EA871 – LAB. DE PROGRAMAÇÃO BÁSICA DE SISTEMAS DIGITAIS

EXPERIMENTO 4 – Módulos SIM, PORT, GPIO e NVIC

Profa. Wu Shin-Ting

OBJETIVO: Proporcionar as noções básicas sobre a programação em C do microcontrolador Kinetis KL25Z em função das características dos dispositivos do mundo externo com os quais ele se comunica.

ASSUNTOS: Programação de sinais digitais nos pinos do microconrtolador; forma de acesso aos registradores envolvidos com o controle destes sinais; módulos SIM, PORT, GPIO e NVIC do KL2x.

O que você deve ser capaz ao final deste experimento?

Entender o princípio básico de operação do microcontrolador KL2x.

Entender a abstração de um *hardware* a nível de programação.

Entender a estrutura básica de um aplicativo baseado em microcontrolador.

Saber diferenciar monitormento de eventos por *polling* e interrupções assíncronas por eventos.

Entender o mecanismo de interrupção.

Saber consultar nos manuais os endereços dos registradores de cada módulo e as funções dos seus *bits*.

Saber ler diagramas de componentes e de máquina de estados da Linguagem de Modelagem Unificada (UML).

Conseguir elaborar um projeto simples usando os módulos SIM, PORT, GPIO e NVIC.

INTRODUÇÃO

No roteiro 2 [1] mostramos, como uma caixa preta, o projeto `rot2_aula` em C [7] que customiza a operação do microcontrolador no controle da frequência de alternância do estado de um *LED* verde conectado no pino 19 da sua porta B. No roteiro 3 [2] apresentamos, também como uma caixa preta, o projeto `rot3_aula` em linguagem de montagem [9] que controla o sinal digital do pino 20 da porta E do microcontrolador KL25Z, ao qual conectamos o canal 0 do analisador lógico Saleae para monitorarmos o tempo de execução de uma sequência de instruções na base de ciclos de instrução das operações envolvidas. Vimos que, trabalhando com os ciclos de instrução de máquina de arquitetura ARM, podemos ter uma melhor estimativa de tempo gasto na execução de um bloco de instruções. Com exercício, foi proposto que sejam embutidos os códigos de estado de espera no projeto `rot2_aula` para facilitar o controle de tempo de espera em múltiplos de 5 us. Neste experimento vamos "dissecar" as duas caixas pretas.

O nosso interesse é usar a linguagem C para programar o KL25Z de maneira eficiente. Uma visão do KL25Z, em diagrama de blocos, é mostrada na Figura 1. Além do núcleo que inclui o processador Cortex-M0+ e o controlador de interrupção, temos circuitos de controle do sistema, um banco de memórias (Flash, RAM e ROM como foi mostrado no roteiro 2 [1]), uma série de

temporizadores, circuitos de interface com sinais analógicos e sinais digitais, diferentes circuitos de interface de comunicação serial, entre outros.

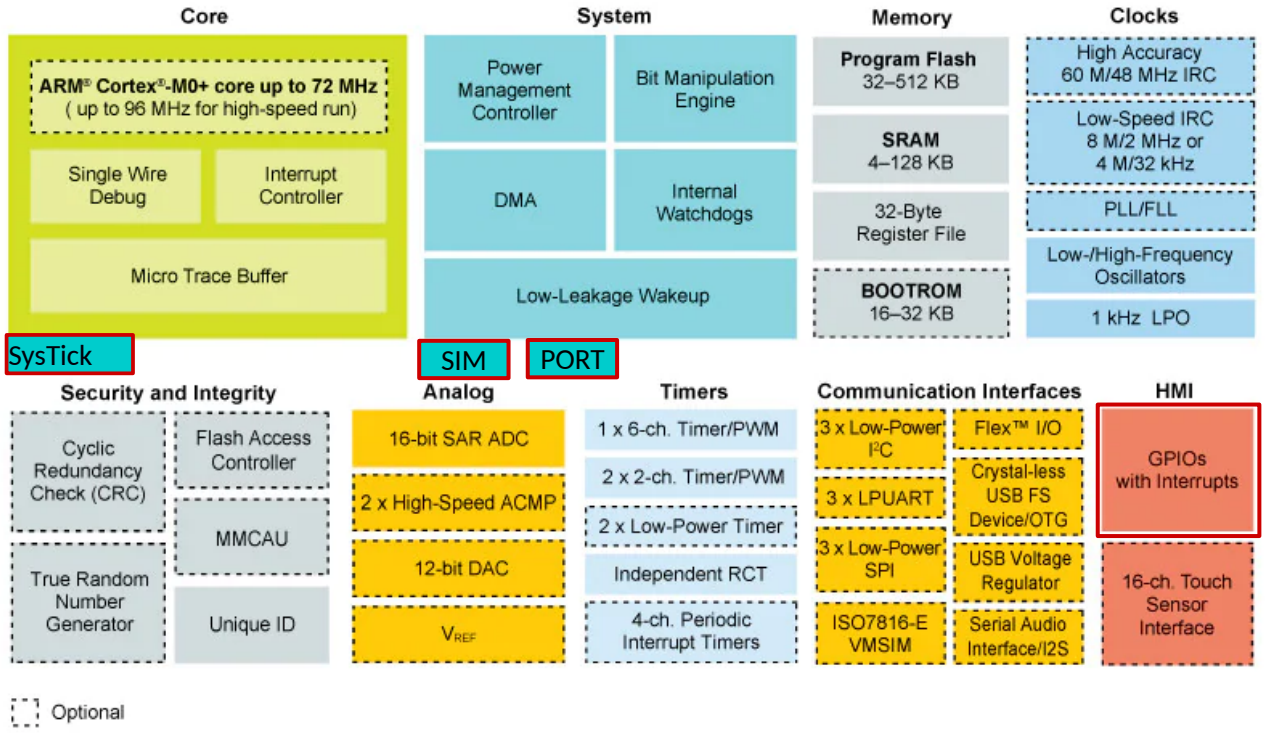

Figura 1: Diagrama de blocos de KL25Z (Fonte: [3])

Abstração de Hardware

Um projetista pode abstrair, sob o ponto de vista de programação, toda a circuitaria dos módulos de um microcontrolador num conjunto de registradores de propósito específico, mapeados em endereços definidos pelo fabricante como vimos no roteiro 3 [2]. Em [4] é dedicado para cada módulo um capítulo. Em todos eles há uma Seção de *Memory map and register definition* contendo mapa de endereços e detalhes de cada registrador. Os registradores são classificados em 3 categorias: registradores de controle/configuração, registradores de dados e registradores de estado. Os **registradores de configuração/controle** são usados pelo processador para configurar e controlar o **modo de operação** do módulo. Tipicamente os seus dados são transformados em sinais de controle. Os **registradores de dados** são usados para **transferência de dados** entre o módulo e o seu mundo externo, incluindo o processador e outros módulos integrados no microcontrolador. Tipicamente, contêm os dados de entrada/saída do módulo. E os **registradores de estado** indicam o **estado de operação** dos circuitos em relação a um modo de operação específico.

Como é o desenvolvedor que define o modo de operação de um módulo, seus registradores de configuração e de controle são setados pelo programador. Os dados de entrada são gerados externamente enquanto os dados de saída são gerados pelo módulo. O *bit* de um registrador de estado (de um evento) é setado (em "1") automaticamente pelo circuito do módulo na ocorrência do evento, sem intervenções do processador. Esses *bit* é denominado de **bandeira** (***flag***). Quando o *bit* opera em conjunto com um controlador de interrupção, ele é denominado **bandeira de interrupção** (***interrupt flag***). Para assegurar que o processador seja notificado da mudança do estado, é

permitido em alguns módulos que o processador resete uma bandeira (em "0") através de um acesso de escrita de '1' conhecido por **escrever-1-limpar**, em inglês *write-1-clear* (w1c).

Veremos ao longo desta disciplina que a quantidade de registradores associados a cada módulo estudado varia com a complexidade e a versatilidade dos módulos.

Configurações Básicas para um Aplicativo

No roteiro 2 [1] foi apresentada a função `__thumb_startup` que implementa a sequência de inicializações recomendadas pelo fabricante (Seção 1.1.4/página 12 em [5]). Essas inicializações são necessárias para carregar os dados essenciais à operação do sistema de modo geral, como o conteúdo de memória, SP e PC e o sinal de relógio do sistema. Projetado para atender uma grande gama de aplicações, é necessário personalizar essas inicializações recomendadas pelo fabricante com configurações adicionais para um modo de operação específico. Só então são executados em cima do *hardware* configurado os comandos que realizam um conjunto de tarefas.

Como uma estratégia para **reduzir o consumo de energia**, somente os módulos essenciais para a operação básica, como ULA, núcleo do processador, memória, *clock* básico etc, tem os sinais de relógio habilitados na inicialização. Os sinais de relógio dos outros módulos precisam ser habilitados por meio dos *bits* nos registradores de controle de *System Integration Module* (`SIM`) (Seção 12.2/página 191 em [4]). Os sinais de relógio são, por sua vez, gerados pelo módulo *Multipurpose Clock Generator* (`MCG`) (Seção 24.3/página 371 em [4]). O capítulo 4 em [5] sintetiza os modos de operação do `MCG` e como configurá-los através dos seus registradores de configuração. Nesta disciplina usaremos a **frequência 20.971.520 Hz** para o sinal MCGOUTCLK do qual são derivados os sinais de relógio do processador, do sistema, do barramento e da plataforma (Seção 5.4/página 116 em [4]). Esses sinais, cujas frequências são configuráveis pelo registrador de configuração `SIM_CLKDIV1` (Seção 5.4/página 116 em [4]), são distribuídos entre os módulos integrados ao KL25Z (Seção 5.7/página 121 em [4]). Para operar a uma frequência específica, **deve-se configurar a frequência do sinal de relógio de barramento de conexão do módulo e habilitar tal sinal.** Acessos a um registrador de um módulo com o seu sinal de relógio desabilitado pode gerar uma exceção do tipo *hardware fault*.

Como uma medida para **reduzir a quantidade de pinos**, e portanto o tamanho do microcontrolador, é aplicada a estratégia de multiplexação dos sinais em cada pino. Para cada porta, há um módulo *Port control and interrupts* (`PORT`) específico para controlar tal configuração (Seção 11.5/página 177 em [4]). No KL25Z, até 8 sinais são multiplexáveis em um pino físico, ou seja, um pino pode servir até 8 módulos diferentes. A seleção de um sinal específico de um módulo se dá pelos *bits* 8, 9 e 10 (`MUX`) do registrador de controle e estado `PORTx_PCRn` (Seção 11.5.1/página 183 em [4]). Figura 2 ilustra os detalhes técnicos desse registrador. Eles incluem o espaço de endereços em que é mapeadoo registrador (*Address*), a função de cada campo de *bits* (neste caso, são 14 campos) e o valor que os *bits* assumem na inicialização (*reset*) do microcontrolador. Somente alguns poucos pinos tem a sua função definida na inicialização do microcontrolador (ver a coluna `Default` da tabela na Seção 10.3.1/página 161 em [4]). Cabe ao desenvolvedor decidir e **configurar a função dos pinos a serem usados**. Os códigos binários `x`, entre 0 e 7, das possíveis

funções (`ALT x`) que um pino pode assumir estão especificados na tabela da Seção 10.3.1/página 161 em [4].

Address: Base address + 0h offset + (4d × i), where i=0d to 31d

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0							ISF	0				IRQC			
W								w1c								
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0					MUX			0	DSE	0	PFE	0	SRE	PE	PS
W																
Reset	0	0	0	0	0	x*	x*	x*	0	x*	0	x*	0	x*	x*	x*

* Notes:
- x = Undefined at reset.

PORTx_PCR*n* field descriptions

Figura 2: Detalhes técnicos do registrador de controle e estado PORTx_PCRn.

Sintetizando, ao optarmos pela configuração típica de MCGOUTCLK em 20971520Hz, devemos ainda customizar a configuração para execução das tarefas de interesse com os módulos selecionados, (1) ativando os sinais de relógio do barramento de conexão desses módulos através dos registradores de `SIM` (Seção 12.2/página 191 em [4]); (2) configurando a frequência de operação desses sinais de relógio através do registrador `SIM_CLKDIV1` (Seção 5.4/página 116 em [4]); e (3) configurando os pinos através dos registradores de `PORT` que servem esses módulos com base nos códigos `ALT` pré-fixados pelo fabricante na tabela da Seção 10.3.1/página 161 em [4]. Os dois módulos `SIM` e `PORT`, classificados como módulos do sistema (*System*), são destacados com linha vermelha na Figura 1.

Configurações Específicas por Módulo

Os microcontroladores podem ser considerados "computadores dedicados para execução de uma tarefa específica do mundo externo (a ele)". O *shield* FEEC871 [13], que se encontra devidamente conectado ao *kit* de desenvolvimento FRDM-KL25Z [5], dispõe de alguns periféricos muito comuns em projetos de sistemas embarcados que auxiliam na execução dessas tarefas. Para isso, microcontroladores precisam se comunicar eles através de circuitos de interfaces. No contexto de sistemas embarcados, uma interface é um conjunto de conexões lógicas (protocolos de comunicação) e físicas (durações dos níveis lógicos, valores de tensão) utilizadas entre um microcontrolador e os periféricos controlados por ele, para assegurar a **compatibilidade funcional**, **elétrica** (níveis de sinais), **temporal** (velocidade dos sinais) e **"mecânica"** (pinagem/conexões físicas) entre eles.

Uma grande parte dos circuitos de interface já se encontra integrada como módulos em KL25Z. Esses módulos de *hardware* dedicado realizam tarefas específicas em paralelo com o processador, incluindo a adequação dos sinais de relógio gerados pelo módulo `SIM` à velocidade de operação do periférico e o controle de fluxo de dados entre ele e o microcontrolador. Dependendo do tipo de

periférico, o projeto de uma interface ("*driver baremetal*") pode se reduzir à escolha de módulos que tenham características elétricas, funcionais e temporais mais próximas possíveis do periférico de interesse. Para desenvolver um programa que controla esses periféricos, é necessário: (1) entender as características funcionais e temporais dos periféricos; (2) identificar os pinos e as portas em que cada um está conectado; e (3) identificar os módulos mais apropriados de um microcontrolador para controlá-los. Quanto mais versátil for o módulo, mais registradores de controle/configuração ele dispõe para serem configurados.

Os módulos de circuitos dedicados para aplicativos em KL25Z são agrupados em 4 classes como ilustra a Figura 1: circuitos que envolvem sinais analógicos na interface (*Analog*), circuitos para medir intervalos de tempo decorrido entre dois eventos (*Timers*), circuitos de comunicação serial com o mundo externo (C*ommunication Interfaces*) e circuitos de comunicação com usuários (humanos) (*HMI*, *Human Machine Interface*). Além disso, está integrado junto com o processador Cortex-M0+ um controlador de interrupções conhecido por NVIC (*Nested Vectored Interrupt Controller*). Veremos que os temporizadores (*Timers*), propriamente ditos, são módulos que não se comunicam com o mundo externo. Eles geram sinais internos, derivados dos ciclos/tiques de relógio (*clock ticks*), para bases de tempo de operação dos outros módulos.

Nesta disciplina trabalharemos com um subconjunto desses módulos; mais especificamente, ADC (Capítulo 28 em [4]), Timer/PWM (Capítulo 31 em [4]), PIT (Capítulo 32 em [4]), LPTMR (Capítulo 33 em [4]), RTC (Capítulo 34 em [4]), UART0 (Capítulo 39 em [4]), UART1 e UART2 (Capítulo 40 em [4]) e GPIO (Capítulo 41 em [4]). Em [5] há exemplos de aplicação desses módulos em diferentes aplicativos, incluindo explicações detalhadas com ênfase no uso de registradores de configuração/controle para definir um modo específico de operação. Aplicaremos também o temporizador do sistema que opera na frequência do processador, `SysTick` (Seção B3.3 em [6]), e o controlador de interrupção `NVIC` (Seção B3.4/página 281 em [6]).

Periféricos: LEDs e chaves

Dentre os módulos da classe *HMI* temos um que se comunica via níveis de tensão (módulo `GPIO`) e um outro, via variações capacitivas (módulo `Touch Sensor`). Nesta disciplina, trabalharem com os periféricos que se comunicam via sinais digitais, cujos níveis de tensão variam entre dois níveis de tensão, tipicamente 0V e 3.3V, sem uma forma de onda específica. Os dois níveis 0V e 3.3V são mapeados, respectivamente em dois níveis lógicos 0 e 1 (ativo-alto) ou 1 e 0 (ativo-baixo). Dizemos que os sinais de tais periféricos são **sinais digitais de propósito geral**, como LEDs, chaves e LCD. Em todos os projetos de interface com periféricos, deve-se levar em conta as **compatibilidades funcionais** (funções de sinais), **elétricas** (níveis de sinais), **temporais** (velocidade dos sinais) e **mecânicas** (pinagens/conexões físicas) entre os sinais do microcontrolador e o periférico. Além de consultar os dados técnicos do microcontrolador, devemos analisar os dados técnicos detalhados nas folhas de dados dos periféricos.

LED (*Light Emitter Diode*) é um diodo (componente semicondutor) emissor de luz que transforma a energia elétrica em energia luminosa [17]. Por não conter o infravermelho no seu espectro de frequência, a sua luz é fria, mas libera a potência dissipada na junção do semicondutor em forma de calor que pode degradar o seu fluxo luminoso. De acordo com a folha de dados de um LED RGB

[18], a tensão direta típica dos três canais são 2,0V (R) e 3,2V (G e B). O tempo de chaveamento dos LEDs é tipicamente na ordem de alguns nanosegundos [19]. Tabela 1 sintetiza comparativamente as características de um LED em relação ao nosso microcontrolador MKL25Z. O tempo de resposta do LED junto com o tempo de alguns ciclos de instrução, na ordem de centenas de nanosegundos, dá a sensação de respostas instantâneas.

	Microcontrolador	LED
Funcional	Sinal digital de propósito geral. Níveis lógicos 1/0	Acende/Apaga
Elétrica	3,3V / 0V	3,3V / 0V
Temporal	1 ciclo de instrução: ~50ns	Chaveamento em alguns nanosegundos
Mecânico	Pino de entrada e terra	2 pinos: catodo e anodo

Tabela 1: Características funcionais, elétricas, temporais e mecâncias do LED em relação a KL25Z.

Chaves SPST (com um pólo e uma posição), momentâneas (passa momentaneamente para um outro estado quando são acionadas) e normalmente abertas (estado em repouso é aberto) são muito comuns em sistemas embarcados. Exemplos típicos são as botoeiras "*Start*", "*Stop*" e "*Reset*" que se encontram na maioria dos sistemas. **Funcionalmente**, os dois níveis lógicos (0/1) correspondem a dois estados que uma chave pode assumir (aberta/fechada). Sob o ponto de vista de **conexões**, as duas pontas de uma chave SPST podem ser conectadas, respectivamente, no pino de entrada do microcontrolador e no terra (Figura 3). Em relação à **compatibilidade elétrica**, devemos considerar **repiques** (***bounces***) dos sinais elétricos por conta das oscilações dos contatos, por um intervalo médio de 20ms, até se estabilizar num estado, tanto no instante de mudar para o estado momentâneo quanto no instante de voltar para o estado normal. Isso resulta em várias bordas de descida e várias bordas de subida tanto no pressionamento quanto na soltura da botoeira, ao invés de uma borda de descida e uma de subida num acionamento como esperado. Há diversas soluções de *debounce*. Elas podem ser por *hardware* e por *software*. A solução por *hardware* é baseada no princípio de filtragem de *bounces* pelo descarregamento lento do capacitor em relação à fonte de *bounces*. A solução por *software* é a inserção de um atraso a partir da detecção da primeira borda de *bounce*, na expectativa de que o valor lido depois do atraso seja um valor no estado estabilizado. Analisando o esquemático do *shield* FEEC871 (Figura 3), é possível constatar que a solução adotada é por *hardware* utilizando um capacitor de 100uF e um resistor de 1K. Com isso, garante-se a **compatibilização elétrica** dos sinais em dois níveis bem definidos.

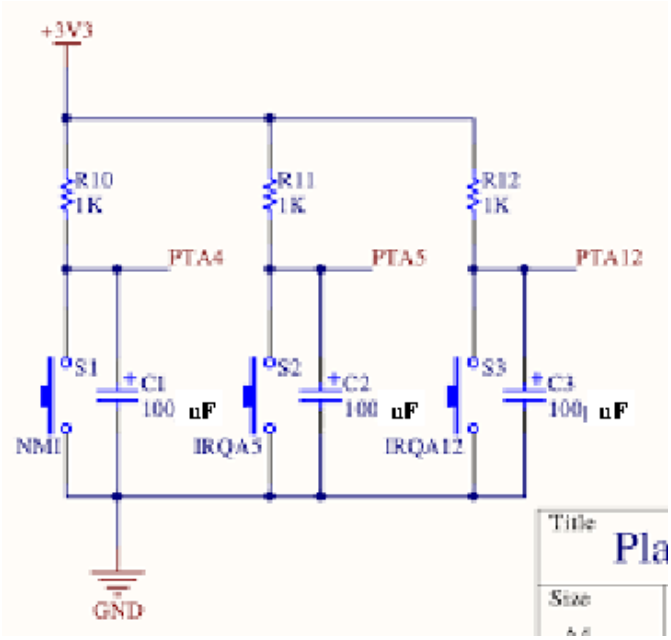

Figura 3: Esquemático da conexão de chaves no *shield* FEEC871.

E em relação à **compatibilidade temporal**, temos o problema de que uma ação dos usuários é **assíncrona** em relação aos ciclos de relógio do processador. Não se sabe exatamente o instante em que a chave pode ser acionada nem o intervalo em que ela fica acionada. Podemos adotar uma solução conhecida por ***polling*** ou por **interrupção** a ser detalhada mais adiante. Assim, aplicando uma solução de repiques (*debouncer*) por *hardware* e outra solução de "sincronismo", conseguimos assegurar os quatro tipos de compatibilidade conforme sintetiza a Tabela 2.

	Microcontrolador	Chave
Funcional	Sinal digital de propósito geral. Níveis lógicos 1/0	Aberta/Fechada
Elétrico	3,3V / 0V	com *debouncer*: 3,3V / 0V
Temporal	1 ciclo de instrução: ~50ns	com uma estratégia de sincronismo: o estado é amostrado periodicamente por *software* (*polling*) ou o evento de interesse interrompe por *hardware* um fluxo de controle (interrupção).
Mecânico	Pino de entrada e terra	Um pólo e uma posição

Tabela 2: Características funcionais, elétricas, temporais e mecâncias de uma chave SPST em relação a KL25Z.

Módulo GPIO

Os módulos `GPIO` (*General Purpose Input Output*), destacados com linha vermelha na Figura 1, são circuitos dedicados ao processamento de sinais digitais de propósito geral. No KL25Z são integrados 5 módulos `GPIO`, denominados `PORTA`, `PORTB`, `PORTC`, `PORTD` e `PORTE`. Cada módulo é responsável por 32 pinos. O circuito do módulo é relativamente simples. Ele processa os

sinais digitais de propósito geral, mapeando automaticamente os valores binários 0 e 1 setados nos seus registradores de entrada/saída em níveis de tensão 0V e 3,3V. Isso permite que o controle dos níveis desses sinais seja feito por um processador digital.

O circuito permite (1) a configuração do sentido do sinal em cada pino individualmente, entrada ou saída, através do registrador de configuração de direção `GPIOx_PDDR`, (2) o controle pelos registradores de controle `GPIOx_PSOR` (setar em '1'), `GPIOx_PCOR` (resetar em '0'), ou `GPIOx_PTOR` (alternar) do nível do sinal de saída em cada pino individualmente no registrador de dados `GPIOx_PDOR`, e (3) o armazenamento do nível de sinal de entrada no registrado de dados `GPIOx_PDIR`. Seção 41.2/página 773 em [4] sintetiza numa tabela os endereços dos registradores e as referências para as descrições detalhadas de cada registrador. Cabe observar aqui que **`GPIOx_PSOR`, `GPIOx_PCOR` e `GPIOx_PTOR` são registradores de controle cujos efeitos nos pinos ocorrem com os acessos de escrita de '1' nos respectivos *bits***. Acessos de escrita de '0' nos *bits* não alteram seus valores. O capítulo 6 em [5] mostra como controlar os pinos de entrada e saída de KL25Z através dos seus registradores de configuração do módulo (`IO`)`PORT`.

Os pinos configurados com o modo de multiplexação `GPIO` usufruem do módulo `PORT` o circuito de detecção de variações dos níveis de tensão nos sinais de entrada (Seção 11.6.3/página 188 em [4]), incluindo a configuração do tipo de variação dos níveis como um evento de interrupção (campo `IRQC` na Figura 2) e o estado de interrupção em cada pino individualmente (campo `ISF` na Figura 2 ou o registrador de estado `PORTx_ISFR` na Seção 11.5.4/página 186 em [4]).

Estrutura Básica de um Aplicativo

Revisitando o aplicativo `rot2_aula` [7], pode-se ver como o módulo `GPIO` é aplicado para controlar os sinais no LED RGB. Os comandos são organizados em quatro partes como descritos anteriormente:

- **inicialização do microcontrolador** via a função `__thumb_startup` no arquivo `Project_Settings/Startup_Code/__arm_start.c`, como vimos no roteiro 2 [1].
- **configuração básica para operação do módulo `GPIO`**: ativação do sinal de relógio do módulo `PORTB` e modo de multiplexação do pino 19 em `GPIO`. A frequência do sinal de relógio do barramento é o sinal de relógio de plataforma (Tabela 5-2/página 121 em [4]) que é derivado do sinal MCGOUTCLK por um divisor (Seção 5.4/página 116 em [4]) configurado no registrador de configuração `SIM_CLKDIV1` (Seção 12.2.12/página 210 em [4]). Sendo `OUTDIV1` = 1 não configurável, a frequência de operação do módulo `GPIOE` é fixa em 20.971.520 Hz.

```
*(uint32_t volatile *) 0x40048038u |= (1<<10);
*(uint32_t volatile *) 0x4004a04cu &= ~0x700;
(*(uint32_t volatile *) 0x4004a04cu) |= 0x00000100;
```

- **configuração do módulo `GPIO` propriamente dita**: sentido do sinal no pino

```
(*(uint32_t volatile *) 0x400ff054u) |= (1<<19);
```

- **fluxo de controle do *LED***: piscadas

```
(*(uint32_t volatile *) 0x400ff048u) = (1<<19);
for (;;) {
    (*(uint32_t volatile *) 0x400FF044u) = (1<<19);
    espera (16384);
    (*(uint32_t volatile *) 0x400FF048u) = (1<<19);
    espera (16384);
}
```

No projeto `rot3_aula` [9] foi usado o módulo `GPIOE` para controlar o estado do pino 4 do *header* H5 do *shield* FEEC871, conectado ao pino 20 da porta `PTE` do microcontrolador [13]. Vamos descrever o projeto sob o ponto de vista dos componentes configurados para implementar a tarefa de gerar uma onda quadrada no pino `PTE20`.

A figura 4 mostra, através da notação de Linguagem de Modelagem Unificada (UML) [10], como os componentes de *hardware* (em vermelho) e *software* (em preto) interagem para atender a configuração necessária ao projeto `rot3_aula`. Após a inicialização recomendada pelo fabricante (`__thumb_startup` no componente de *software* `Startup Code`), foram configurados os registradores de configuração dos módulos `SIM`, `PORTE` e `GPIOE`.

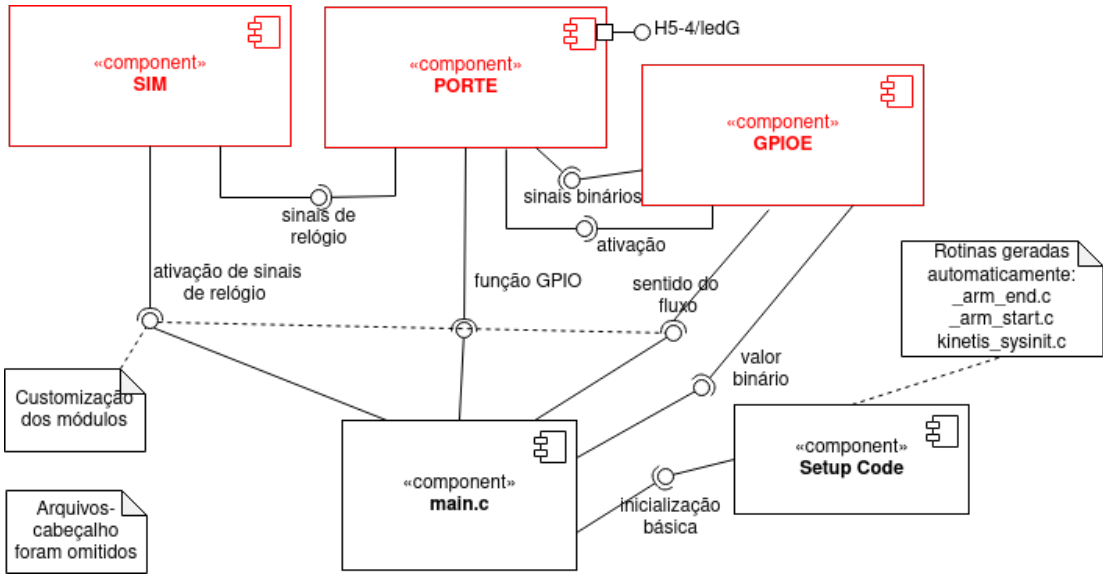

Figura 4: Diagrama de componentes do projeto `rot3_aula` em notação UML (editado em [12]).

O pino 20 da porta E é configurado para receber o sinal do *bit* 20 do registrador `GPIOE_PDOR`. Esse sinal só é habilitado para a porta E se o sinal de relógio da porta E (`PORTE`) estiver habilitado, ou seja, `SIM_SCGC5`[13]=1 (Seção 12.2.9/página 206 em [4]) e o fluxo de sinais digitais do pino estiver configurado para saída via o registrador de direção (`GPIOE_PDDR`) (Seção 41.2.6/página 778 em [4]). Por padrão, o fluxo dos pinos alocados para `GPIOx` é de entrada (0). Como o sentido do fluxo é de saída, deve-se setar o *bit* correspondente ao pino no registrador `GPIOE_PDDR` em 1 (Seção 41.2.6/página 778 em [4]). A figura 5 ilustra as interações entre os registradores.

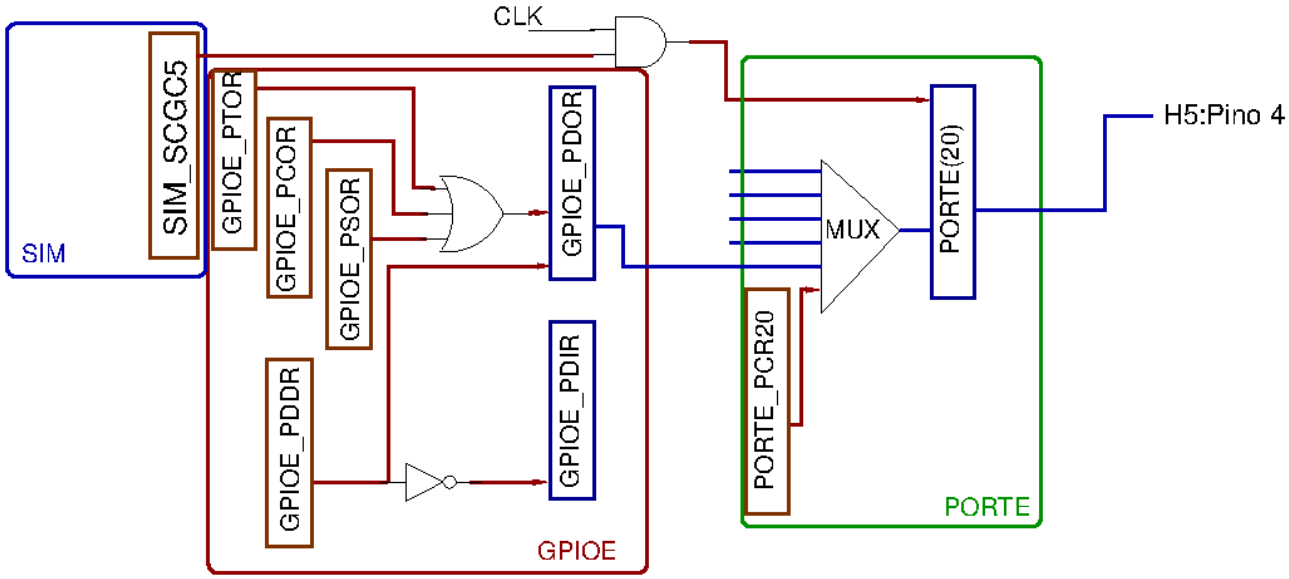

Figura 5: Registradores providos pelos módulos do microcontrolador para configuração do *hardware* do projeto `rot3_aula`.

Finalmente, sobre o *hardware* configurado é programada a forma de onda quadrada no pino `PTE20` usando os registradores de `GPOIE`. A figura 6 é uma representação do fluxo de controle da tarefa num diagrama de máquina de estados, também em notação UML [10]. Distinguem-se dois **estados** no projeto `rot3_aula`, *bit* 20 da `PORTE` em nível lógico 0 ou em 1, igualmente espaçados no tempo. Esse espaçamento é controlado pela função `espera` (*software*). Os registradores de controle, `GPIOx_PSOR` (Seção 41.2.2/página 776 em [4]), `GPIOx_PCOR` (Seção 41.2.3/página 776 em [4]) foram usados para setarem e resetarem, respectivamente, o valor binário no *bit* 20 do registrador de dados `GPOIE_PDOR`.

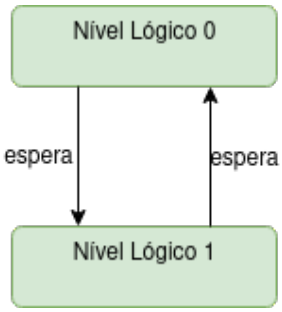

Figura 6: Diagrama de máquina de estados do projeto `rot3_aula` em UML (editado em [12]).

Note que os diagramas de componentes e de máquina de estados descrevem igualmente o projeto `rot2_aula`. Basta substituirmos `PORTE` por `PORTB` e o pino 20 pelo pino 19 para obtermos os diagramas do projeto `rot2_aula`.

Identificação de Eventos de Entrada Significativos

Num sinal digital de propósito geral só se distinguem quatro eventos característicos: borda de subida, borda de descida, nível alto e nível baixo. Portanto, é usual usar um ou mais de um desses eventos para notificar um processador de certas ocorrências assíncronas (em relação à frequência do relógio do processador). Existem circuitos dedicados que ao detectarem os eventos configurados, setam automaticamente o *bit* de estado correspondente em '1' para notificar o processador.

***Polling* e Interrupção**

A pergunta é como o processador consegue receber essa notificação se está ocupado com o processamento das instruções carregadas no segmento `.text` da memória.

Duas técnicas amplamente difundidas são: *polling* por *software* e interrupção por *hardware*. Por ***polling*,** inserem-se os comandos no código que verificam periodicamente o *bit* de estado (ou o valor de entrada). Na ocorrência do evento de interesse, executam-se os comandos de processamento relacionados. Essa técnica apresenta várias desvantagens, sendo as principais o desperdício de tempo de processamento na varredura das entradas, o tempo de resposta a uma entrada crítica e o consumo de energia para manter o processador em atividade de amostragem. Por isso, a estratégia por **interrupção** é a mais recomendada para um microcontrolador responder a um evento externo. Por **interrupção**, requer-se um circuito especial que interrompa assincronamente a execução do processador quando o *bit* de estado estiver setado (em '1') e desvie o seu fluxo de controle para uma função de comandos de processamento relacionado com o evento, denominada **rotina de serviço**, sem uma chamada explícita.

Módulo NVIC

O microcontrolador KL25Z dispõe de um *hardware* específico, o `NVIC` (*Nested Vectored Interrupt Controller*), para monitorar e processar as exceções ocorridas no núcleo e os eventos assíncronos gerados pelos periféricos e envia sinais de controle para o processador desviar o fluxo de controle para uma função chamada *handler* ou **rotina de serviço** (ISR - *Interrupt Service Routine*) (Cap.3/página 29, em [3]). Quando se conclui a execução da sequência de instruções da rotina de serviço, o processador retorna ao fluxo original. Operando junto com o processador, o `NVIC` proporciona uma forma eficiente e fácil para tratar as múltiplas e aninhadas (*nested*) solicitações de interrupção por *hardware*.

Vimos no roteiro 2 [1] que, no momento de criação de um novo projeto, o ambiente CodeWarrior gera automaticamente os arquivos `Project_Settings/Startup_Code/kinetis_sysinit.c` e `Project_Settings/Linker_Files/MKL25Z128_flash.ld`. Esses contém, respectivamente, o conteúdo do segmento `.vectortable` (tabela de vetor de interrupções) e a posição da memória onde este segmento deve ser realocado. Essa realocação ocorre automaticamente na carga do *firmware* no microcontrolador. Cada entrada `i` da tabela contém o

endereço (inicial das instruções) da rotina de serviço que processa uma requisição de número de exceção i. O número i varia de 0 a 15 para **exceções causadas por eventos síncronos gerados pelo processador** e i=IRQ (*Interrupt Request*)+16 para **interrupções por eventos assíncronos gerados por outras fontes** (Tabela 3-7/página 52 em [4]). O *bit* IRQ dos registradores `NVIC_ISER` e `NVIC_ICER` (Seção B.3.4.2, página 283, em [6]) habilita (escrevendo '1', *write-1-to-enable*) e desabilita (escrevendo '1', *write-1-to-clear*) o atendimento de um IRQ, respectivamente. Quando um IRQ é habilitado, são atendidas em primeiro lugar as solicitações pendentes. Para pular essa etapa, deve-se limpar as suas pendências escrevendo '1' (*write-1-to-clear*) no *bit* correspondente de `NVIC_ICPR` (Seção B.3.4.2, página 283, em [6]).

Como as **interrupções** do nosso microcontrolador são **vetorizadas** (por **módulo**), quando o controlador `NVIC` torna ativa uma requisição, passa para o processador a sua identificação (número de exceção) através da qual o processador consegue obter o endereço da rotina de serviço pela tabela de vetores e carregá-lo no contador de programa (PC), de forma que a próxima instrução a ser executada seja a da rotina de serviço/tratamento da interrupção requisitada. Quando há mais de uma solicitação de interrupção, o `NVIC` arbitra por *hardware* a exceção/interrupção de maior prioridade de atendimento com base na política de arbitragem adotada pelo microcontrolador, nos níveis de prioridade pré-fixados e no nível de prioridade em cada grupo configurado no registrador de configuração `NVIC_IPRn` (Seção B.1.5, página 218, em [6]).

Configuuração de Níveis de Prioridade

O **nível de prioridade de atendimento** é uma característica que o circuito NVIC precisa para determinar a ordem em que as interrupções solicitadas devem ser atendidas e, assim, garantir que as solicitações mais críticas sejam atendidas antes. Em KL25Z, as interrupções podem ser geradas por exceções, como *Reset*, *NMI* (*Non-Maskable Interrupt*) e *HardFault*, por interrupções a nível do processador, como *SysTick*, e por 32 interrupções externas (Seção B.1.5.1/página 218 em [6]). O nível de prioridade é representado por um número inteiro, onde valores mais baixos indicam uma prioridade mais alta. São reservados o**s níveis fixos -3, -2 e -1, respectivamente, para as exceções *Reset*, *NMI* e *HardFault***. **Os níveis de prioridade das interrupções são, por sua vez, configuráveis por *software***.

Na arquitetura ARM, o NVIC usa um sistema de prioridades hierárquicas. Em KL25Z são pré-fixadas (por *hardware*) em 8 níveis de prioridade, representados por 0 a 7, as 32 interrupções. Cada nível cobre 4 interrupções externas (quarta coluna da Tabela 3-7/página 52 em [4]), ou seja é pré-fixado para uma IRQ x, de 0 a 31, o nível de prioridade IPR

$$n = x / 4$$

Dentre cada nível de prioridade n distinguem-se quatro subníveis (0 a 3) configuráveis por *software* através do registrador `NVIC_IPRn` (*Interrupt Priority Register*) de 4 *bytes* sendo os **2 *bits* mais significativos de cada *byte*** 0 (x%4), 1 (x%4), 2 (x%4) e 3 (x%4) alocados, respectivamente, para os níveis de prioridade das interrupções externas `x=n*4+0`, `x=n*4+1`, `x=n*4+2` e `x=n*4+3`. Cada interrupção pode ser configurada com um valor de 0 a 3, onde 0 indica a prioridade mais alta e 3 indica a prioridade mais baixa.

Assim, quando se trata de solicitações de um mesmo `IPRn`, a de menor valor no subnível tem a

maior prioridade no atendimento e a de maior valor, a menor prioridade. Por exemplo, para as solictações de PORTA e PORTD do mesmo nível de prioridade `IPR7`, será atendido PORTD antes se for configurado o subnível 0 para PORTD e 1 para PORTA. Se as solicitações forem de diferentes níveis de prioridade `IPRn`, o subnível de prioridade setado por *software* prevalece sobre o nível de prioridade fixado por *hardware*. Por exemplo, para as solicitações de UART0 (`IPR3`) e PORTA (`IPR7`), PORTA será atendido antes se setarmos o subnível 3 para UART0 e 0 para PORTA.

Atendimentos pelo NVIC

A tabela 3-7/página 52 em [4] lista todas possíveis fontes de exceções/interrupções com seus respectivos números de exceção na coluna `Vector` e o número de requisição de interrupção na coluna `IRQ`. Note que nem todos os módulos são atendidos pelo NVIC, mesmo que sejam capazes de detectar eventos, como `PORTB`, `PORTC` e `PORTE`. Para esses módulos somente a técnica *polling* é aplicável. Neste caso, podemos tirar proveito apenas da capacidade do módulo de identificação da ocorrência dos eventos através da leitura dos bits de estado monitorados pelo módulo.

Um módulo pode discriminar uma série de eventos para tratamentos distintos. Num módulo `PORTx` são discriminados 32 eventos, um para cada pino. Entretanto, o `NVIC` só reconhece `PORTA` e `PORTD` como duas fontes de interrupção associados, respectivamente, aos números de exceção, 46 (IRQ=30) e 47 (IRQ=31) (Tabela 3-7/página 52 em [4]). A diferenciação dos eventos para tratamentos distintos acontece por *software* dentro das rotinas de serviço. Nas rotinas de serviço são lidos os registradores de estado do módulo que desencadeou a interrupção para identificar os eventos e tratá-los separadamente. Note que as **rotinas de serviço são funções cujas invocações não são programadas**. As suas "chamadas" podem acontecer em qualquer ponto. Portanto, **não há trocas diretas de dados entre elas e o programa que elas servem por meio de passagem explítica de argumentos**.

Programação de Interrupção

Com tantas funções relativas ao processamento de interrupções já implementadas em KL25Z (*hardware*) e no IDE CodeWarrior (*software*), **o trabalho do programador se reduz a configurar os módulos do microcontrolador para que eles operem no modo de interrupção e a customizar a forma de tratamento de cada exceção através da programação das respectivas rotinas de serviço**. Algumas diretrizes gerais à programação de uma rotina de serviço são: (1) entender a lógica do mecanismo de interrupção implementado; (2) habilitar todos os *bits* de habilitação de interrupção que existem entre a fonte de evento e o núcleo de processamento que efetivamente trata a exceção/interrupção e limpar as pendências/solicitações anteriores; (3) definir a rotina de serviço cuja interface já se encontra declarada no arquivo `Project_Settings/Startup_Code/kinetis_sysinit.c`; (4) dentro da rotina de serviço, detectar a fonte do sinal de interrupção através de uma varredura (*polling*) das *flags* de estado e inserir códigos de tratamento para cada fonte de interesse; (5) dependendo do módulo, limpar a flag de estado de interrupção do evento com um acesso de escrita de ‘1’ (w1c) – para o módulo GPIO é necessário este acesso de escrita para que um evento já tendido não provoque novas interrupções. Procure deixar os códigos mais simples, mais lineares (sem laços de iteração) e mais

rápidos possível. **Evite estados de espera nas rotinas de serviço**. A Figura 7 ilustra a relação entre os registradores do módulo `NVIC` com o processador e os outros módulos, como `PORTx`. O projeto `rot4_aula` [8] exemplifica uma programação de processamento de interrupções.

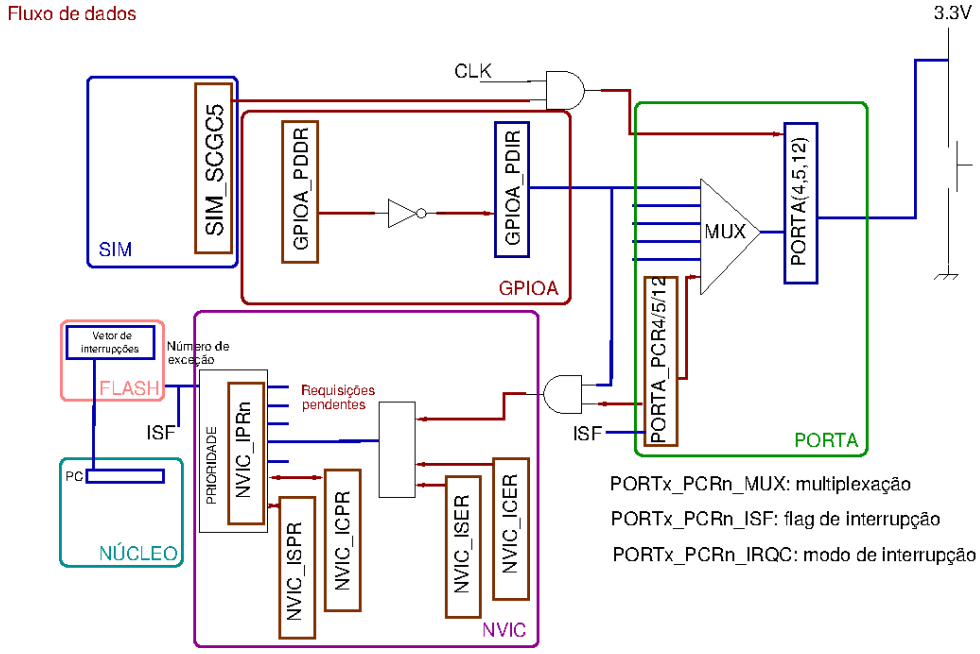

Figura 7: Fluxo de dados no processamento de uma interrupção.

EXPERIMENTO

Vamos desenvolver um projeto `ledRGB` em que a cor do *LED* RGB do *kit* FRDMKL25Z [11] em que as botoeiras NMI, IRQA5 e IRQA12 **alternam** o estado dos LEDs R, G e B, respectivamente, na **borda de descida** dos sinais de entrada conforme mostra o diagrama. Conforme os dados na Tabela 5/página 11 em [11], os canais R, G e B do *LED* estão conectados, respectivamente, nos pinos 18 (`PTB18`) e 19 (`PTB19`) da porta `PTB`, e no pino 1 da porta `PTD` e, pelo esquemático do *shield* FEEC871 [13], as botoeiras `NMI`, `IRQA5` e `IRQA12` nos pinos 4 (`PTA4`), 5 (`PTA5`) e 12 (`PTA12`) da porta `PTA`. Esses dispositivos são **ativo-baixo** (o canal correspondente acende com o pino em nível baixo e a botoeira correspondente é acionada com o pino em nível baixo). A figura 8 mostra o diagrama de máquina de estados do projeto. Os estados dos *LEDs* R, G e B são controlados separadamente pelas 3 botoeiras, gerando diferentes combinações de cores.

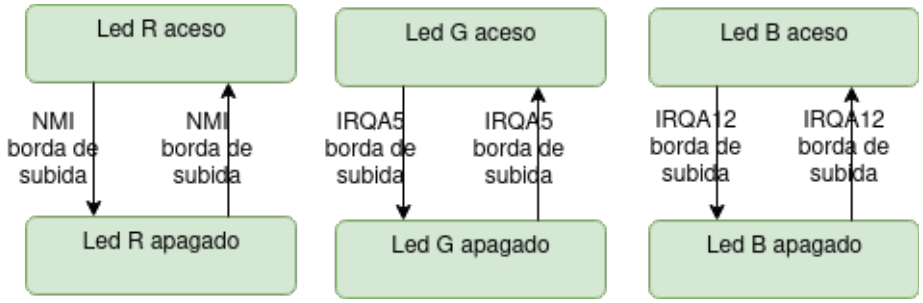

Figura 8: Diagrama de máquina de estados do projeto `ledRGB`, em UML (editado em [12]).

A figura 9 apresenta o diagrama de componentes sobre os quais o diagrama de estados apresentada na figura 6 é implementado.

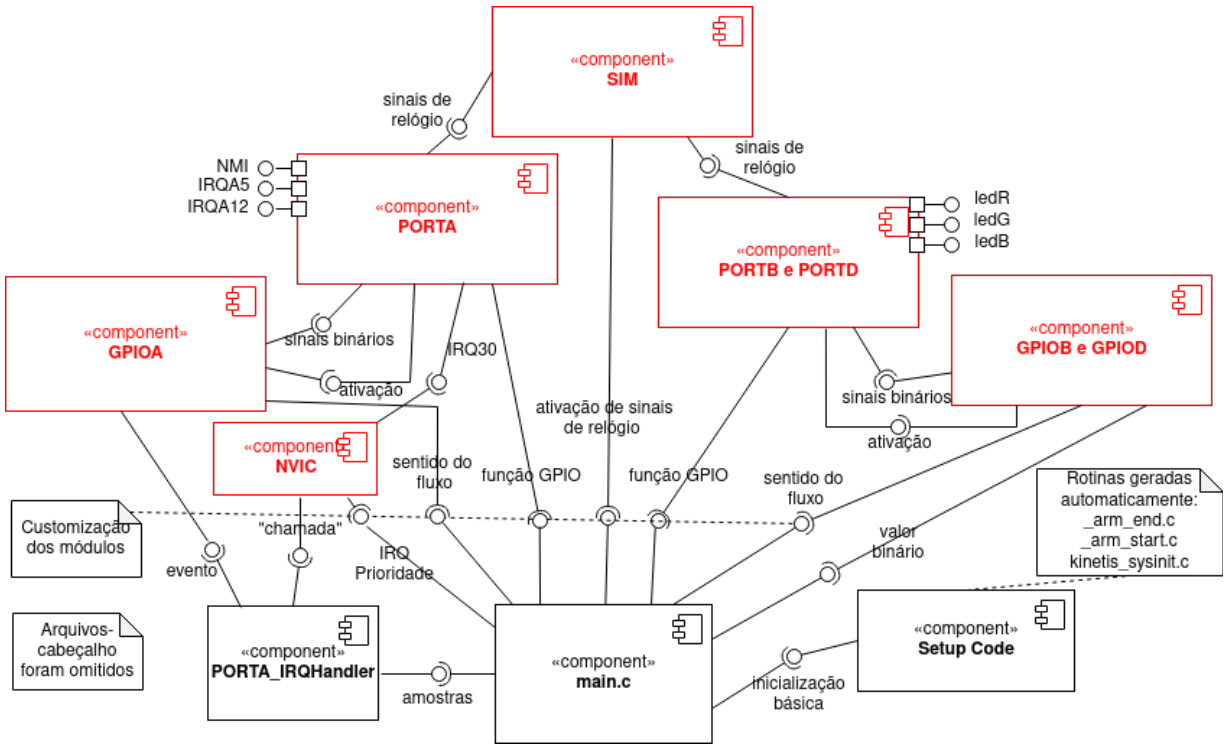

Figura 9: Diagrama de componentes do projeto `ledRGB` em UML (editado em[12]).

Segue-se um roteiro para o desenvolvimento do projeto:

1 Carregue o projeto `rot4_aula` [8] no IDE CodeWarrior e gere o executável. Conectando o *kit* LED RGB nos pinos de H5 pode ver efeitos dos sinais gerados pelas cores dos LEDs. E conecte dois canais do analisador lógico Saleae [14] nos pinos `PTE21` e `PTE22` do microcontrolador (correspondem aos pinos 3 e 2 do *header* H5) e o seu terra no pino 5 de H5. A pinagem do *shield* FEEC871 é mostrado em [13] e os pinos, do LED ao conector à placa, no *kit* LED RGB correspondem aos pinos 0 a 5 de H5. Ao longo dos experimentos, mantenha até 2 *breakpoints* ativados.

1.a Execute o programa no modo `Run`, inicie a captura dos sinais por um intervalo de 10s e aperte 3 vezes a botoeir NMI e 2 vezes a botoeira IRQA5. Anote a sequência das cores observadas e analise-as comparativamente o sinal gerado pelo analisador.

1.b **Configurações do microcontrolador:** Execute o programa no modo `Debug` para analisar os papéis dos registradores na implementação de uma terefa num microcontrolador. Monte numa tabela de 6 colunas com os registradores acessados até o fim da execução da função `inicializacao`. A primeira contém a identificação dos registradores. A segunda contém os valores dos registradores com a parada do fluxo de execução na linha `config_basica` (após a execução de `__thumb_startup`), a terceira os valores dos registradores com a parada do fluxo de execução na linha `config_especifica` (após a configuração do modo de operação dos módulos alocados), a quarta, os valores dos registradores com a parada do fluxo de execução na linha `inicializacao` (após a inicialização dos registradores de dados para a tarefa a ser executada), a quinta, os valores assumidos pelos registradores no laço

de execução da tarefa de controle do estado dos LEDs pelas botoeiras, e a sexta coluna, a função de cada registrador na implementação.

1.c **Identificação de Eventos de Entrada Significativos:** Quais são os eventos usados para identificar variações nos sinais de entrada das botoeiras `NMI` e `IRQA5`? Justifique com base nos códigos programados.

1.d ***Polling* e Interrupção:** Sendo o processamento de interrupção por *hardware*, uma forma para certificar se a configuração de interrupção está correta é colocar um ponto de parada no início da rotina de serviço correpondente ao evento de interrupção e disparar um evento de interrupção que, no caso, correponde ao acionamento de uma das botoeiras `NMI` ou `IRQA5`. Se a interrupção estiver devidamente habilitada, o fluxo de controle é automaticamente desviado para a rotina de serviço na ocorrência de um evento de interrupção. Em qual das rotinas do arquivo `main`, deve-se colocar um ponto de parada no início do seu bloco de instruções para verificar qual das duas botoeiras tem o mecanismo de interrupção configurada? Acione as duas botoeiras e responda qual delas leva o fluxo de controle para o ponto de parada setado, ou seja, qual delas está com a interrupção habilitada? Quais configurações adicionais foram necessárias para essa botoeira na função `config_especifica`? Como é o processamento da outra botoeira sem interrupção na função `main` para que as duas botoeiras tenham comportamentos equivalentes?

1.e **Periféricos: LEDs e chaves:** Como o *shield* FEEC871 é provido de circuitos de *debouncer* por *hardware*, quantas vezes você espera que o fluxo seja desviado para a rotina `PORTA_IRQHandler` quando pressionada e solta a boteira com interrupção habilitada? Funciona o circuito *debouncer* no *shield* que você testou? Faça **testes de unidade** de *debouncer*: Coloque um ponto de parada na primeira instrução de `PORTA_IRQHandler`, execute o programa e pressione a botoeira. Ao parar o fluxo de execução no ponto de parada setado, retome (*Resume*) a execução do programa com a botoeira pressionada. Sempre mantendo a botoeira pressionada, retome o fluxo de execução e repita o procedimento até que não entre mais em `PORTA_IRQHandler`. Daí solta a botoeira. Conte o número de entradas. Solte a botoeira e repita o procedimento para contar quantas vezes o fluxo foi desviado para a rotina PORTA_IRQHandler na soltura da botoeira.

1.f **Módulos GPIOx:** Entre `GPIOx_PCOR`, `GPIOx_PSOR`, `GPIOx_PTOR`, `GPIOx_PDDR`, `GPIOx_PDIR` e `GPIOx_PDOR`, em quais deles acessos de escrita de '0' nos seus *bits* não tem efeito sobre os pinos correspondentes? Em quais deles acessos de escrita de '0' tem efeito no estado do pino correspondente? Responda com base nos operadores em C aplicados neles em `config_especifica` e `PORTA_IRQHandler`.

1.g **Módulo NVIC:** Acessos de escrita de '0' nos *bits* dos registradores `NVIC_ISER`, `NVIC_ICER`, `NVIC_ISPR` e `NVIC_ICPR` tem efeito sobre o estado de IRQ correspondente? Responda com base nos operadores em C aplicados para "configurar" `NVIC_ISER`, `NVIC_ICER`, e `NVIC_IPRn` na rotina `config_especifica`.

1.h **Módulos GPIOx e NVIC:** Analisando os valores assumidos pelos registradores durante a execução do laço `for` da função `main`, conclui-se que o módulo `GPIOx` (x=A e E) participa da execução da tarefa. E o módulo `NVIC`, quais registradores foram usados na sua configuração? Note que os valores dos seus registradores não foram alterados na execução do

laço. NVIC participa da execução? Justifique com base no seu entendimento sobre o mecanismo de interrupção.

1.i **Atendimento pelo NVIC:** O que aconteceria se removermos as duas linhas de instruções que configuram o evento de interrupção na função `config_especifica`

```
*(uint32_t volatile *) 0x40049014u &= ~0xF0000;
(*(uint32_t volatile *) 0x40049014u) |= 0x000B0000;
```

ou a linha de habilitação do IRQ 30 no módulo NVIC pela instrução

```
(*(uint32_t volatile *) 0xe000e100u) = (1<<30);
```

E se configuramos um evento de interrupção no pino PTE23 (corresponde ao pino 1 do *header* H5) e conectarmos uma botoeira a este pino, este evento de interrupção pode ser atendido pelo controlador NVIC? Justifique as suas respostas com base no seu entendimento pelo mecanismo de interrupção.

1.j **Programação de Interrupção:** A escolha do nome da função `PORTA_IRQHandler` é arbitrária? Para que servem as duas linhas abaixo na função?

```
if ((*(uint32_t volatile *) 0x40049014u) & 0x1000000)
e
(*(uint32_t volatile *) 0x40049014u) |= (1<<24);
```

2 Desenvolva o projeto `ledRGB` com o estado dos LEDs R, G e B controlados pelas botoeiras `NMI`, `IRQA5` e `IRQA12`, respectivamente. Recomenda-se os seguintes passos que procuram reusar os códigos do projeto `rot4_aula` por ter um fluxo de controle bem similar ao do projeto `ledRGB`.

2.a Crie um novo projeto (Seção 2.1/página 4 em [15]).

2.b Sobreescreva o arquivo `main.c` do projeto `rot4_aula` sobre `main.c` do novo projeto e faça os **testes funcionais** para certificar o porte (Seção 2.2.3/página 14 em [15]).

2.c Substitua pino `PTE21` (ativo-alto) pelo pino `PTB18` (ativo-baixo) e o pino `PTE22` (ativo-alto) pelo pino `PTB19` (ativo-baixo). Faça **testes de unidade** se os LEDs do *kit* foram substituídos corretamente pelos LEDs de FRDMKL25Z.

2.d Configure o evento de interrupção por borda de subida para as 3 botoeiras:

- duas botoeiras `NMI` e `IRQA5`. Faça **testes de unidade** que verifica se ao acioná-las o fluxo é desviado para `PORTA_IRQHandler`.
- a botoeira `IRQA12`. Faça **testes de unidade** que verifica se ao acioná-la o fluxo é desviado para `PORTA_IRQHandler`.

Neste ponto, o interesse é só na configuração de interrupção das 3 botoeiras. Não é feito nenhum controle dos estados dos LEDs em resposta aos eventos de inerrupção gerdos.

2.e Programe as respostas dos LEDs R e G em relação aos eventos de interrupção em `PORTA_IRQHandler`:

- no lugar de espelhar o sinal da botoeira, só é necessário alternar o estado dos LEDs nas bordas de subida do sinal. Substitua o bloco de tratamento do *bit* de *flag* 5 (0b100000) pelos comandos de alternância (*toggle*) do estado do LED G. Faça **testes de unidade** que verifica se o estado do LED G é alternado ao soltar a botoeira `IRQA5`.
- remova o bloco de instruções dentro do escopo `for` na função `main` e adicione o tratamento do *bit* de *flag* 4 com os comandos de alternância (*toggle*) do estado do LED R e o *reset* da *flag* de interrupção. Faça **testes de unidade** que verifica se o estado do LED R é alternado ao soltar a botoeira `NMI`.

2.f Programe a resposta do LED B em relação **aos eventos de interrupção** em `PORTA_IRQHandler`

- ative o módulo `PORTD`, configure o modo de multiplexação do pino `PTD1`, configure o módulo `GPIOD`. Faça **testes de unidade** inicializando o LED B no estado aceso (*clear*).
- adicione o tratamento do *bit* de *flag* 12 com os comandos de alternância (*toggle*) do estado do LED B e o *reset* da *flag* de interrupção. Faça **testes de unidade** que verifica se o estado do LED B é alternado ao soltar a botoeira `IRQA12`.

2.g Habilite *Print Size* para uma simples análise do tamanho de memória ocupado. Gere um executável e faça **teste funcionais** do projeto acionando as 3 botoeiras aleatoriamente para verificar se as cores exibidas são condizentes com as esperadas.

2.h Documente as funções implmentadas e gere uma documentação do projeto (Seção 2.9/página 47 em [15]).

RELATÓRIO

O relatório deve ser devidamente identificado, contendo a identificação do instituto e da disciplina, o experimento realizado, o nome e RA do aluno. Para este experimento, responda o item 1 e registre os resultados dos testes de unidade malsucedidos no item 2 e suas ações de correção num arquivo em pdf. Exporte o projeto `ledRGB` **devidamente documentado para um arquivo depois de aplicar *Clean* no projeto e apagar a pasta de documentação html e latex**. Suba os dois arquivos no sistema *Moodle*.

REFERÊNCIAS

[1] Roteiro 2
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/roteiros/roteiro2.pdf
[2] Roteiro 3
http://www.dca.fee.unicamp.br/cursos/EA871/1s2023/roteiros/roteiro3.pdf
[3] KL2x: Kinetis® KL2x-72/96 MHz, USB Ultra-Low-Power Microcontrollers (MCUs) based on Arm® Cortex®-M0+ Core
https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/kl-series-cortex-m0-plus/kinetis-kl2x-72-96-mhz-usb-ultra-low-power-microcontrollers-mcus-based-on-arm-cortex-m0-plus-core:KL2x
[4] KL25 Sub-Family Reference Manual, Rev. 3, September 2012
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KL25P80M48SF0RM.pdf
[5] Freescale. Kinetis L Peripoeral Module Quick Reference
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/KLQRUG.pdf
[6] ARMv6-M Architecture Reference Manual.
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/ARMv6-M.pdf
[7] rot2_aula.zip
https://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot2_aula.zip
[8] rot4_aula.zip
https://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot4_aula.zip
[9] rot3_aula.zip
https://www.dca.fee.unicamp.br/cursos/EA871/1s2023/codes/rot3_aula.zip
[10] Wu, S.-T. Linguagens de Modelagem
https://www.dca.fee.unicamp.br/cursos/EA871/references/complementos_ea871/linguagens_modelagem.pdf
[11] Freescala FRDM-KL25Z User's Manual
https://www.dca.fee.unicamp.br/cursos/EA871/references/ARM/FRDMKL25Z.pdf
[12] Diagrams.net (editor de diagramas *online*)
https://www.diagrams.net/
[13] Nova Versão do Esquemático do shield FEEC871

https://www.dca.fee.unicamp.br/cursos/EA871/references/complementos_ea871/Esquematico_EA871-Rev3.pdf
[14] Analisador Lógico Saleae
https://www.saleae.com/pt/
[15] Wu, S.T. Ambiente de Desenvolvimento – *Software*
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf
[16] Doxygen: Special Commands
https://www.doxygen.nl/manual/commands.html
[17] Laboratório de Iluminação. LED – O que é, e como funciona.
https://hosting.iar.unicamp.br/lab/luz/dicasemail/led/dica36.htm
[18] RED/GREEN/BLUE Triple Color LED
https://www.dca.fee.unicamp.br/cursos/EA871/references/datasheet/YSL-R596CR3G4B5C-C10.pdf
[19] What is the latency of an LED?
https://electronics.stackexchange.com/questions/86717/what-is-the-latency-of-an-led

Revisado em Janeiro de 2023.
Revisado em Agosto de 2022.
Revisado em Março e Julho de 2021.
Revisado em Setembro de 2020.