UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM **ENGENHARIA DE COMPUTAÇÃO**
SÉTIMO SEMESTRE

QXD0143 - MICROCONTROLADORES

Relatório 03: Interrupções Externas em Microcontroladores ARM

QUIXADÁ - CE
2022

471047 — PEDRO HENRIQUE **MAGALHÃES BOTELHO**

427602 — ERICK **CORREIA SILVA**

472012 — FRANCISCO ITALO DE **ANDRADE MORAES**

Relatório 03: Interrupções Externas em Microcontroladores ARM

Orientador: Prof. Dr. Thiago Werlley Bandeira da Silva

Terceiro relatório escrito para a disciplina de Microcontroladores, no curso de graduação em Engenharia de Computação, pela Universidade Federal do Ceará (UFC), campus em Quixadá.

QUIXADÁ - CE
2022

Conteúdo

Lista de Figuras

1 Introdução

Esse é o terceiro relatório da disciplina de **Microcontroladores**, ministrada pelo professor Thiago W. Bandeira. Nesse relatório é discutido sobre entrada e saída por meio de interrupções, utilizando microcontroladores ARM®, onde são apresentadas as práticas realizadas em laboratório.

Nas práticas realizadas em laboratório foi utilizada a placa de desenvolvimento **Freedom FRDM-KL25Z**, da Kinetis/NXP, com o microcontrolador **MKL25Z128VLK4**.

Este documento contém informações e imagens retiradas do manual de referência do microcontrolador usado, como mostra as figura 01:

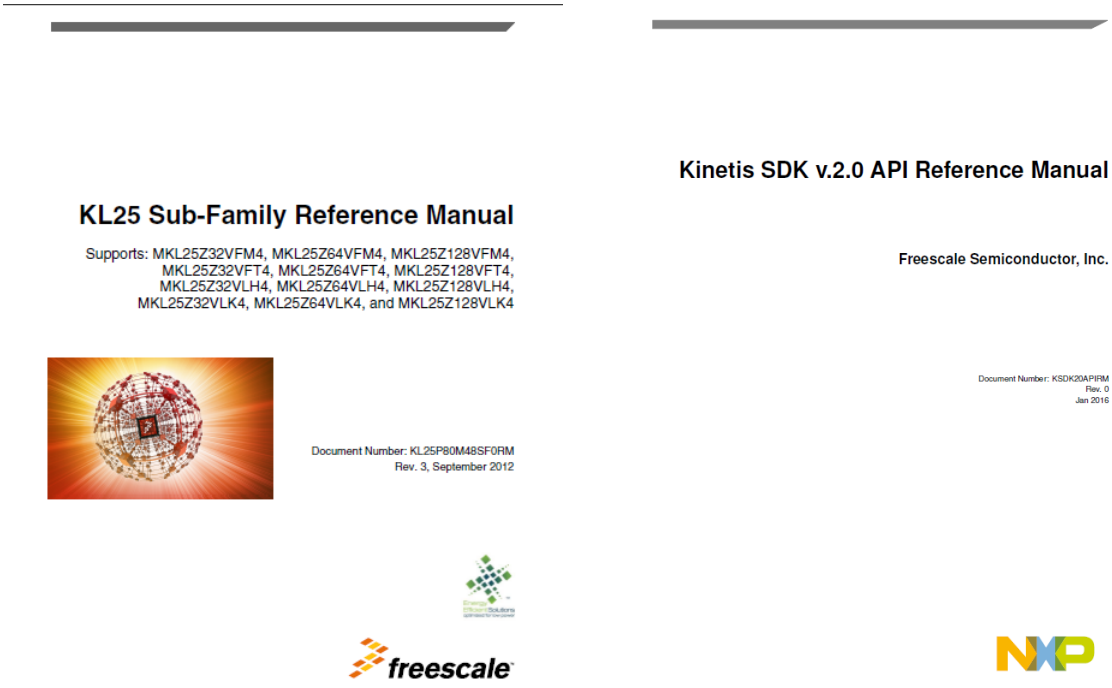

Figura 1: Capas dos Manuais de Referência do MKL25Z128VLK4 e do SDK

Também foi utilizado o manual de referência do *Software Development Kit* da Kinetis, o **SDK**, como mostra a figura 01. O SDK é uma API que fornece diversas ferramentas para a programação de microcontroladores da Kinetis/NXP.

Os projetos da disciplina estão disponíveis em um repositório no Github, em [1], onde estão os relatórios, códigos-fonte e esquemáticos (os projetos completos encontram-se comprimidos em arquivos .zip). O link também se encontra nas referências.

2 Interrupções em Microcontroladores ARM

Vamos agora discutir rapidamente sobre interrupções no ARM, para que possamos partir para a prática. Vamos, primeiramente, discorrer sobre as duas técnicas de entrada e saída mais simples: *polling* e interrupção. A partir daí iremos entrar nos detalhes de interrupção.

2.1 E/S por *Polling*

Polling é o processo em que a CPU aguarda que um dispositivo externo esteja pronto, constantemente verificando seu *status* de pronto, como uma *flag*. Somente quando a *flag* for verdadeira é que a tarefa (que precisava de alguma ação realizada no dispositivo) é realizada.

Veja no código abaixo um exemplo de *polling* em um sistema *super-loop* (que executa dentro de um laço infinito), tipo de sistema embarcado mais simples.

```
1 while(1) {
2     if(dispositivoPronto()) realizarTarefa();
3 }
```

Também chamado de "espera ocupada", já que as vezes é necessário que a CPU espere que o dispositivo esteja pronto. Nesta situação, quando uma operação de E/S é necessária, o computador não faz nada além de verificar o status da E/S do dispositivo até que esteja pronto, consumindo seu processamento, já que estará esperando. Em um sistema simples, a espera ocupada é aceitável se nenhuma ação for possível/necessária até o acesso de E/S.

```
1 while(1) {
2     while(!dispositivoPronto());
3     realizarTarefa();
4 }
```

O *polling* tem a desvantagem de que, se houver muitos dispositivos para verificar, e portanto muitas tarefas para realizar, uma tarefa pode demorar mais a ser executada, ficando dependente do tempo empregado em uma tarefa anterior a ela. Outro problema é o sincronismo entre a verificação do dispositivo e o dispositivo estar pronto naquele exato momento.

No exemplo abaixo a tarefa 03 precisa esperar a verificação do dispositivo 02 terminar e a tarefa 02 executar, no **melhor caso**, o que pode ser um grande problema.

```
1 while(1) {
2     if(dispositivo01Pronto()) realizarTarefa01();
3     while(!dispositivo02Pronto());
4     realizarTarefa02();
5     if(dispositivo03Pronto()) realizarTarefa03();
6 }
```

Isso pode ocorrer com a verificação de um botão, por exemplo. Se o botão for pressionado rapidamente, porém o sistema não verificou o estado do pino nesse exato momento a tarefa correspondente não será executada.

2.2 E/S por Interrupção

Outra forma de gerenciar entradas e saídas em um sistema é por meio de **interrupção**, na qual um periférico que precisa realizar uma transferência de dados (mandar ou receber dados) notifica a CPU quando estiver pronto para comunicação, e solicita a atenção do processador para realizar algum processamento de dados. A CPU então pára o que está fazendo para realizar uma tarefa e volta para o que estava fazendo.

Os passos básicos de uma interrução são descritos abaixo:

1. Dispositivo manda um sinal de interrupção à CPU, o **interrupt request**, ou IRQ.
2. CPU interrompe a tarefa atual e chama uma função que irá tratar a interrupção.
3. A função executa, ao final, o fluxo de execução retoma de onde parou.
4. O sistema continua sua execução normalmente.

A função que irá tratar a interrupção se chama ISR (Interrupt Service Routine), ou rotina de serviço de interrupção, uma rotina de software que contém instruções que devem ser efetuadas todas as vezes que a interrupção for lançada. "Tratar uma interrupção" significa realizar uma tarefa que deve ser executada quando a interrupção for efetuada, não deixando o sinal IRQ "passar despercebido" pela CPU.

Quando a IRQ é atendida a ISR é invocada, não estando ela no código do programa em si, mas em um local especial da memória, que é acessado todas as vezes que uma interrupção é lançada, a **tabela de vetores**. Falaremos dela mais adiante. Temos então o código dividido em duas partes:

- **Laço de Background**: A função principal, com o laço infinito, que realiza as mesmas atividades constantemente.
- **Função de Foreground**: A ISR, uma função chamada somente quando a interrupção é lançada. Interrompe o fluxo da *main* para ser executada e volta à execução normal após sua execução.

Uma interrupção pode ser acionada internamente a partir do microprocessador (as exceções) ou externamente (por um periférico interno do microcontrolador ou pelos pinos). A interrupção alerta a CPU para uma ocorrência, como um evento baseado em tempo (um determinado período de tempo decorreu ou um tempo específico foi atingido, por exemplo), uma mudança de estado ou o início ou fim de um processo.

2.3 Interrupções Externas do Microcontrolador

Focaremos nesse relatório em interrupções geradas nos pinos do microcontrolador. Os pinos são controlados pelo periférico **PORTx**, portanto será por meio dele que realizaremos o manejo de interrupções.

Quando a **IRQ** é lançada é ativada uma *flag* de interrupção, a **ISF** (Interrupt Status Flag). Essa *flag* é ativada toda vez que é detectado um **evento de interrupção**. O sinal IRQ pode ser configurado para ser enviado assim que o evento de interrupção for detectado, sendo ele uma característica pré-definida do sinal de entrada no pino, como uma borda de subida, descida, ambas, nível lógico alto ou baixo.

Por exemplo: se o pino **PTB1** for configurado para receber interrupções na forma de borda de subida, todo as bordas de subida do sinal elétrico serão entendidas como um sinal IRQ, habilitando a ISF.

A configuração do evento de interrupção é feito modificando os bits **IRQC** (Interrupt Request Config) do registrador PCR, como mostra a figura 02. É também no registrador PCR que temos acesso à *flag* ISF. Alternativamente, temos um registrador ISFR, do PORTx, onde cada bit corresponde a um pino. Se o bit n estiver 1 é porque o pino n gerou uma interrupção

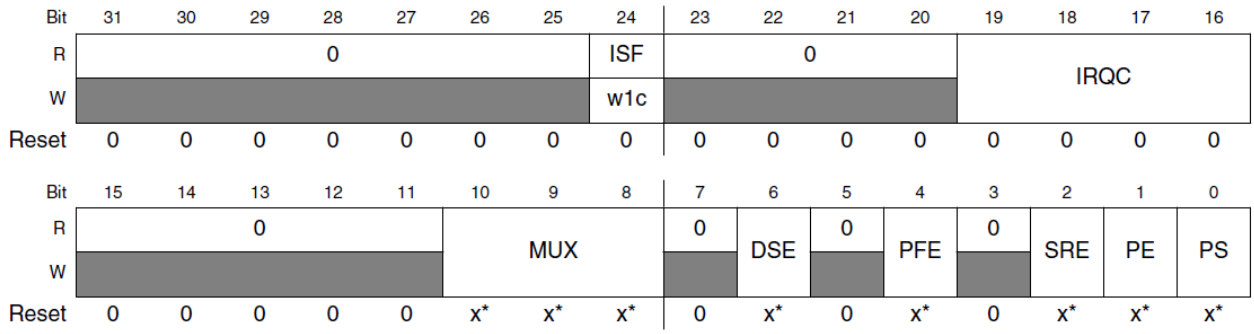

Figura 2: Registrador de Controle de Pino (PCR)

As opções para definição do evento de interrupção são descritas abaixo.

- 1000 - Interrupt when logic zero.
- 1001 - Interrupt on rising edge.
- 1010 - Interrupt on falling edge.
- 1011 - Interrupt on either edge.
- 1100 - Interrupt when logic one.

Obviamente, para configurar o pino por meio do periférico PORT deve-se habilitar o clock para ele, usando o registrador SCGC5, e, para que o periférico PORT "escute" o evento de interrupção o pino deve estar configurado como entrada, no registrador PDDR.

2.4 O Controlador de Interrupções NVIC

Como o microcontrolador ARM tem muitas fontes de interrupção temos um módulo acoplado ao núcleo que controla todas as IRQ: o **NVIC (Nested Vector Interrupt Controller)**. O NVIC, a grosso modo, prioriza e aninha interrupções. Podemos ter, então, tratamento de interrupções baseado em **prioridades**, e interrupções **aninhadas**, o que nos permite um maior grau de controle de interrupções.

2.4.1 Habilitando Interrupções Globais

O NVIC nos permite 48 fontes de interrupções, sendo 16 internas ao núcleo ARM e 32 periféricos externos (PORT, Timers, ADC, etc). É no NVIC que habilitamos a interrupção global do periférico (umas das 32), para que a IRQ possa ser tratada pelo NVIC. Podemos habilitar isso de duas formas: usando uma função ou no próprio registrador do NVIC. As exceções internas do núcleo ARM não precisam ser habilitadas manualmente no NVIC.

O NVIC tem alguns registradores de controle. Entre eles temos o ISER (Interrupt Set Enable Register), onde podemos habilitar uma interrupção correspondente ao pino. Por exemplo, parar habilitarmos uma interrupção para o PORTD fazemos:

```
1 NVIC->ISER[0] |= (1 << 31);
```

Podemos também usar a função NVIC_EnableIRQ, passando o número da IRQ global por meio de uma constante da biblioteca "core_cm0plus.h':

```
1 NVIC_EnableIRQ(PORTD_IRQn);
```

O NVIC, para sinalizar à CPU que uma interrupção deve ser tratada o informa por meio do **PPB** (*Private Peripheral Bus*, que liga os componentes próximos ao núcleo), retirando o trabalho de gerenciar os sinais IRQ do núcleo e reduzindo a latência da interrupção. A comunicação entre o NVIC, os periféricos do microcontrolador e o núcleo ARM são evidenciados na figura 03.

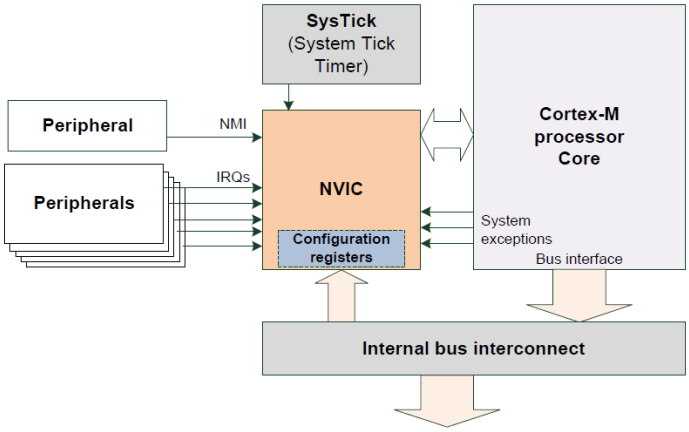

Figura 3: Comunicação entre o NVIC, os periféricos e a CPU

2.4.2 Aninhamento e Priorização de Interrupções

O NVIC também pode lidar com interrupções que ocorrem quando outras interrupções estão sendo tratadas ou quando a CPU está restaurando seu estado anterior e retomando de onde parou. Chamamos essas múltiplas interrupções sendo tratadas (e esperando para serem tratadas) no NVIC de **interrupções aninhadas**. O termo "aninhado" refere-se ao fato de que após uma interrupção terminar a CPU pode ter que tratar outra sem retornar ao fluxo principal.

O NVIC, além de gerenciar as IRQ que chegam também prioriza interrupções, ou seja, estabelece um critério de priorização de requisições onde uma interrupção pode ser tratada antes de outra por ter maior prioridade. No NVIC, um número de interrupções pode ser definido (48 no caso), e cada interrupção recebe uma prioridade, sendo "0" a prioridade mais alta. Além disso, a interrupção mais crítica pode se tornar não mascarável (**NMI**), o que significa que não pode ser desabilitada (mascarada). Apenas três interrupções tem prioridades pré-definidas: **Reset** (-3), **NMI** (-2) e **Hard Fault** (-1), como mostra a figura 04, onde podemos ver as exceções do núcleo ARM, assim como as IRQ externas.

Exception	Vector	Core	Priority
1	Reset	M0/M4/M3/M7	-3
2	NMI	M0/M4/M3/M7	-2
3	HardFault	M0/M3/M4/M7	-1
4	MemManageFault	M3/M4/M7	User
5	BusFault	M3/M4/M7	User
6	UsageFault	M3/M4/M7	User
7-A	Reserved		
B	SVCall	M0/M3/M4/M7	User
C	Debug Monitor	M3/M4/M7	User
D	Reserved		
E	PendSV	M0/MM3/M4/M7	User
F	SysTick	M0(optional)/M3/M4/M7	User
10-...	IRQ0, IRQ1, ... (Vendor)	M0: 0x10-0x47 M4/M7: IRQ0-239	User

Figura 4: Exceções/Interrupções do ARM e seus níveis de prioridade.

Uma função do NVIC é garantir que as interrupções de prioridade mais alta sejam concluídas antes das interrupções de prioridade mais baixa, mesmo que a interrupção de prioridade mais baixa seja acionada primeiro. Por exemplo, se uma interrupção de prioridade mais baixa estiver sendo tratada e ocorrer uma interrupção de prioridade mais alta, a CPU interromperá a interrupção de prioridade mais baixa e processará a de prioridade mais alta primeiro.

Da mesma forma, um esquema de tratamento conhecido como **tail-chaining** especifica que, se uma interrupção estiver pendente enquanto o ISR para outra interrupção de maior prioridade for concluída, o processador iniciará imediatamente o ISR para a próxima interrupção,

sem restaurar seu estado anterior. Esse processo é evidenciado na figura 05, onde temos uma ISR de menor prioridade e uma de maior prioridade, estando elas aninhadas.

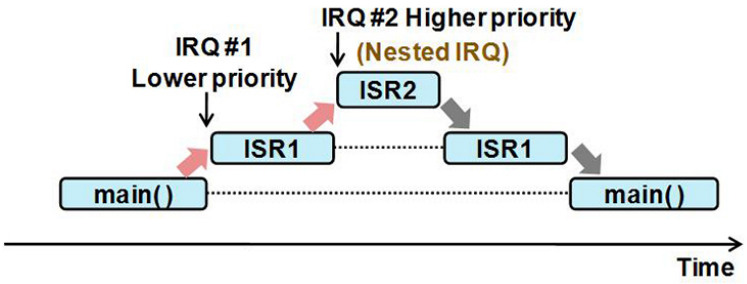

Figura 5: Aninhamento entre duas interrupções de diferentes prioridades.

Os esquemas de controle, priorização e aninhamento do NVIC, então, reduzem a latência e a sobrecarga que as interrupções normalmente introduzem à CPU e garantem um baixo consumo de energia, mesmo com alta carga de interrupções no controlador.

2.5 Vetores de Interrupção

Resumindo o que vimos até agora: Quando ocorre uma interrupção, um sinal de interrupção é gerado, o IRQ, o que faz com que a CPU pare sua operação atual, salve seu estado atual (seu contexto) e inicie a rotina de tratamento da interrupção, a ISR, associado à interrupção lançada. Quando o processamento da interrupção é concluído, a CPU restaura seu estado anterior e retoma de onde parou. Mas como são invocadas as ISR na memória, já que não são chamadas no programa principal?

As ISR ficam "guardadas" em um espaço reservado, no começo da memória de programa: na **Tabela de Vetores de Interrupção**. Um vetor de interrupção é o endereço de uma ISR, que é executada pela CPU quando ocorre uma interrupção. O NVIC, então, usa uma tabela na memória que contém os endereços das ISR (que estão em algum lugar do programa) para cada interrupção. Logo, cada vetor corresponde a uma ISR. Quando uma interrupção é acionada, o processador obtém o endereço da tabela de interrupção, o coloca o PC, salva o contexto atual, assim como o endereço de retorno, e salta para o começo da rotina.

Podemos generalizar exceções e interrupções como **exceções**. Podemos diferencia-las da seguinte forma: exceções sempre são resultado direto de instruções, sendo um evento síncrono, e a interrupção um evento assíncrono. Cada exceção recebe um número exclusivo: 0, 1, 2 e assim por diante. Quando ocorre uma exceção, o hardware coloca o número da exceção em um registrador, que é usado o número como o índice em um vetor de exceção, sendo multiplicando por 4, já que cada endereço no ARM possui 32-bits, como mostra a figura 06.

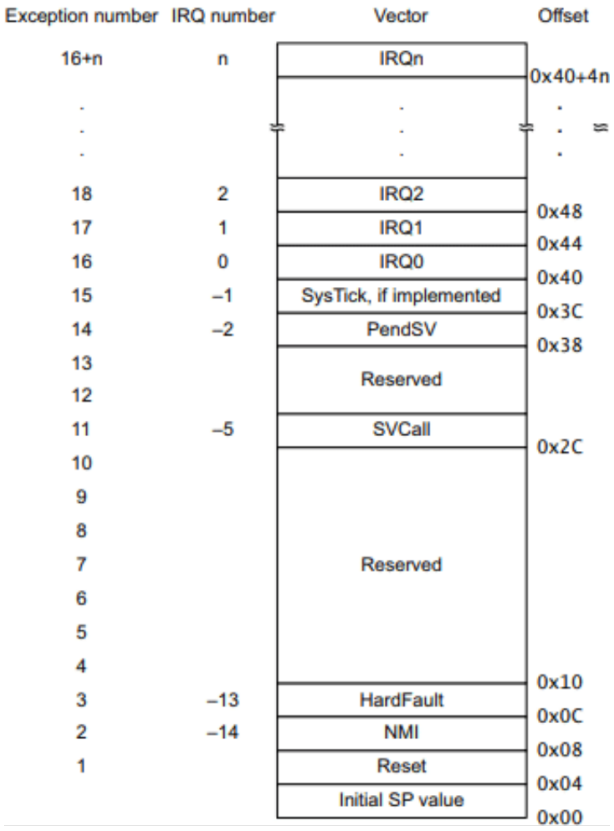

Figura 6: Tabela de Vetores de Interrupção do ARM.

Perceba que o número da IRQ é negativo quando é interno ao núcleo ARM, por questões de representação, mas o número da exceção é geral ao processador.

Uma pequena diferença ocorre entre a forma como o hardware do processador trata interrupções e exceções. Pensamos em uma interrupção como ocorrendo entre dois instruções, já que a CPU só verifica por interrupções ao final do ciclo de execução de uma instrução. Assim, a instrução foi concluída e a próxima instrução não começou. No entanto, ocorre uma exceção durante uma instrução. Assim, quando o processador retorna da exceção, o contador de programa não avançou e a instrução pode ser reiniciada. A reinicialização é especialmente importante para exceções de falha de página, por exemplo. Quando um ocorre uma falha de página, o sistema operacional deve ler a página ausente do disco para memória, definir a tabela de páginas e, em seguida, executar novamente a instrução que causou a falha.

2.6 Configuração da Interrupção Externa

A configuração de uma interrupção externa ao núcleo ARM se dá por alguns poucos passos, que iremos evidenciar abaixo. Iremos, aqui, mostrar a configuração de uma interrupção externa ao microcontrolador, pelos pinos.

2.6.1 Definir o Evento de Interrupção

Como dito anteriormente, devemos definir qual o evento que irá acionar o IRQ. Isso é feito no registrador **PCR**, no bit IRQC, um campo de bits dedicado aos parâmetros de interrupção. O campo vai do bit 16 ao 19 e pode-se configurar os seguintes eventos de interrupção:

- 0000 - Interrupção Desabilitada (Padrão).
- 1000 - Nível Lógico Baixo.
- 1001 - Borda de Subida.
- 1010 - Borda de Descida.
- 1011 - Ambas as Bordas.
- 1100 - Nível Lógico Alto.

A escolha de qual evento usar fica à escolha do projetista da aplicação. Para habilitarmos interrupção em ambas a bordas nos pinos PTA12 e PTD4 fazemos o seguinte:

```
1 PORTA->PCR[12] |= (0b1011U << 16);
2 PORTD->PCR[4]  |= (0b1011U << 16);
```

2.6.2 Habilitar a Interrupção Global no NVIC

Como dito anteriormente devemos habilitar a interrupção global, para interrupções externas ao processador, para que o NVIC possa as tratar. Devemos, para tanto, utilizar ou a função do CMSIS, ou o registrador **ISER** do NVIC. Iremos habilitar as interrupções para PORTA e PORTD, onde estão localizados os botões.

Podemos fazer isso usando o registrador Interrupt Set Enable Register (ISER). Dessa forma precisamos ativar os bits referentes ao número da IRQ referente a periférico, no caso, PORTA é IRQ30 e PORTD é IRQ31 (os únicos periféricos PORTx que aceitam interrupção):

```
1 NVIC->ISER[0] |= (1 << 30) | (1 << 31);
```

Ou usando a função pré-definida na biblioteca CMSIS:

```
1 NVIC_EnableIRQ(PORTA_IRQn);
2 NVIC_EnableIRQ(PORTD_IRQn);
```

2.6.3 Definir a Prioridade da Interrupção

Podemos também definir a prioridade da interrupção, o qual é uma etapa opcional, já que por padrão todas as interrupções externas tem prioridade máxima. Temos quatro níveis de prioridade: de 3 (menor) a 0 (maior). A priorização por software não afeta **reset**, **hard fault** e **NMI**, já que sempre terão maior prioridade que todas as outras IRQ.

O critério de desempate para duas interrupções simultâneas pendentes com mesma prioridade é que a interrupção com o menor número IRQ será executada.

Podemos definir a prioridade usando a função **NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)** da biblioteca CMSIS, no arquivo **core_cm0plus.h**, como mostra o código abaixo:

```
1 NVIC_SetPriority(PORTA_IRQn, 0);
2 NVIC_SetPriority(PORTD_IRQn, 1);
```

Dessa forma, habilitamos PORTA com uma prioridade maior que PORTD, podendo então uma interrupção provinda de PORTD interromper o tratamento de uma exceção provinda de PORTA.

Alternativamente, podemos usar os registradores de prioridade de interrupção, IP[0] ao IP[7], do NVIC para definir as prioridades. Os registradores IP[m] são divididos em 4 partes cada, de 8-bits cada. Essas partes são os campos de bits destinados a definir a prioridade de uma IRQn. Ou seja, a prioridade de cada IRQn é definida usando um campo de bits exclusivo dos registradores IP[m]. Para descobrirmos qual IP[m] usamos para uma IRQn, e quais bits usamos, podemos usar a seguinte fórmula:

IRQn: IP[**floor(n/4)**], bits **8*(n%4)** a **8*(n%4 + 1) - 1**.

Vejamos alguns exemplos:

- IRQ0: IP[0], bits 0 a 7.
- IRQ1: IP[0], bits 8 a 15.
- IRQ4: IP[1], bits 0 a 7.
- IRQ30: IP[7], bits 16 a 23.

Ou, alternativamente, podemos usar a tabela abaixo:

Registrador	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
IP[0]	IRQ3	IRQ2	IRQ1	IRQ0
IP[1]	IRQ7	IRQ6	IRQ5	IRQ4
IP[2]	IRQ11	IRQ10	IRQ9	IRQ8
IP[3]	IRQ15	IRQ14	IRQ13	IRQ12
IP[4]	IRQ19	IRQ18	IRQ17	IRQ116
IP[5]	IRQ23	IRQ22	IRQ21	IRQ20
IP[6]	IRQ27	IRQ26	IRQ25	IRQ24
IP[7]	IRQ31	IRQ30	IRQ29	IRQ28

Portanto, para definirmos as prioridades das interrupções em PORTA (IRQ30) e PORTD (IRQ31), usando os registradores do NVIC, usamos:

```
NVIC->IP[7] &= ~(0b11111111U << 16); // Prioridade 0 (Maxima) para PORTA
NVIC->IP[7] |=  (0b00000001U << 24); // Prioridade 1 para PORTD
```

Não iremos definir prioridades no código, ficando PORTA e PORTD com prioridades iguais.

2.6.4 Definir a Rotina de Tratamento (ISR)

Devemos, ao final, definir a ISR que irá tratar a interrupção. A ISR será programada por nós no código do programa, e seu endereço será colocado em um lugar reservado da memória, a tabela de vetores de interrupção. O compilador irá atribuir o vetor na tabela de forma transparente. Portanto, iremos definir uma ISR no código e o o compilador irá atribuir o endereço dessa ISR ao seu vetor correspondente.

Para definirmos uma ISR devemos implementar uma função já pré-definida, declaradas no arquivo de cabeçalho **startup_<mcu>.h**. O compilador irá compilar essa função e colocar seu endereço na tabela de vetores. O modelo da ISR sempre será: **void <PERIFERICO>_IRQHandler(void)**.

Por exemplo: a rotina de interrupção para o PORTA se chama: **void PORTA_IRQHandler(void)**.

No arquivo de cabeçalho todas as ISR estão definidas com declaração fraca (palavra-chave **weak**). Isso quer dizer que tem uma implementação (também definida como weak) daquela função que será usada caso nenhuma outra implementação seja feita (implementação forte, padrão). Na implementação fraca temos um laço infinito, somente. Portanto todas as vezes que queremos usar uma interrupção devemos programar a ISR, que irá realizar uma tarefa todas as vezes que a interrupção for realizada. Caso nenhuma ISR seja definida pelo programador a ISR padrão será atribuída ao vetor, e caso ocorra uma interrupção o processador ficará preso em um laço infinito. Veja abaixo um exemplo da implementação de ISR:

```
void PORTA_IRQHandler(void) {
    // TAREFA DA INTERRUPCAO PARA O PERIFERICO PORTA
}
```

2.7 Tratamento da Interrupção Externa

Quando uma interrupção é detectada no periférico PORT, é ativada a *flag* de interrupção do pino onde ocorreu o evento: o bit **ISF** do registrador PCR, ou no bit correspondente no registrador **ISFR**, ambos do periférico PORTx. Quando o NVIC detecta uma interrupção, indicada por meio e uma *flag* de interrupção pendente do periférico, uma flag de pendente no registrador ISPR (Interrupt Set Pending Register) no NVIC é ativada.

Quando o NVIC invoca a ISR para tratar tal interrupção a flag de pendente no NVIC é limpa usando o registrador ICPR (Interrupt Clear Pending Register), ou seja, a *flag* de pendente do NVIC é limpa automaticamente. Porém, o mesmo não pode ser dito para o periférico. A *flag* de pendente do periférico, no caso PORT, deve ser limpa manualmente ao tratar a interrupção. Caso contrário, a ISR será chamada novamente, já que ainda aparecerá como se a interrupção estivesse pendente. Dessa forma, o programa nunca continuará normalmente.

Portanto, ao início da ISR devemos verificar em qual pino ocorreu a interrupção, checando o registrador ISFR, ou o bit ISF do registrador PCR, para que possamos fazer a tarefa correspondente ao pino, já que um periférico PORT tem vários pinos, e apenas uma ISR. Podemos ter vários pinos configurados para interrupção, e, para cada um deles, uma tarefa diferente. E, ao final, devemos limpar a *flag*, atribuindo nível lógico alto a ela. Vejamos abaixo um exemplo de ISR para PORTD, em que, ocorrido uma interrupção no pino PTD4 devemos mudar o estado de um LED em PTC8:

```
1 void PORTD_IRQHandler(void) {
2     if(PORTD->ISFR & (1U << 4)) {    // Verificar se a interrupcao foi em PTD4
3         GPIOC->PTOR |= (1U << 8);    // Trocar o nivel do LED
4         PORTD->ISFR |= (1U << 4);    // Escrever no bit para limpa-lo
5     }
6 }
```

Podemos, então, resumir todos os passos de uma interrupção abaixo:

1. Dispositivo manda um sinal de interrupção à CPU. Esse processo é dividido em algumas partes:
   (a) Dispositivo manda um sinal no pino.
   (b) O periférico PORTx, que controla o pino, manda um sinal para o NVIC, o controlador de interrupções.
   (c) O NVIC irá gerenciar a interrupção baseado em sua prioridade, e notificar a CPU quando for cabível.
2. CPU interrompe a tarefa atual e chama uma função que irá tratar a interrupção.
3. A função executa, ao final, o fluxo de execução retoma de onde parou.
4. O sistema continua sua execução normalmente.

3 Acionamento de LED usando Botão com Interrupção

A atividade prática acerca do assunto de interrupção consiste em fazer que determinado LED acenda, ou apague, ao toque de um botão. A utilização de interrupções se torna muito útil quando devemos criar aplicações que recebem sinais de forma aleatória, ou seja, os sinais os quais o processador deve trabalhar não são recebidos em tempo determinado, e, para que o sinal seja tratado assim que recebido, faz-se necessário o recurso de interrupção.

Um exemplo palpável é uma aplicação que executa uma certa sequência de acendimento de LEDs ao toque de um botão. Enquanto a CPU está executando a sequência de LEDs, ela não estará verificando se um novo sinal, no caso, o pressionamento do botão, foi recebido, e assim ele será perdido, se utilizarmos *polling*. Uma solução, então, é utilizarmos interrupção, onde a CPU será avisada que o botão foi pressionado, sem precisar ficar verificando.

3.1 Código usando Registradores de Controle

Abaixo está o código para a aplicação exigida na prática, onde dois botões controlam o acionamento de dois LEDs:

```
#include "Switch_Led_Interrupt.h"

#include "MKL25Z4.h"

int main(void) {
    __disable_irq();
    init_pins();
    __enable_irq();
    while(1) {

    }
}

void init_pins(void) {
  // Enable Clock for PORTA, PORTC and PORTD
  SIM->SCGC5 |= (1U << 9) | (1U << 11) | (1U << 12);

  // Set GPIO Function for the pins
  SW1_PORT->PCR[SW1_PIN] |= (0b001U << 8);
  SW2_PORT->PCR[SW2_PIN] |= (0b001U << 8);
  LED_D1_PORT->PCR[LED_D1_PIN] |= (0b001U << 8);
  LED_D3_PORT->PCR[LED_D3_PIN] |= (0b001U << 8);

  // Initialize buttons as input
  SW1_GPIO->PDDR &= ~SW1_MASK;
  SW2_GPIO->PDDR &= ~SW2_MASK;

  // Initialize LEDs as output
  LED_D1_GPIO->PDDR |= LED_D1_MASK;
```

```
30    LED_D3_GPIO->PDDR |= LED_D3_MASK;
31
32    // Config Interrupt for the Buttons (Interrupt in Either Edge)
33    SW1_PORT->PCR[SW1_PIN] |= (0b1011U << 16);
34    SW2_PORT->PCR[SW2_PIN] |= (0b1011U << 16);
35
36    // Enable Interrupt for PORTA and PORTD
37    NVIC->ISER[0] |= (1U << 30) | (1U << 31);
38  }
39
40  /*
41   * ISR for PORTA peripheral. Checks if PTA12 is high.
42   * If so, toggle the level at PTA13.
43   */
44  void PORTA_IRQHandler(void) {
45    // Checks if the interrupt was generated by SW2
46    if(SW2_PORT->ISFR & SW2_MASK) {
47      LED_D3_GPIO->PTOR |= LED_D3_MASK;
48
49      // Clear the Interrupt Flag
50      SW2_PORT->ISFR |= SW2_MASK;
51    }
52  }
53
54  /*
55   * ISR for PORTD peripheral. Checks if PTD4 is high.
56   * If so, toggle the level at PTC8.
57   */
58  void PORTD_IRQHandler(void) {
59    // Checks if the interrupt was generated by SW1
60    if(SW1_PORT->ISFR & SW1_MASK) {
61      LED_D1_GPIO->PTOR |= LED_D1_MASK;
62
63      // Clear the Interrupt Flag
64      SW1_PORT->ISFR |= SW1_MASK;
65    }
66  }
```

Vê-se que o sistema consiste de um laço infinito para execução de qualquer código - **laço de background** - e um modelo de função que executa uma série de passos programados quando uma IRQ é disparada - **função de foreground**.

Examinando a função **PORTx_IRQHandler()** vemos que ela realiza os seguintes passos:

- **Verifica se o sinal detectado veio do pino do botão**: O sinal que se espera vem do pino do botão, podendo ser detectado caso a flag **ISFR** - responsável por indicar uma interrupção pendente - apresentar sinal lógico alto. Devemos, então, verificar se a interrupção veio do pino do botão verificando se a interrupção pendente no registrador ISFR corresponde ao pino do botão.
- **Inversão do sinal**: O registrador **PTOR** - Port Toggle Output Register - é responsável

por inverter o sinal de saída, sendo essa instrução executada apenas se a interrupção for detectada.

- **Limpa a flag de interrupção**: a flag de interrupção deve ser limpa manualmente sempre ao final da ISR, caso contrário as instruções que devem ser executadas caso uma interrupção for detectada serão executadas independentemente de haver interrupção ou não, já que a flag de pendente sempre está setada!

Examinemos, agora, a função **init_pins()**:

- **Habilitação do *clock* para os módulos de GPIO:** usamos o registrador SCGC5, do módulo SIM, para habilitar o *clock* para os módulos de GPIO, neste caso, para os módulos GPIOA (bit 9), GPIOC (bit 11) e GPIOD (bit 12).
- **Multiplexação dos pinos:** Como queremos usar a função de **GPIO** dos pinos devemos configurar o multiplexador de função de todos os pinos que serão usados para GPIO (Alt. 1, representado por 0b001). Podemos fazer isso pelo registrador *PCR*, configurando os bits 8, 9 e 10, referentes ao MUX.
- **Selecionar a função de entrada aos pinos dos botões**: O valor zero na posição do bit relacionado ao pino no registrador *PDDR* é associado à função de entrada do pino. Por isso, limpamos o bit correspondente aos pinos dos botões: **PTA12** e **PTD4**.
- **Selecionar a função de saída aos pinos dos LEDs**: De forma semelhante, o valor 1 no bit correspondente ao pino no registrador *PDDR* é associado à função de saída. Portanto ativamos os bits correspondentes aos pinos dos LEDs: **PTA13** e **PTC8**.
- **Configuração da interrupção nos pinos de entrada**: no registrador *PCR* configuramos qual o evento que dispara uma interrupção. No nosso caso, utilizamos o evento de detecção de ambas as bordas (subida e descida).
- **Habilitar interrupção global das PORTA e PORTD**: o último passo é habilitar a interrupção global de PORTA e PORTD, para que o NVIC possa gerenciar e repassar à CPU as interrupções ocorridas nesses periféricos.

Usando a função __disable_irq() antes da função de inicialização desabilita as interrupções durante a inicialização dos componentes do sistema, e __enable_irq() habilita as interrupções no decorrer do programa. Isso serve para que a inicialização dos componentes do sistema não seja interrompida, podendo levar a resultados indesejados.

3.2 Código usando SDK API

Vamos agora examinar o código que realiza as mesmas funções que o anterior, porém, dessa vez usando a API da NXP, o SDK, como mostra o código abaixo:

```c
#include "SW_LED_IRQHandler.h"

#include "MKL25Z4.h"

#include "fsl_clock.h"
#include "fsl_port.h"
#include "fsl_gpio.h"

int main(void) {
    __disable_irq();
    init_pins();
    __enable_irq();
    while(1) {

    }
}

void init_pins(void) {
  // Enable Clock
  CLOCK_EnableClock(kCLOCK_PortA);
  CLOCK_EnableClock(kCLOCK_PortC);
  CLOCK_EnableClock(kCLOCK_PortD);
  CLOCK_EnableClock(kCLOCK_PortE);

  // Set IO Function
  PORT_SetPinMux(SW1_PORT, SW1_PIN,    kPORT_MuxAsGpio);
  PORT_SetPinMux(SW2_PORT, SW2_PIN,    kPORT_MuxAsGpio);
  PORT_SetPinMux(LED_D1_PORT, LED_D1_PIN, kPORT_MuxAsGpio);
  PORT_SetPinMux(LED_D3_PORT, LED_D3_PIN, kPORT_MuxAsGpio);

  // Configuration variables
  const gpio_pin_config_t sw1_config    = {kGPIO_DigitalInput,  0};
  const gpio_pin_config_t sw2_config    = {kGPIO_DigitalInput,  0};
  const gpio_pin_config_t led_d1_config = {kGPIO_DigitalOutput, 0};
  const gpio_pin_config_t led_d3_config = {kGPIO_DigitalOutput, 0};

  // Initialize Pins
  GPIO_PinInit(SW1_GPIO, SW1_PIN,    &sw1_config);
  GPIO_PinInit(SW2_GPIO, SW2_PIN,    &sw2_config);
  GPIO_PinInit(LED_D1_GPIO, LED_D1_PIN, &led_d1_config);
  GPIO_PinInit(LED_D3_GPIO, LED_D3_PIN, &led_d3_config);

  // Config Interrupt for SW2
  PORT_SetPinInterruptConfig(SW2_PORT, SW2_PIN, kPORT_InterruptEitherEdge);
  NVIC_EnableIRQ(PORTA_IRQn);

  PORT_SetPinInterruptConfig(SW1_PORT, SW1_PIN, kPORT_InterruptEitherEdge);
  NVIC_EnableIRQ(PORTD_IRQn);
}

void PORTA_IRQHandler(void) {
  if(BitTst(PORT_GetPinsInterruptFlags(SW2_PORT), SW2_PIN)) {
    PORT_ClearPinsInterruptFlags(SW2_PORT, SW2_MASK);
    GPIO_TogglePinsOutput(LED_D3_GPIO, LED_D3_MASK);
```

```
55    }
56  }
57
58  void PORTD_IRQHandler(void) {
59    if(BitTst(PORT_GetPinsInterruptFlags(SW1_PORT), SW1_PIN)) {
60      PORT_ClearPinsInterruptFlags(SW1_PORT, SW1_MASK);
61      GPIO_TogglePinsOutput(LED_D1_GPIO, LED_D1_MASK);
62    }
63  }
```

A aplicação, novamente, consiste apenas da função **init_pins()**, já que o ato de piscar os LEDs vai ocorrer na duas ISRs. Assim, pode-se escrever outros códigos referentes à outras funções no laço infinito.

Em resumo, a função **init_pins()**, a qual se resume todo o código, performa os seguintes passos:

- **Habilitar do clock para os módulos GPIO**: Por meio da função do SDK de habilitar o *clock*, **CLOCK_EnableClock()**, e das macros **kCLOCK_PortX**, que possuem os identificadores dos periféricos PORTx para habilitar os *clocks*.
- **Multiplexar os pinos**: A função **PORT_SetPinMux()** é usada para configurar a função que será exercida pelo pino, ou seja, GPIO (Alt. 1), definida pela macro $kPORT\_MuxAsGpio$. No código, são cofigurados os pinos que vão receber sinais dos botões, PTA12 e PTD4, e os que vão emitir sinais para o exterior, os quais serão visíveis pelos LEDs, PTA13 e PTC8.
- **Inicialização dos pinos**: Usando variáveis do tipo **gpio_config_t** para configurar a direção de cada pino. Se for saída ($kGPIO\_DigitalOutput$) ou de entrada ($kGPIO\_DigitalInput$). Os pinos são inicializados usando a função $GPIO\_PinInit()$.
- **Configurando a interrupção**: A função **PORT_SetPinInterruptConfig()** é usada para associar a porta, o pino da porta, e o tipo o tipo de evento a ser usado para disparar o sinal de interrupção. No código, foram configurados os pinos para que o evento de interrupção seja em ambas as bordas (borda de subida e de descida) do sinal recebido pelos pinos, definida pela macro **kPORT_InterruptEitherEdge**. A função **NVIC_EnableIRQ()** habilita a interrupção global no NVIC para as portas por meio de constantes, que representam o número de IRQ. Como a placa em que estamos trabalhando tem suporte para interrupção apenas nos periféricos PORTA e PORTD, essa função recebe as macros referentes apenas a esses dois: **PORTA_IRQn** e **PORTD_IRQn**.

E, por último, nas rotinas de interrupção estão a lógica para piscar o LED. É verificado se a interrupção foi no pino do botão (ou seja, irá verificar pelo bit respectivo em **ISFR**) por meio da função **PORT_GetInterruptStatusFlag()**. Se sim, irá realizar a rotina de interrupção, limpando a *flag* manualmente com a função **PORT_ClearInterruptFlags()**. O sinal do pino é, então, invertido usando-se a função **GPIO_TogglePinsOutput()**.

3.3 Arquivo de Cabeçalho das Práticas

A seguir é mostrado o arquivo de cabeçalho utilizados nas práticas, contendo a declaração de funções implementadas, assim como macros utilizados.

```
#ifndef SW_LED_IRQHANDLER_H
#define SW_LED_IRQHANDLER_H

// Checks if the bit is high in value.
#define BitTst(value, bit) ((value) & (1 << bit))

// LED Interface pins.
#define LED_D1_PIN 8       // PTC8
#define LED_D3_PIN 13      // PTA13

#define LED_D1_PORT PORTC
#define LED_D3_PORT PORTA

#define LED_D1_GPIO GPIOC
#define LED_D3_GPIO GPIOA

// LED Interface masks.
#define LED_D1_MASK 1 << LED_D1_PIN
#define LED_D3_MASK 1 << LED_D3_PIN

// Switch Interface pins.
#define SW1_PIN 4        // PTD4
#define SW2_PIN 12         // PTA12

#define SW1_PORT PORTD
#define SW2_PORT PORTA

#define SW1_GPIO GPIOD
#define SW2_GPIO GPIOA

// Switch Interface masks.
#define SW1_MASK 1 << SW1_PIN
#define SW2_MASK 1 << SW2_PIN

/*
 * Initializes pins PTD4 and PTA12 as digital input,
 * as well as PTC8 and PTA13 as digital output.
 *
 * It also configures interrupt for the A and D ports,
 * to be enabled by rising and falling edge event.
 */
extern void init_pins(void);

#endif
```

Nesse arquivo temos a definição de: macros referente aos pinos, macros de operações *bitwise* e da função **init_pins()**.

3.4 Circuito Utilizado na Prática

O circuito usado para essa prática é mostrado na figura 07. Note que os capacitores têm uma função importantíssima, pois absorver os sinais indesejados e o ruído evita que em um pressionamento de botão dispare vários sinais de interrupção.

Perceba que esse mesmo circuito foi utilizado na prática com *polling*.

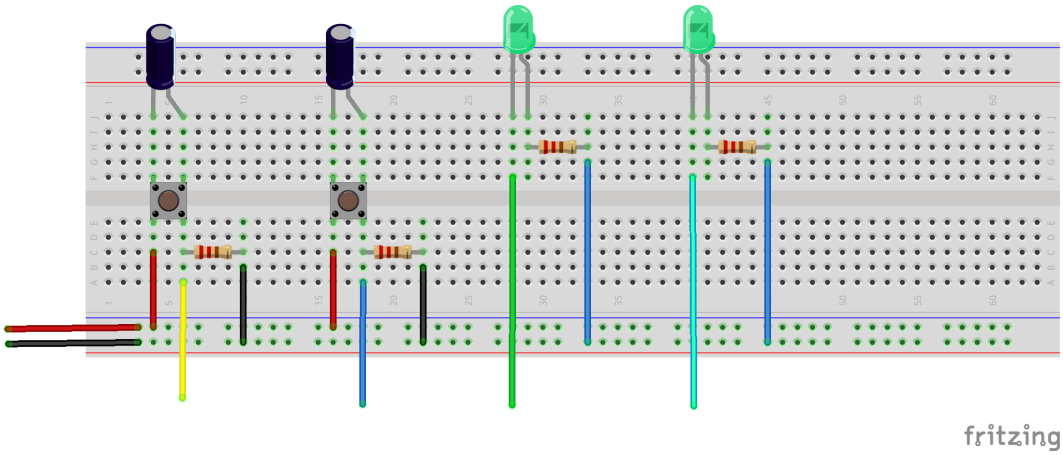

Figura 7: Circuito montado para a prática de interrupções.

4 Conclusão

Ao realizarmos a prática a equipe obteve uma noção melhor de como funciona a entrada e saída em sistemas embarcados, assim como configuramos interrupção em microcontroladores. Além disso, os conteúdos teóricos ajudaram a revisar temas estudados em outras disciplinas como TPSE I e Arquitetura de Computadores II.

Também aprendemos quando devemos usar *polling* e interrupções para entrada e saída: se o evento for **determinístico**, ou seja, sabemos quando vai acontecer, devemos usar *polling*. Se for **probabilístico**, não sabemos quando vai ocorrer, usamos interrupções.

Outro benefício que a prática proporcionou à equipe foi o melhor entendimento de como se comportam os sinais com e sem filtros, e como isso pode afetar o funcionamento de um projeto, tendo em vista que uma das etapas da prática era montar e observar em um osciloscópio a diferença entre um sinal com e sem filtro passa baixa.

Ao final podemos concluir que a realização das práticas fixou e melhorou os ensinamentos passados nas aulas teóricas da disciplina de microcontroladores, e ajudou no entendimento prático de conteúdos vistos em outras disciplinas como Eletrônica Fundamental e Processamento Digital de Sinais.

Referências

[1] Repositório da disciplina "microcontroller_practices". https://github.com/pedrobotelho15/microcontroller_practices. Accessed: 2022-04-06.