior do que a prioridade da interrupção em execução. Assim, o sistema pode responder imediatamente a eventos mais críticos, mesmo que uma interrupção de menor prioridade esteja sendo atendida.

Para tratamento de interrupções aninhadas, o *software* deve estar preparado para aceitar outra interrupção/exceção, mesmo antes de terminar de tratar a interrupção/exceção atual. Vale a pena ressaltar que o tratamento de interrupções aninhadas é uma escolha feita pelo *software*, em função da configuração da prioridade de interrupção e do controle de interrupção ajustada pelo desenvolvedor conforme as necessidades específicas das aplicações. Ela não é imposta pelo *hardware*.

Uma consideração crucial nesse contexto é a diferença entre subrotinas aninhadas e subrotinas reentrantes para processamento de interrupções. **Subrotinas aninhadas** se referem a funções que podem chamar outras funções, inclusive aquelas que podem estar em execução simultaneamente. Isso pode complicar o controle do fluxo de execução e a manutenção do estado, especialmente quando se trata de interrupções aninhadas, onde uma interrupção pode chamar outra antes que a primeira tenha sido completamente tratada.

Subrotinas reentrantes, por outro lado, são projetadas para serem seguras para chamadas simultâneas. Isso significa que uma subrotina reentrante pode ser interrompida no meio de sua execução e, posteriormente, ser chamada novamente antes que a execução anterior seja concluída. A subrotina deve então ser capaz de retomar a execução de maneira correta e segura sem corromper os dados ou falhar na execução. Esse comportamento é essencial para o tratamento eficaz de interrupções, especialmente em sistemas multitarefa e sistemas embarcados, onde a reentrância pode permitir que múltiplos eventos sejam gerenciados de forma eficiente.

Os problemas desafiadores de reentrância incluem garantir que todas as variáveis usadas em uma subrotina reentrante sejam locais ou devidamente protegidas contra **condições de corrida** (em inglês *race conditions*) e corrupções de dados. Além disso, o uso de recursos compartilhados, como *buffers* de dados e variáveis globais, precisa ser cuidadosamente sincronizado para evitar inconsistências. Desenvolvedores enfrentam o desafio de implementar mecanismos para proteger esses recursos e garantir que as subrotinas possam ser interrompidas e retomadas sem problemas.

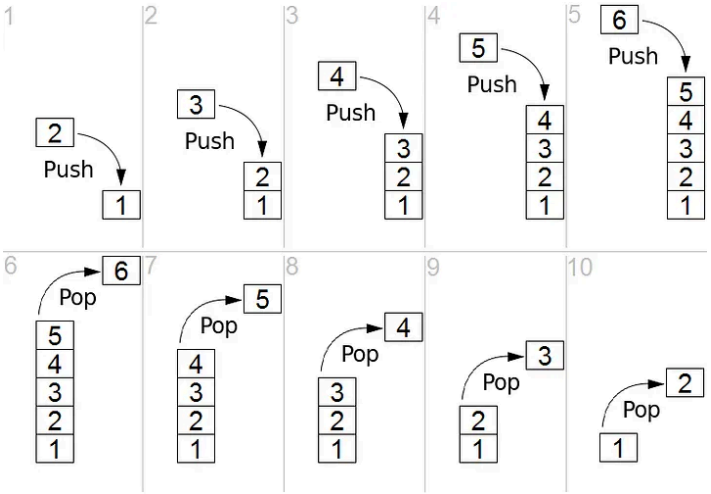
Portanto, embora o tratamento de interrupções aninhadas e a configuração de prioridade por *hardware* proporcionem flexibilidade no gerenciamento de eventos, a implementação de subrotinas reentrantes pelo desenvolvedor é essencial para assegurar a robustez e a

confiabilidade de sistemas em ambientes altamente concorrentes e de alta prioridade. A capacidade de gerenciar interrupções de maneira segura e eficiente é um aspecto crítico do desenvolvimento de um sistema embarcado. Isso exige um domínio profundo tanto do cenário de concorrência quanto das técnicas de programação, para alcançar um equilíbrio cuidadoso entre flexibilidade, desempenho e segurança.

Idealmente, uma interrupção aninhada deve ser reentrante, o que significa que o sistema deve preservar a integridade dos dados e do estado para permitir que uma ISR suspensa retome sua execução a partir do ponto em que foi interrompida, após a conclusão da ISR de maior prioridade. De acordo com o Manual da ARM, para garantir que uma ISR reentrante possa lidar corretamente com interrupções aninhadas, é essencial salvar o estado atual da interrupção, proteger os recursos compartilhados, identificar corretamente as fontes de interrupção e gerenciar as *flags* de interrupção de maneira eficaz.

ESTRUTURA DE DADOS: PILHAS

A pilha é um subespaço contíguo da memória destinada ao armazenamento de dados temporários. O acesso à pilha é feito no modo *Last In, First Out* (LIFO) onde o último dado inserido é o primeiro a ser removido. Duas operações são associadas a essa estrutura de dados: **empilhar** (do inglês ***push***) um novo elemento sobre o topo e **desempilhar** (do inglês ***pop***) do topo um elemento. O endereço do topo é armazenado numa variável para facilitar o seu processamento. Hoje em dia todos os processadores suportam a implementação de pilhas, provendo o registrador denominado **ponteiro de pilha SP** (do inglês ***stack pointer***), e instruções de empilhamento e desempilhamento, como as instruções PUSH e POP, além de operações de memória como LDR e STR.



Existem convenções de gerenciamento de pilha, como ilustra a figura, Na **pilha ascendente** (em inglês, *full ascending stack*), a memória é alocada em direção ao alto endereço (ou seja, os endereços de memória aumentam conforme mais dados são empilhados). Ao adicionar um novo item à pilha, você move o ponteiro da pilha para um endereço de memória maior. Na **pilha descendente** (em inglês, *full descending stack*), a memória é alocada em direção ao baixo endereço (ou seja, os endereços de memória diminuem conforme mais dados são empilhados). Ao adicionar um novo item à pilha, você move o ponteiro da pilha para um

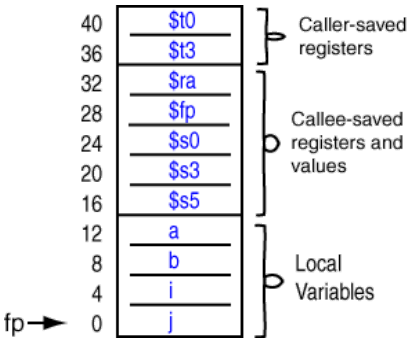
endereço de memória menor. Na arquitetura ARM, a abordagem padrão é usar uma pilha descendente. Ou seja, a pilha começa em um endereço alto e cresce em direção a endereços baixos.

A pilha é fundamental para operações como chamadas de funções e o tratamento de interrupções ou exceções. Para uma implementação detalhada e completa dessa estrutura em C, consulte o artigo disponível no GeeksForGeeks.

De acordo com a AAPCS (do inglês ARM *Architecture Procedure Call Standard*), os primeiros quatro parâmetros são passados nos registradores r0 a r3. Parâmetros adicionais são passados na pilha. Por exemplo, para a função, os parâmetros "a", "b", "c" e "d" são passados nos registradores r0, r1, r2 e r3, enquanto os parâmetros "e" e "f" são empilhados e acessados na função "Exemplo 1" via pilha:

```
void Exemplo 1 (int a, int b, int c, int d, int e, int f) {
        // Código da função
}
```

A pilha é amplamente utilizada para armazenar variáveis locais em uma função devido à sua capacidade de alocar e desalocar memória de forma rápida e eficiente. Como parte do processo de chamada de uma função, o compilador de C gera as instruções necessárias para salvar e restaurar o estado dos registradores na pilha, como também para reservar o espaço para as variáveis locais na pilha. A figura ilustra o empilhamento dos registradores que a função chamadora precisa preservar (em inglês *caller-saved registers*), registradores que a função chamada precisa preservar (em inglês *callee-saved registers and values*) e variáveis locais (em inglês *local variables*) da função chamada, realizado por essas funções.



É comum denominar de **quadro da pilha** (em inglês ***stack frame***) o bloco de memória da pilha resenvado para processamento de uma função chamada. Define-se ainda como **ponteiro do quadro** (em inglês ***frame pointer***) um endereço-base, tipicamente o topo da pilha. Assim, as instruções da função chamada podem acessar todos os seus dados pré-empilhados usando deslocamentos fixos em relação ao ponteiro do quadro fp, ao invés de variar o ponteiro da pilha SP, empilhando e desempilhando as variáveis locais na execução das instruções. Ao final da função, basta desemepilhar o quadro para restaurar o estado da pilha antes da chamada da função. O registrador r7 é tipicamente utilizado como ponteiro de quadro.

A menos das operações dedicadas à transferência de operandos entre o processador e a

memória, todas as instruções da arquitetura ARM envolvem somente os registradores. Como são poucos os registradores quando o porcessador executa no modo Thumb, a pilha é também muito utilizada para salvar e restaurar os valores dos registradores.

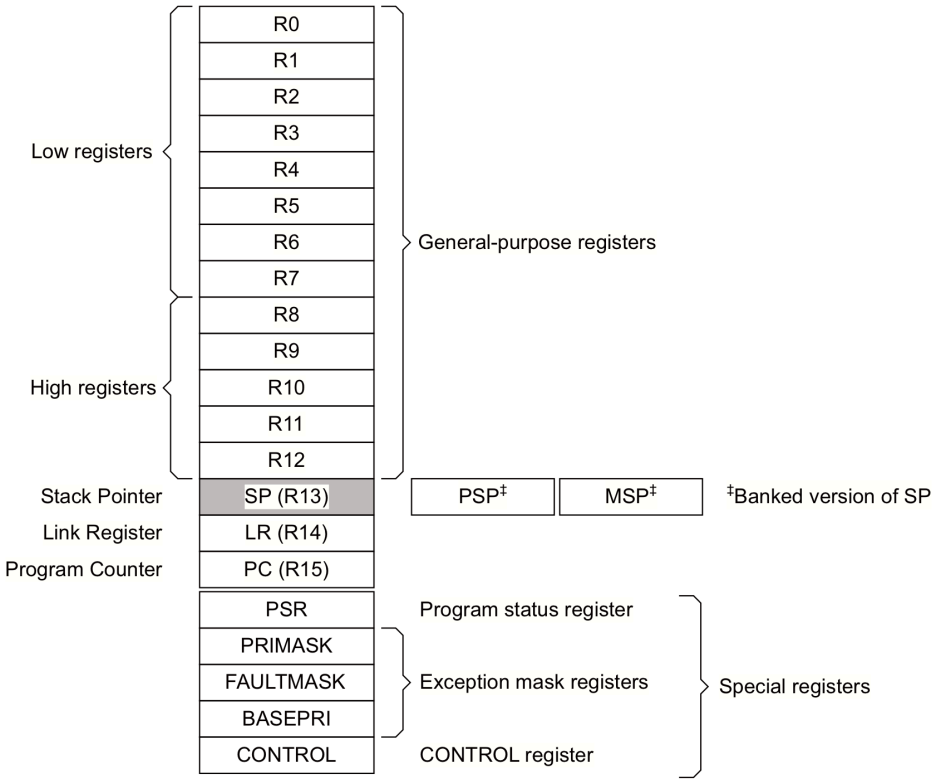
Os processadores ARM Cortex-M são equipados de *hardware* de **ponteiro de pilha dedicado (R13)** para operações de pilha. Eles possuem uma instrução PUSH para armazenar dados na pilha e uma instrução POP para recuperar dados. O ponteiro de pilha atualmente selecionado é ajustado automaticamente para cada operação PUSH e POP. Como a arquitetura ARM usa uma pilha descendente, o ponteiro da pilha sempre aponta para os últimos dados armazenados na memória da pilha e o ponteiro da pilha pré-decrementa para cada nova operação PUSH.

O STM32H7A3ZIT6-Q mantém, em particular, dois ponteiros de pilha:

MSP (do inglês *Main Stack Pointer*) é o ponteiro-padrão, configurado durante a inicialização do sistema e usado sempre no modo Handler e no modo Thread supervisor. Sua alteração é menos comum e geralmente feita apenas em configuração inicial ou em situações onde é necessário alterar o comportamento do sistema de exceções.

PSP (do inglês *Process Stack Pointer*) pode ser alterado durante a execução do código, especialmente em sistemas de multitarefa. É usado no modo Thread com multi-processos. Em um ambiente com múltiplas tarefas, como sistemas operacionais em tempo real, o PSP pode ser alterado para manter uma pilha separada para cada tarefa, permitindo que o sistema gerencie a execução de várias tarefas de maneira eficiente.

Quando estamos executando nossos projetos em *bare-metal*, sem um gerenciador de multi-tarefas, somente o ponteiro MSP é usado. O PSP foi integrado visando a implantação de sistemas operacionais em tempo real (RTOS) no microcontrolador.



A alocação de um espaço de endereços da memória para a pilha é feita pelo desenvolvedor. Para facilitar a programação do microcontrolador, o *script* "STM32H7A3ZITXQ_FLASH.ld", gerado pelo STM32CubeIDE ao criar um novo projeto, define como o topo da pilha _estack = ORIGIN(RAM)+LENGTH(RAM) = 0x24000000 + 2^{10} * 1024) = 0x24000000 + 0x100000 = 0x24100000, como vimos no Roteiro 2. Este endereços corresponde ao endereço mais alto em que a unidade de memória AXI SRMA3 está mapeado. Pelo script, o tamanho mínimo da pilha é 0x400. Sendo a pilha descendente, os endereços cobertos pela pilha é 0x240FFC00.

```
33
34 /* Entry Point */
35 ENTRY(Reset_Handler)
36
37 /* Highest address of the user mode stack */
38 _estack = ORIGIN(RAM) + LENGTH(RAM);    /* end of RAM */
39 /* Generate a link error if heap and stack don't fit into RAM */
40 _Min_Heap_Size = 0x200;      /* required amount of heap  */
41 _Min_Stack_Size = 0x400; /* required amount of stack */
42
43 /* Specify the memory areas */
44 MEMORY
45 {
46   ITCMRAM (xrw)  : ORIGIN = 0x00000000, LENGTH = 64K
47   FLASH (rx)     : ORIGIN = 0x08000000, LENGTH = 2048K
48   DTCMRAM1 (xrw) : ORIGIN = 0x20000000, LENGTH = 64K
49   DTCMRAM2 (xrw) : ORIGIN = 0x20010000, LENGTH = 64K
50   RAM (xrw)      : ORIGIN = 0x24000000, LENGTH = 1024K
51   RAM_CD (xrw)   : ORIGIN = 0x30000000, LENGTH = 128K
52   RAM_SRD (xrw)  : ORIGIN = 0x38000000, LENGTH = 32K
53 }
54
```

INTERRUPÇÕES/EXCEÇÕES ANINHADAS EM ARM

A arquitetura ARM suporta interrupções aninhadas, permitindo a configuração de prioridades e a preempção de interrupções de maior prioridade. Nessa arquitetura, uma interrupção/exceção só pode interromper (ou antecipar) a execução de uma outra ISR, se a prioridade de atendimento desta ISR for superior à prioridade da ISR em execução. As instruções executadas no modo *Thread* tem sempre a menor prioridade de execução. Vale chamar atenção de que na arquitetura ARM Cortex-M a prioridade de atendimento de uma exceção é inversamente proporcional ao nivel de prioridade associado a ela. Quanto maior o nível, menor a prioridade de atendimento. Por exemplo, como os níveis de prioridade de *Reset*, *NMI* e *HardFault* são, respectivamente, -3, -2 e -1 e os níveis de prioridade configuráveis por *software* são sempre maior ou igual a 0, estas três exceções tem sempre maiores prioridades no atendimento. Cabe também lembrar aqui que não se deve alterar o nível de prioridade de uma exceção quando ela estiver habilitada ou ativa. A tabela de vetores de interrupção com suas respectivas prioridades de atendimento foi extraída do Manual de Referência.

Table 123. NVIC(1)

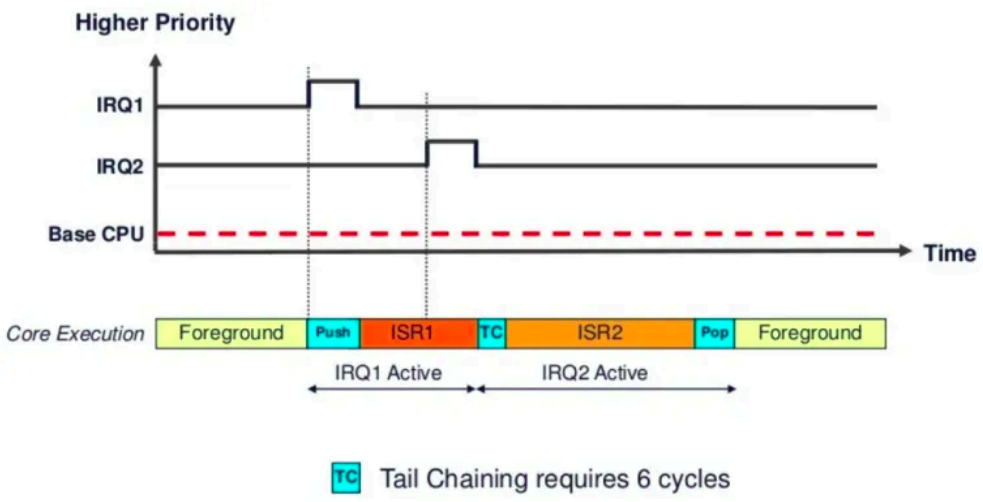
Signal	Priority	NVIC position	Acronym	Description	Address offset
-	-	-	-	Reserved	0x0000 0000
-	-3	-	Reset	Reset	0x0000 0004
-	-2	-	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
-	-1	-	HardFault	All classes of fault	0x0000 000C
-	0	-	MemManage	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
exti_pwr_pvd_wkup	8	1	PVD_PVM	PVD through EXTI line detection interrupt	0x0000 0044

Com exceção de *Reset*, que só tem dois estados básicos, uma interrupção/exceção pode ter mais estados além dos dois estados básicos:

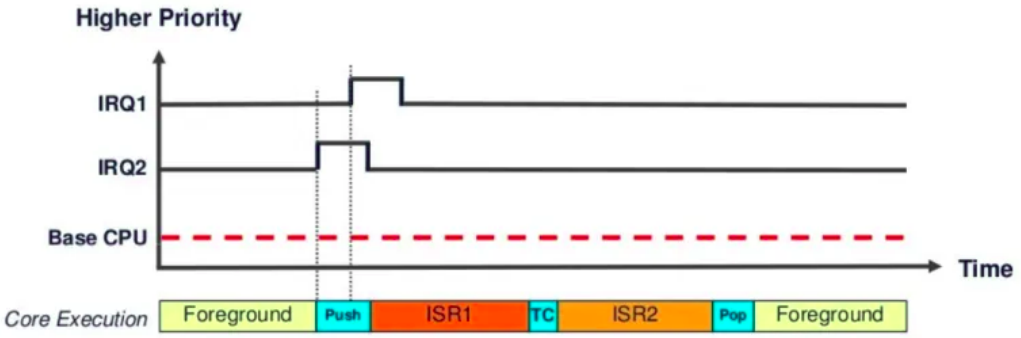
ativo: a interrupção/exceção está em andamento e está sendo atendida pelo processador.
inativo: a interrupção não está sendo atendida pelo processador.

Uma interrupção/exceção pode ser encontrada nos estados:
pendente: quando ela é gerada e está aguardado para ser atendida. Este estado ocorre quando a interrupção/exceção é solicitada, mas o processador ainda não começou a tratá-la, possivelmente porque está atendendo uma interrupção de prioridade maior.
ativo e pendente: uma interrupção/exceção está atualmente ativa e uma nova solicitação para a mesma interrupção foi recebida, colocando-a no estado pendente novamente mesmo estando sendo atendida.

Duas caracteríticas presentes na arquitetura ARM para reduzir a latência no chaveamento entre as ISRs é **encadeamento de interrupções** (em inglês *tail-chaining*) e **atraso de chegada de interrupções** (em inglês *late-arriving*) em interrupções aninhadas. Em vez de salvar e restaurar o contexto (os estados dos registradores) ao alternar entre ISR1 e ISR2 na figura abaixo, em ***tail-chaining*** o processador otimiza o tempo de resposta ao manter o contexto (de ISR1) quando outra interrupção (ISR2) ocorre enquanto uma interrupção está sendo processada.

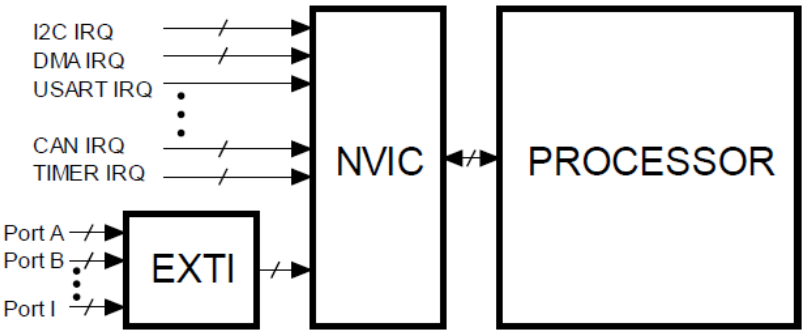


A técnica ***late-arriving***, por sua vez, otimiza o processo de salvamento de estados quando chega uma ISR de prioridade maior (ISR1) enquanto está salvando o estado do fluxo principal para executar uma outra ISR (ISR2), ilustrado na figura abaixo. Neste caso, como o salvamento é o mesmo, o processador apenas conclui o salvamento do contexto atual e chaveia para a ISR de maior prioridade (ISR1).



Controldor de Interrupções Aninhadas (NVIC)

O processamento de interrupções enfrenta desafios críticos, como a gestão de prioridades, aninhamento de interrupções, latência e gerenciamento do estado dos registradores. Para abordar esses problemas, as interrupções são gerenciadas por um módulo denominado **Controlador de Interrupções** em muitos microcontroladores. A arquitetura ARM introduziu o **NVIC** (do inglês ***Nested Vectored Interrupt Controller***), um *hardware* dedicado para gerenciamento de interrupções. A figura sintetiza a relação entre os módulos CPU, NVIC e o controlador EXTI, que veremos adiante, do STM32H7A3ZIT6-Q.



O NVIC do STM32H7A3ZIT6-Q oferece as seguintes funcionalidades:

- Suporte de 1 até 240 interrupções.
- Cada interrupção pode ter um nível de prioridade programável de 0 a 255. Um nível de prioridade mais alto corresponde a uma prioridade mais baixa, sendo que o nível 0 representa a prioridade mais alta.
- Detecção de sinais de interrupção tanto em nível quanto em pulso.
- Repriorização dinâmica das interrupções durante a execução.
- Agrupamento das prioridades em campos de prioridade de grupo e subprioridade, permitindo uma configuração refinada das prioridades das interrupções.
- Suporte de *tail-chaining*, facilitando a gestão de múltiplas interrupções.
- Suporte de Interrupção externa não mascarável (NMI).

Para garantir que o NVIC detecte a interrupção sensível ao nível ou de pulso, o periférico deve ativar o sinal de interrupção por pelo menos um ciclo de clock, durante o qual o NVIC detecta o pulso e trava a interrupção. Quando o processador entra na ISR, ele remove automaticamente o estado pendente da interrupção passando-o para ativo.

O NVIC suporta o conceito de **interrupções aninhadas**, onde uma interrupção em andamento pode ser interrompida por uma interrupção de maior prioridade. Isso é feito através de um controle eficiente dos estados das interrupções, utilizando uma pilha de interrupções que mantém o contexto de cada interrupção enquanto permite que o processador trate interrupções mais urgentes.

O modelo de gerenciamento de interrupções aninhadas no NVIC envolve a utilização de um sistema de prioridades e um mecanismo de vetorização. Quando ocorre uma interrupção, o NVIC verifica a tabela de vetores para localizar a ISR correspondente e, se a interrupção em questão tem uma prioridade superior ao nível de prioridade da ISR em execução, o NVIC interrompe a execução da ISR atual e passa o controle para a ISR de maior prioridade. Após o tratamento da interrupção de alta prioridade, o NVIC retorna ao ponto de interrupção anterior, continuando o processamento da ISR que havia sido interrompida. Esse processo é gerenciado de forma eficiente através da **pilha de interrupções**, que mantém o contexto de cada interrupção ativa, almejando que o sistema retome a execução após a conclusão do tratamento das interrupções.

O circuito de processamento de cada linha n de requisição (IRQn) é configurável através de um conjunto de registradores listados na tabela extraída do Manual de Programação:

Table 40. NVIC register summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100-0xE000E11C	NVIC_ISER0-NVIC_ISER7	RW	Privileged	0x00000000	*Interrupt set-enable registers on page 185*
0XE000E180-0xE000E19C	NVIC_ICER0-NVIC_ICER7	RW	Privileged	0x00000000	*Interrupt clear-enable registers on page 186*
0XE000E200-0xE000E21C	NVIC_ISPR0-NVIC_ISPR7	RW	Privileged	0x00000000	*Interrupt set-pending registers on page 186*
0XE000E280-0xE000E29C	NVIC_ICPR0-NVIC_ICPR7	RW	Privileged	0x00000000	*Interrupt clear-pending registers on page 187*
0xE000E300-0xE000E31C	NVIC_IABR0-NVIC_IABR7	RW	Privileged	0x00000000	*Interrupt active bit registers on page 188*
0xE000E400-0xE000E4EF	NVIC_IPR0-NVIC_IPR59	RW	Privileged	0x00000000	*Interrupt priority registers on page 188*
0xE000EF00	STIR	WO	Configurable(1)	0x00000000	*Software trigger interrupt register on page 189*

Registradores de habilitação de IRQn (NVIC_ISERx): Habilita a IRQn quando escreve ‘1’/ativa o *bit* n%32 do registrador NVIC_ISERx, onde x = n/32.

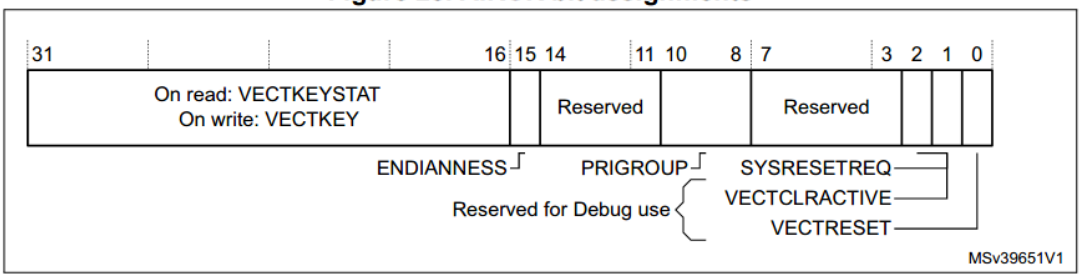
Registrador de desabilitação de uma IRQn (NVIC_ICERx): Desabilita a IRQn quando escreve ‘1’/ativa o *bit* n%32 do registrador NVIC_ICERx, onde x = n/32.

Registrador de habilitação da pendência de uma IRQn (NVIC_ISPRx): Habilita a pendência de IRQn quando escreve ‘1’/ativa o *bit* n%32 do registrador NVIC_ISPRx, onde x = n/32.

Registrador de desabilitação da pendência de uma IRQn (NVIC_ICPRx): Desabilita a pendência de IRQn quando escreve ‘1’/ativa o *bit* n%32 do registrador NVIC_ICPRx, onde x = n/32.

Registrador de prioridade de uma IRQn (NVIC_IPRx): Configura a prioridade de IRQn quando seta a prioridade no *byte* n%4 do registrador NVIC_IPRx, onde x = n/4. Os 8 *bits* disponíveis para configuração de prioridade são divididos em dois campos: um para “prioridade de grupo” com preempção e outro para “subprioridade” sem preempção.

Figure 28. AIRCR bit assignments



O campo PRIGROUP do registrador AIRCR (do inglês *Application Interrupt and Reset Control Register*) permite configurar a quantidade de *bits* mais significativos reservados para a “prioridade de grupo” e a quantidade de *bits* menos significativos reservados para a “subprioridade”. No entanto, o STM32HA3ZIT-Q, como outros microcontroladores da família STM32, permite a configuração da prioridade de interrupção com até 16 níveis distintos, mas a divisão exata entre “prioridade de grupo” e “subprioridade” é flexível e depende do valor definido no campo PRIGROUP, como mostra as 5 últimas linhas na Tabela 56 do Manual de Programação. Vale também observar que, para fazer um acesso de escrita no campo PRIGROUP, é necessário escrever simultaneamente a chave VECTKEY=0x05FA nos 2 *bytes* mais significativos.

Table 56. Priority grouping

	Interrupt priority level value, PRI_*N*[7:0]			Number of	
PRIGROUP	**Binary point(1)**	**Group priority bits**	**Subpriority bits**	**Group priorities**	**Subpriorities**
0b000	bxxxxxxx.y	[7:1]	[0]	128	2
0b001	bxxxxxx.yy	[7:2]	[1:0]	64	4
0b010	bxxxxx.yyy	[7:3]	[2:0]	32	8
0b011	bxxxx.yyyy	[7:4]	[3:0]	16	16
0b100	bxxx.yyyyy	[7:5]	[4:0]	8	32

0b101	bxx.yyyyyy	[7:6]	[5:0]	4	64
0b110	bx.yyyyyyy	[7]	[6:0]	2	128
0b111	b.yyyyyyyy	None	[7:0]	1	256

No STM32H7AI3ZIT6-Q, a prioridade do grupo é o único fator que determina a preempção de uma exceção/interrupção. Quando o processador está processando uma interrupção, outra interrupção com a mesma prioridade de grupo não substitui a ISR em execução. Se houver várias interrupções pendentes com a mesma prioridade de grupo, a ordem de processamento é definida pelo campo de subprioridade. Caso várias interrupções pendentes possuam a mesma prioridade de grupo e subprioridade, a interrupção com o menor número de IRQ será processada primeiro.

Adicionalmente, há um conjunto de registradores **NVIC_IABRx** de leitura dos estados das linhas de interrupção, *bit* em '1' corresponde ao estado ativo e *bit* em '0' ao estado inativo. E os 8 *bits* menos significativos do registrador STIR (do inglês *Software Trigger Interrupt Register*) permitem acionar uma interrupção por *software*, com valores no intervalo de 0 a 239. Por exemplo, um valor de 0x03 escrito no STIR acionará a interrupção IRQ3.

O processador automaticamente empilha o estado do processador ao entrar na ISR e desempilha este estado ao sair dela, garantindo um tratamento de exceções/interrupções com baixa latência e sem sobrecarga adicional de instrução.

O STM32CubeIDE dispõe das funções da interface CMSIS da ARM que facilitam a configuração.

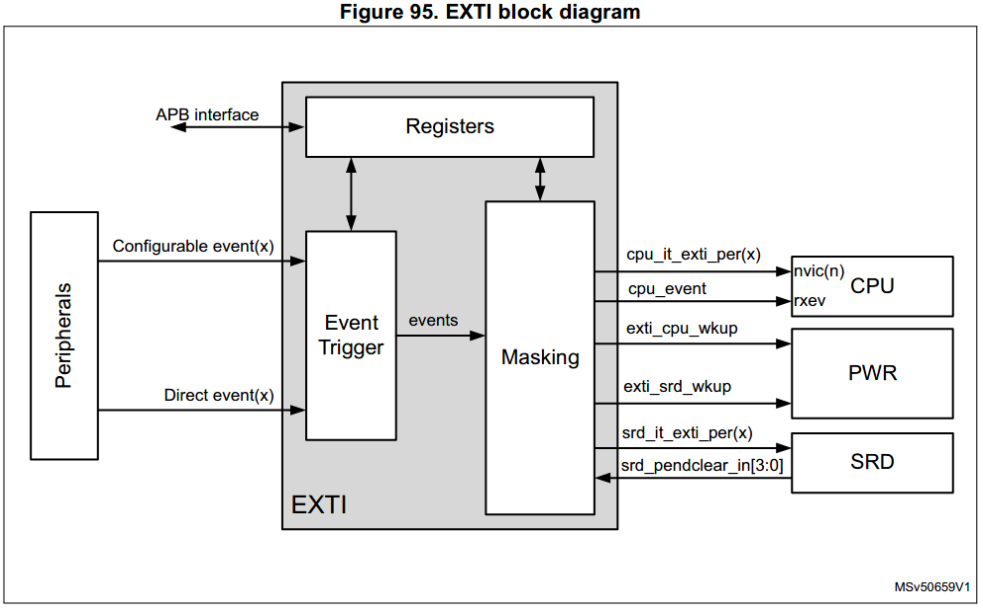
Table 49. CMSIS functions for NVIC control

CMSIS interrupt control function	Description
`void NVIC_SetPriorityGrouping(uint32_t priority_grouping)`	Set the priority grouping
`void NVIC_EnableIRQ(IRQn_t IRQn)`	Enable IRQn
`void NVIC_DisableIRQ(IRQn_t IRQn)`	Disable IRQn
`uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)`	Return true (IRQ-Number) if IRQn is pending
`void NVIC_SetPendingIRQ (IRQn_t IRQn)`	Set IRQn pending
`void NVIC_ClearPendingIRQ (IRQn_t IRQn)`	Clear IRQn pending status
`uint32_t NVIC_GetActive (IRQn_t IRQn)`	Return the IRQ number of the active interrupt
`void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)`	Set priority for IRQn
`uint32_t NVIC_GetPriority (IRQn_t IRQn)`	Read priority of IRQn

Controlador de evento e interrupção estendida (EXTI)

O EXTI (do inglês *Extended Interrupt and Events Controller*) é um módulo projetado para gerenciar eventos provenientes de sinais externos e gerar sinais de interrupção para o processador (CPU) e sinais de acordar do modo de economia de energia tanto para PWR (do inglês *Power Control System Module*) como para SRD (do inglês *Smart Run Domain*). Ele é composto por três principais blocos funcionais que trabalham em conjunto para gerenciar interrupções e eventos.

Figure 95. EXTI block diagram

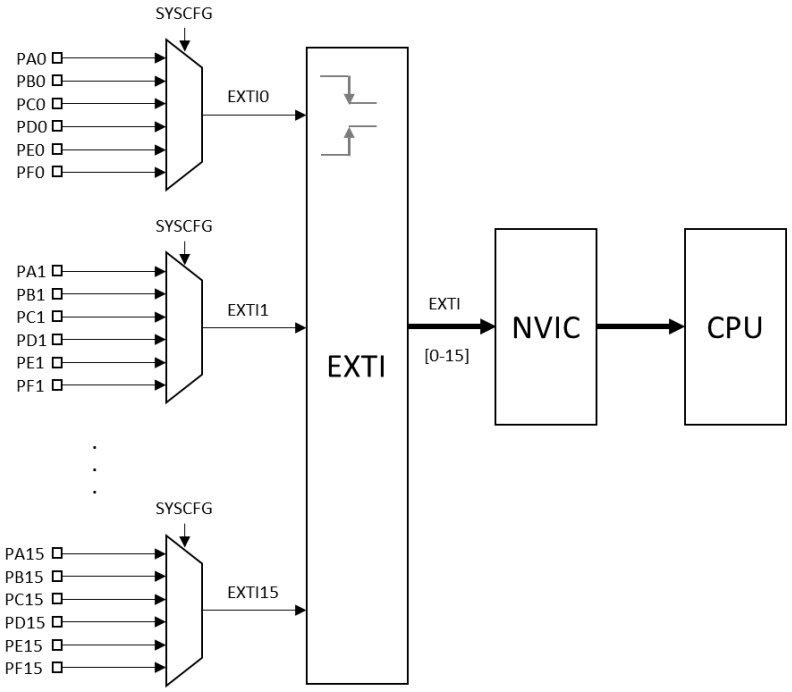


Primeiramente, há o **bloco de registradores (*Registers*)**, que é acessado através do barramento APB (do inglês *Advanced Peripheral Bus*). Este bloco contém todos os registradores necessários para a configuração e controle do EXTI, permitindo por exemplo que o desenvolvedor defina quais pinos GPIO devem gerar interrupções e como esses eventos devem ser tratados. No Manual de Referência há uma descrição detalhada de cada registrador.

Em segundo lugar, o **bloco de disparo de entrada de eventos (*Event Trigger*)** é responsável pela lógica de detecção das bordas nos sinais de entrada. Este bloco monitora as mudanças nos sinais digitais conectados aos pinos GPIO, como transições de baixo para alto (borda de subida) e de alto para baixo (borda de descida), configurando quais eventos devem gerar uma interrupção ou sinalizar uma condição específica.

Por fim, o **bloco de mascaramento (*Masking*)** se encarrega da distribuição e controle dos eventos detectados. Ele é responsável por direcionar os sinais de eventos para as diferentes saídas, como *wakeup*, interrupção e eventos gerais, além de gerenciar o mascaramento dessas saídas. Isso significa que o bloco de mascaramento pode habilitar ou desabilitar a geração de interrupções e eventos, garantindo que apenas as condições desejadas sejam processadas.

Ao configurar um pino GPIO como entrada, ele pode ser associado ao módulo EXTI para monitorar mudanças no estado do seu sinal, em níveis e pulsos. O EXTI permite que os sinais nesses pinos gerem solicitações de interrupções com base em eventos específicos, como bordas de subida (transição de baixo para alto), bordas de descida (transição de alto para baixo), ou mudanças de nível. Essas solicitações poderão ser processadas pelo NVIC para disparar interrupções, como ilustra a figura. Configurações adicionais devem ser feitas nos blocos de disparo de entrada e de mascaramento para especificar os tipos de eventos de interesse na entrada e o mascaramento a ser aplicado nos sinais de saída do EXTI.



Os registradores EXTI_FTSR1 e EXTI_RTSR1 permitem a configuração dos eventos de disparo para os sinais EXTIn. O EXTI_FTSR1 é responsável pela detecção de borda de descida, enquanto o EXTI_RTSR1 é utilizado para configurar a detecção de borda de subida. Esses registradores permitem a configuração individual dos eventos de disparo para cada um dos 15 sinais EXTIn.

20.6.2 EXTI falling trigger selection register (EXTI_FTSR1)

Address offset: 0x04

Reset value: 0x0000 0000

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.	TR21	TR20	TR19	TR18	TR17	TR16
										rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

20.6.1 EXTI rising trigger selection register (EXTI_RTSR1)

Address offset: 0x00

Reset value: 0x0000 0000

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.	TR21	TR20	TR19	TR18	TR17	TR16
										rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:22 Reserved, must be kept at reset value.

Bits 21:0 **TR[21:0]:** Rising trigger event configuration bit of configurable event input x (x= 21 to 0)

0: Rising trigger disabled (for event and Interrupt) for input line

1: Rising trigger enabled (for event and Interrupt) for input line

Note: The configurable event inputs are edge triggered, no glitch must be generated on these inputs.

If a rising edge on the configurable event input occurs while writing to the register, the associated pending bit is not set.

Rising and falling edge triggers can be set for the same configurable event input. In this case, both edges generate a trigger.

O registrador EXTI_CPUMR1 é, por sua vez, o registrador de mascaramento dos 32 eventos disparáveis, numerados de 0 a 31, entre os 89 eventos disparáveis.

20.6.18 EXTI interrupt mask register (EXTI_CPUIMR1)

Address offset: 0x80

Reset value: 0xFFC0 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MR31	MR30	MR29	MR28	MR27	MR26	MR25	MR24	MR23	MR22	MR21	MR20	MR19	MR18	MR17	MR16
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
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:22 **MR[31:22]:** CPU interrupt mask on direct event input x (x = 31 to 22)

0: Interrupt request from Line x is masked

1: Interrupt request from Line x is unmasked

Note: The reset value for direct event inputs is set to 1 in order to enable the interrupt by default.

Bits 21:0 **MR[21:0]:** CPU interrupt mask on configurable event input x (x = 21 to 0)

0: Interrupt request from Line x is masked

1: Interrupt request from Line x is unmasked

Note: The reset value for configurable event inputs is set to 0 in order to disable the interrupt by default.

Controlador de Configuração do Sistema (SYSCFG)

Observe que vários pinos de entrada de GPIOs são mapeados para um sinal de saída EXTIn. Este mapeamento é uma técnica eficiente para controlar múltiplos pinos com um único circuito. Isso permite que um conjunto de pinos de entrada seja configurado para gerar uma saída unificada, simplificando o desenho do projeto e reduzindo a quantidade de circuitos necessários. No entanto, essa flexibilidade vem com um custo adicional: a necessidade de configurar o mapeamento através de registradores adicionais, como explica o Manual de Referência: "*For the sixteen GPIO event inputs, the associated GPIO pin has to be selected in the SYSCFG_EXTICRn register. The same pin from each GPIO maps to the corresponding EXTI event input*."

Os registradores "SYSCFG_EXTICRm" do periférico SYSCFG são essenciais para configurar como os sinais dos pinos de entrada são roteados para o sinal EXTIn. É importante prestar atenção especial à sua configuração para assegurar o comportamento desejado do sistema. Embora esse processo permita uma personalização detalhada, ele requer um cuidado extra na programação e na verificação dos registradores para evitar erros de mapeamento e garantir o funcionamento correto do circuito.

No microcontrolador STM32H7A3ZIT6-Q, os pinos "n" de todas as portas Px são gerenciados por um módulo cuja saída EXTIn é um sinal multiplexado desses pinos. A multiplexação é gerida pelos quatro registradores SYSCFG_EXTICRm do periférico **SYSCFG (do inglês *System Configuration Controller*)**. Cada registrador é responsável por selecionar, através dos *bytes* 0, 1, 2 e 3, até quatro pinos [(m-1)*4:(m-1)*4+3], que são configurados como entradas para os sinais EXTI{(m-1)*4}, EXTI{(m-1)*4+1}, EXTI{(m-1)*4+2} e EXTI{(m-1)*4+3}, respectivamente. Por exemplo, os registradores SYSCFG_EXTICR1 e SYSCFG_EXTICR3, conforme descritos no Manual de Referência, controlam a seleção dos pinos [0:3] e [8:11] das portas Px para as saídas EXTI0, EXTI1, EXTI2, EXTI3, EXTI8, EXTI9, EXTI10 e EXTI11, respectivamente. Observe que cada *byte* "i" no registrador SYSCFG_EXTICRm seleciona um pino (m-1)*4+i.

12.4.2 SYSCFG external interrupt configuration register 1 (SYSCFG_EXTICR1)

Address offset: 0x08

Reset value: 0x0000 0000

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
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **EXTIx[3:0]**: EXTI x configuration (x = 0 to 3)

These bits are written by software to select the source input for the EXTI input for external interrupt / event detection.

0000: PA[x] pin
0001: PB[x] pin
0010: PC[x] pin
0011: PD[x] pin
0100: PE[x] pin
0101: PF[x] pin
0110: PG[x] pin
0111: PH[x] pin
1000: PI[x] pin
1001: PJ[x] pin
1010: PK[x] pin
Other configurations: reserved

12.4.4 SYSCFG external interrupt configuration register 3 (SYSCFG_EXTICR3)

Address offset: 0x10

Reset value: 0x0000 0000

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
EXTI11[3:0]				EXTI10[3:0]				EXTI9[3:0]				EXTI8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **EXTIx[3:0]**: EXTI x configuration (x = 8 to 11)

These bits are written by software to select the source input for the AEIC input for external interrupt / event detection.

0000: PA[x] pin
0001: PB[x] pin
0010: PC[x] pin
0011: PD[x] pin
0100: PE[x] pin
0101: PF[x] pin
0110: PG[x] pin
0111: PH[x] pin
1000: PI[x] pin
1001: PJ[x] pin
1010: PK[x] pin
Other configurations: reserved

Note: PK[11:8] are not used.