UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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

QXD0143 - MICROCONTROLADORES

Relatório 02: Entrada e Saída Digital no ARM usando o Periférico de *GPIO*

QUIXADÁ - CE
2022

471047 — PEDRO HENRIQUE **MAGALHÃES BOTELHO**

427602 — ERICK **CORREIA SILVA**

472012 — FRANCISCO ITALO DE **ANDRADE MORAES**

Relatório 02: Entrada e Saída Digital no ARM usando o Periférico de *GPIO*

Orientador: Prof. Dr. Thiago Werlley Bandeira da Silva

Segundo 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 segundo relatório da disciplina de **Microcontroladores**, ministrada pelo professor Thiago W. Bandeira. Nesse relatório é discutido sobre a entrada e saída digital de dados utilizando microcontroladores ARM®, onde são apresentadas as práticas realizadas em laboratório sobre o periférico de **General Purpose Input/Output (GPIO)**.

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:

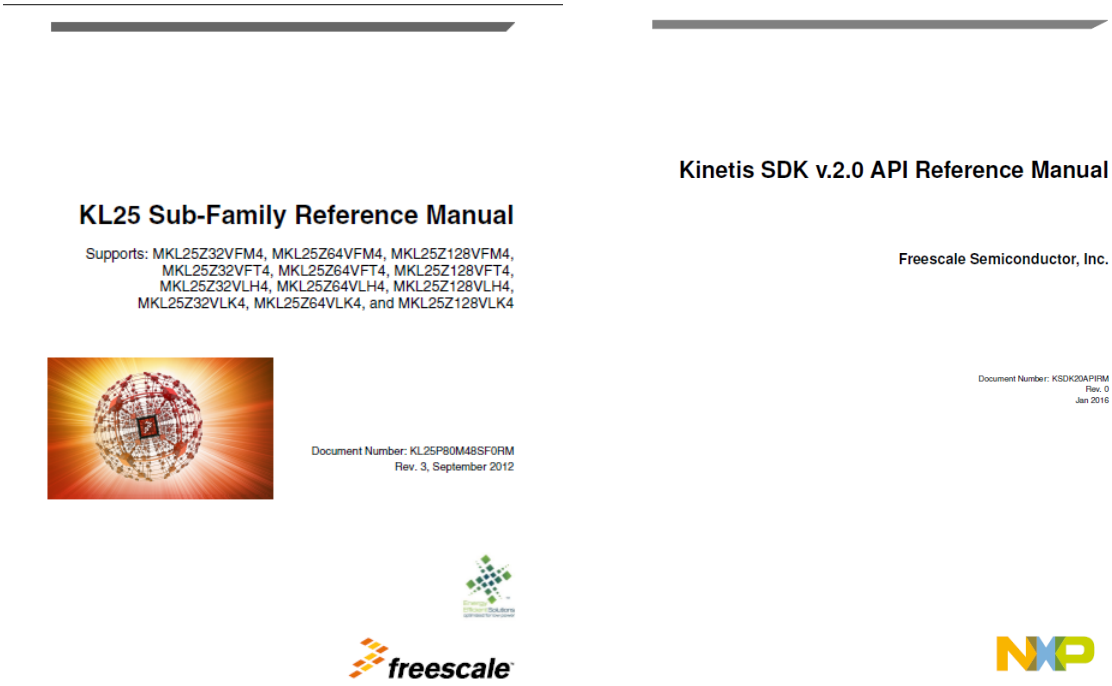

Figura 1: Capas dos Manuais de Referência do MKL25Z128VLK4 e do SDK

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 Microcontrolador ARM Utilizado

Para essa prática foram utilizados um microcontrolador ARM, como dito anteriormente: o MKL25Z128VLK4. Uma opção barata e eficiente de microcontrolador com núcleo ARM. Vamos discorrer sobre as características dele rapidamente.

O MKL25Z128VLK4 é um microcontrolador muito interessante, muito utilizado para sistemas operacionais de tempo real e de fácil configuração, sendo mais voltado para aplicações mais simples e com baixo consumo. Conta com vários recursos interessantes, como touch capacitivo, acelerômetro, LED RGB integrado e debugger.

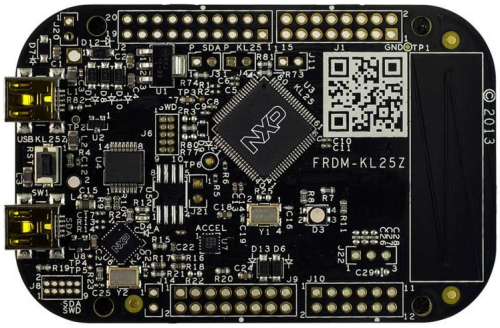

Figura 2: Freedom FRDM-KL25Z com MCU MKL25Z128VLK4

As principais características do MKL25Z128VLK4 são:

- Núcleo ARM Cortex-M0+ com frequência de clock de até 48MHz
- 16 kB de Memória SRAM
- 128 kB de Memória Flash
- Empacotamento LQFP de 80 pinos.
- 5 portas de pinos de entrada e saída
- Até 66 linhas de I/O, controladas por controladoras PORT

3 Entrada e Saída digital no ARM

O microcontrolador, como dito no último relatório, tem todos os componentes de um computador, inclusive periféricos, como mostra a figura 03. Esses periféricos fazem a interface da CPU com o mundo externo. Um desses periféricos é o **GPIO**, *General Purpose Input Output*.

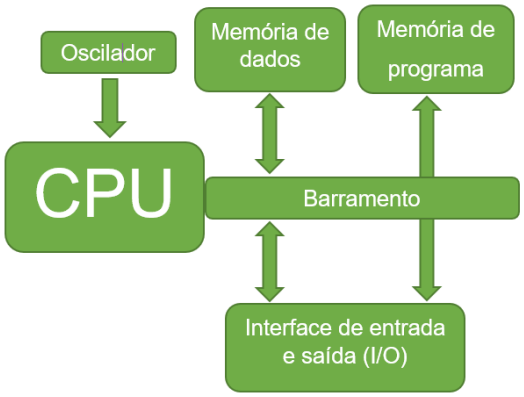

Figura 3: Organização básica de um microcontrolador

Os pinos do microcontrolador são controlados por um periférico específico (periférico **PORT**, no MCU MKL25Z128VLK4), no qual podemos configurar pinos do microcontrolador, como a função desempenhada por ele, se irá funcionar com interrupções, DMA, etc...

Os pinos são multiplexados para várias funções, já que o encapsulamento do chip tem um número limitado de pinos e o microcontrolador tem muitas funções. Logo múltiplas funções estão mapeadas para um único pino, e devemos selecionar a função específica que será utilizada.

Utilizamos então o periférico de GPIO para mandar tensões aos pinos (atuar), ou ler tensões dos pinos (sentir), realizando assim entrada e saída digital.

3.1 Mapeamento de Memória no ARM

Todos os periféricos do microcontrolador, assim como as memórias, estão mapeadas em um espaço de endereçamento. Isso quer dizer que podemos acessar os componentes do MCU por meio de um endereço. Os periféricos possuem registradores, que estão mapeados na memória, onde todo o controle destes periféricos é feito. Chamamos isso de **mapeamento de memória**. O mapeamento de memória do MCU utilizado está representado na figura 04.

Sendo um processador de 32-bits o ARM Cortex-M0+, assim como o ARM Cortex-M3, tem 4GB de espaço de endereçamento e os componentes podem ser mapeados nos endereços disponíveis dentro desse espaço. Isto é, separar um bloco para acessar determinado periférico.

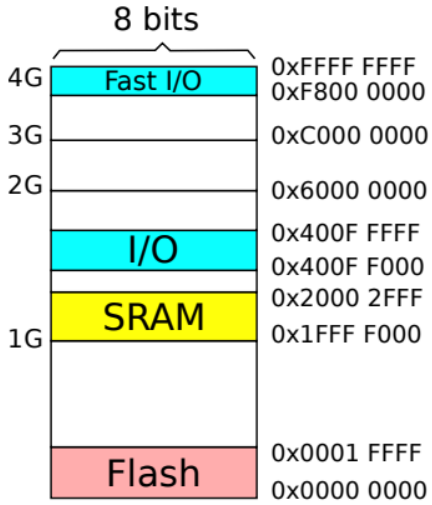

Figura 4: Mapa de Memória Básico do ARM Cortex-M0+

No espaço de endereçamento dos periféricos temos os registradores de controle, que utilizamos para configurar o periférico. Portanto, devemos mapeá-los no código para utilizarmos no nosso firmware, isto é, um sistema rodando diretamente sobre o hardware, também conhecido como **bare metal**, escrito em C.

Temos alguns componentes importantes:

- Flash: a memória de programa, onde fica o código do programa e dados fixos.
- SRAM: a memória de dados, dados variáveis utilizados durante o programa.
- I/O: Registradores de controle dos periféricos ligados ao APB (Advanced Peripheral Bus) para entrada e saída de dados, como timers, ADC e GPIO.
- Fast I/O: Registradores de controle dos periféricos ligados ao AHB (Advanced High-performance Bus), que necessitam operar de forma mais rápida, como DMA.

3.2 Configurando os Pinos do Microcontrolador

Os pinos do microcontrolador para entrada e saída estão mapeados em 5 portas: A, B, C, D e E, onde cada porta tem no máximo 32 pinos, sendo que nem todos estão implementados. Isso significa que uma porta pode não ter os 32 pinos utilizáveis. Cada porta tem um módulo do controlador PORT, que provê registradores de controle para cada pino, como o registrador **PCR (Pin Control Register)**, que nos permite configurar a função do pino, configurações de interrupção, etc...

Um pino pode ter várias funções, como UART, GPIO, ADC, DAC, etc... Portanto devemos selecionar qual a função será utilizada para o pino específico em um **multiplexador de**

função que todos os pinos possuem. No nosso caso selecionaremos a função de GPIO para os pinos utilizados.

O multiplexador de função do pino, como mostra a figura 05, nos permite selecionar, via software no registrador PCR do pino, qual função ele irá desempenhar. Esse registrador PCRn, onde n é um pino de 0 a 31, está mantido em um controlador PORTx, onde x é a porta do pino, de A a E.

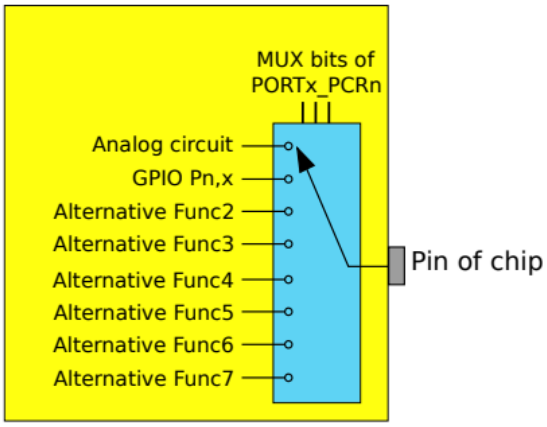

Figura 5: Multiplexador de Função do Pino

Para utilizar a função de GPIO precisamos configurar os bits referentes ao MUX no registrador PCR. Se colocarmos 001 nos bits 10-8 estaremos utilizando a função de GPIO para aquele pino, como mostra a figura 06:

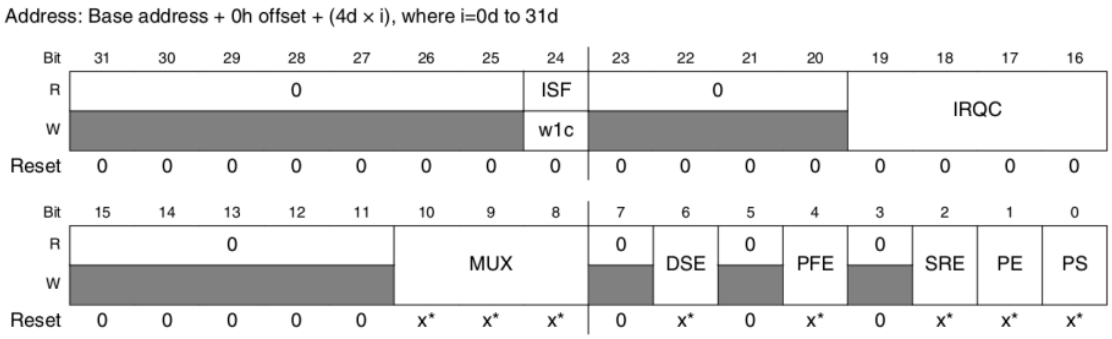

Figura 6: Multiplexador de Função do Pino

Em C utilizamos o registrador PCR da seguinte forma.

```c
typedef struct{
  uint32_t PCR[32];
} PORT_t;

#define PORTB ((PORT_t*) (0x4004A000))
//...
PORTB->PCR[18] |= (1 << 8);     // Pino 18 da porta B sera usado como GPIO
```

Perceba que o periférico é mapeado com uma *struct*, que tem, em seu interior, os registradores de controle. Perceba que temos um vetor de 32 em PCR, isso porque temos um PCR para cada um dos 32 pinos da porta! Na linha 5 pegamos o bloco de memória de PORTB e colocamos para ser usado como uma struct. Isso organiza muito o código.

Usamos a operação bitwise | (OR) para escrever no registrador, com uma máscara correspondente ao bit.

Mas atenção! Tente não confundir o periférico PORT com GPIO. O periférico que oferece interface para o pino é o PORT. Ele que direciona os dados ao pino e realiza seu controle, já o periférico que diz que dados devem ser emitidos ao pino é o GPIO. O esquema completo pode ser visualizado na figura 07.

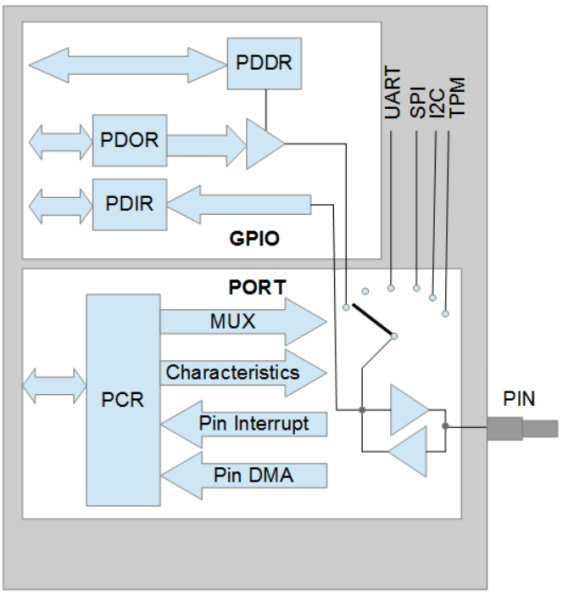

Figura 7: Esquema do Periférico PORT

3.3 Habilitando o Clock para os Periféricos

Antes de usarmos um periférico devemos habilitar o clock para ele. Todos os componentes em uso precisam estar sincronizados por meio de um sinal em comum, o clock. Utilizaremos o módulo de integração do sistema para habilitar o clock para os periféricos, o **SIM**.

O registrador System Clock Gating Control Register 5 (SCGC5) oferece o controle do clock para as portas A a E, controladas pelo PORT. Devemos habilitar o clock antes de podermos usar o periférico, e os periféricos que não estão em uso devem ter o clock desabilitado para que a energia seja poupada.

Na figura 08 temos o registrador SCGC5 e os respectivos bits das portas. Devemos ativar o bit referente a porta que iremos utilizar.

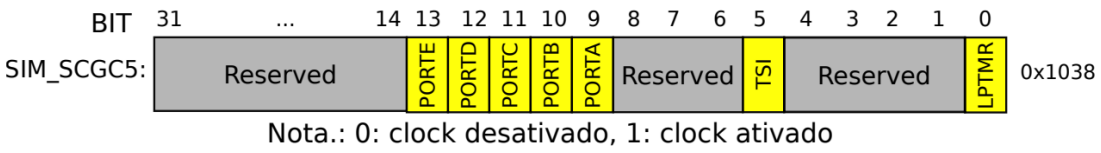

Figura 8: Registrador SCGC5 do Módulo de Integração do Sistema (SIM)

No código abaixo vemos como habilitar o clock para o controlador PORTB, em C. Dessa forma podemos utilizar todos os pinos mapeados para a porta B.

```
SIM->SCGC5 |= (1 << 10);     // Habilita o clock para a porta B.
```

3.4 Entrada e Saída Digital com GPIO

Feitas as configurações de clock e de multiplexador podemos finalmente configurar o GPIO para entrada e saída. O periférico de GPIO nos fornece uma grande variedade de registradores para controlar a entrada e saída de dados via software. Cada bit de um registrador do GPIo corresponde a um pino. Por exemplo, o pino PTB18 corresponde ao bit 18 dos registradores do GPIOB.

Primeiramente precisamos definir se o pino será de entrada de dados ou de saída de dados. Essa configuração será feita no registrador **PDDR**, Data Direction. Se o pino for de entrada colocaremos 0 no bit correspondente, e se for de saída colocaremos 1.

Podemos aplicar uma tensão no pino por meio do registrador **PDOR**, Data Output, colocando 0 no bit correspondente para uma tensão de 0V, ou 1 para uma tensão de +3,3V.

Podemos também utilizar três registradores adicionais: escrevendo 1 no bit correspondente ao pino no registrador **PSOR**, Set Output, teremos nível lógico alto no pino, escrevendo 1 no bit do registrador **PCOR**, Clear Output, teremos nível lógico baixo, e escrevendo 1 no bit do registrador **PTOR**, Toggle Output, iremos trocar o nível lógico do pino: se era alto, será baixo, e se era baixo será alto. Ler desses registradores sempre retorna 0.

O registrador **PDIR**, Data Input, nos permite obter uma entrada do pino, ou seja, obter o nível lógico aplicado ao pino, seja 0V, para 0, ou +3,3V, para 1. Usaremos a operação bitwise & (AND) para ler desse registrador com uma máscara correspondente ao bit.

Em resumo, podemos usar os registradores do GPIO da seguinte forma:

```
GPIOB->PDOR |=  (1 << 18);    // Coloca PTB18 em nivel logico alto.
GPIOB->PDOR &= ~(1 << 18);    // Coloca PTB18 em nivel logico baixo.
GPIOB->PSOR |=  (1 << 18);    // Coloca PTB18 em nivel logico alto.
GPIOB->PCOR |=  (1 << 18);    // Coloca PTB18 em nivel logico baixo.
GPIOB->PTOR |=  (1 << 18);    // Troca o nivel logico de PTB18.
if(GPIOB->PDIR & (1 << 17)) { ... } // Verifica o nivel logico em PTB17.
```

4 Piscando o LED RGB nas 7 cores

A primeira prática consiste em piscar um LED RGB nas 7 corres, ou seja, nas 7 possíveis combinações de cores RED, GREEN e BLUE. O LED RGB está embutido na placa FRDM-KL25Z, portanto não precisamos de um circuito externo, apenas usar os pinos mapeados para o LED.

A seguir é mostrado duas formas de piscar o LED RGB da placa KL25Z. Na primeira forma são utilizados os registradores de controle dos periféricos diretamente, ou seja, são modificados "manualmente" os registradores necessários para realizar o **blink**. Na segunda forma utilizamos a API SDK (Software Development Kit), da NXP, um conjunto de funções e constantes prontas para serem usadas, para auxiliar no desenvolvimento da prática.

Os códigos estão estruturados em funções, sendo a principal a função 'int main(void)", que realiza a chamada da função de inicialização e faz o processo de piscar o LED nas cores. Outra função muito importante é a "void init_pins(void)" onde realizamos a inicialização dos pinos utilizados, seguindo os mesmos passos descritos na seção 3. Primeiramente habilitamos o clock das portas utilizadas, PORTB e PORTD, e configuramos a função dos pinos como GPIO (no MUX) e por fim setamos os pinos referentes aos LED (pinos PTD1, PTB18 e PTB19), que compõem o LED RGB, como saídas digitais.

Temos ainda a função de delay, que, informado uma quantidade de tempo em milisegundos deixa o processador ocupado, gerando um atraso, ou seja, um tempo ocioso. Na função "main", logo após chamarmos a função de inicialização dos pinos, iniciamos o *loop* infinito, contendo as combinações necessárias para acender cada combinação de cor. Perceba que usamos macros para ativar e desativar bits.

4.1 Blink RGB com Registradores de Controle

```
1  #include "Rainbow_Blinky.h"
2  #include "MKL25Z4.h"
3
4  int main(void) {
5    init_pins();
6    while(1){
7      // Red Color:
8      SetBit(GPIOB->PSOR, LED_RED_BIT);
9      SetBit(GPIOB->PCOR, LED_GREEN_BIT);
10     SetBit(GPIOD->PCOR, LED_BLUE_BIT);
11     delay_ms(PULSE_WIDTH);
12
13     // Green Color:
14     SetBit(GPIOB->PCOR, LED_RED_BIT);
15     SetBit(GPIOB->PSOR, LED_GREEN_BIT);
16     SetBit(GPIOD->PCOR, LED_BLUE_BIT);
17     delay_ms(PULSE_WIDTH);
18
19     // Blue Color:
```

```
20      SetBit(GPIOB->PCOR, LED_RED_BIT);
21      SetBit(GPIOB->PCOR, LED_GREEN_BIT);
22      SetBit(GPIOD->PSOR, LED_BLUE_BIT);
23      delay_ms(PULSE_WIDTH);
24
25      // Yellow Color:
26      SetBit(GPIOB->PSOR, LED_RED_BIT);
27      SetBit(GPIOB->PSOR, LED_GREEN_BIT);
28      SetBit(GPIOD->PCOR, LED_BLUE_BIT);
29      delay_ms(PULSE_WIDTH);
30
31      // Magenta Color:
32      SetBit(GPIOB->PSOR, LED_RED_BIT);
33      SetBit(GPIOB->PCOR, LED_GREEN_BIT);
34      SetBit(GPIOD->PSOR, LED_BLUE_BIT);
35      delay_ms(PULSE_WIDTH);
36
37      // Cyan Color:
38      SetBit(GPIOB->PCOR, LED_RED_BIT);
39      SetBit(GPIOB->PSOR, LED_GREEN_BIT);
40      SetBit(GPIOD->PSOR, LED_BLUE_BIT);
41      delay_ms(PULSE_WIDTH);
42
43      // White Color:
44      SetBit(GPIOB->PSOR, LED_RED_BIT);
45      SetBit(GPIOB->PSOR, LED_GREEN_BIT);
46      SetBit(GPIOD->PSOR, LED_BLUE_BIT);
47      delay_ms(PULSE_WIDTH);
48    }
49  }
50
51  void init_pins(void) {
52    // Enable the clock for the PORTB and PORTD peripherals.
53    SetBit(SIM->SCGC5, CLOCK_PORTB_BIT);
54    SetBit(SIM->SCGC5, CLOCK_PORTD_BIT);
55
56    // Select the GPIO function for pins PTB18, PTB19 and PTD1.
57    SetBit(PORTB->PCR[LED_RED_BIT], MUX_GPIO_FUNCTION);
58    SetBit(PORTB->PCR[LED_GREEN_BIT], MUX_GPIO_FUNCTION);
59    SetBit(PORTD->PCR[LED_BLUE_BIT],  MUX_GPIO_FUNCTION);
60
61    // Make PTB18, PTB19 and PTD1 OUTPUT pins.
62    SetBit(GPIOB->PDDR, LED_RED_BIT);
63    SetBit(GPIOB->PDDR, LED_GREEN_BIT);
64    SetBit(GPIOD->PDDR, LED_BLUE_BIT);
65  }
66
67  void delay_ms(int ms) {
68    int i, j;
69    for(i = 0; i < ms; i++) {
70      for(j = 0; j < 7000; j++);
71    }
72  }
```

A função "delay_ms" não é ideal, já que deixa a CPU ocupada durante o tempo de espera. O cálculo do *delay* é simples:

$$\text{Tempo Gasto} = \text{Número de Ciclos} \times \text{Inverso da Frequência do Processador}$$

Tendo que o microcontrolador da placa KL25Z opera por padrão a uma frequência de 21 MHz, e assumindo que cada iteração do for consuma 3 ciclos do clock, temos:

$$\text{Tempo Gasto} = 3 \times \text{Número de Iterações} \times \frac{1}{21 \times 10^6}$$

Para calcularmos o número de iterações para um delay de 1ms, temos:

$$\text{Número de Iterações} = \text{Tempo gasto (em milisegundos)} \times \frac{21 \times 10^6}{3000}$$

$$1\text{ms} \times \frac{21 \times 10^6}{3000} = 7000 \text{ iterações}$$

Para obter a quantidade máxima de iterações apenas multiplicamos a quantidade de iterações do laço externo com o externo, obtendo então a fórmula: $ms \times 7000$.

```
void delay_ms(int ms) {
  for(int i = 0; i < ms; i++) {
    for(int j = 0; j < 7000; j++);
  }
}
```

4.2 Arquivo Rainbow Blinky

Abaixo temos o arquivo de cabeçalho, com os macros e diretivas de pré-processamento, que auxiliam na programação de um sistema embarcado, dando mais sentido aos códigos.

```
#ifndef RAINBOW_BLINKY_H
#define RAINBOW_BLINKY_H

// Bit Manipulation Macros:
#define SetBit(value, bit)    (value |=  (1 << bit))
#define ClearBit(value, bit)  (value &= ~(1 << bit))
#define ToggleBit(value, bit) (value ^=  (1 << bit))
#define TestBit(value, bit)   (value &   (1 << bit))

// RGB LEDs Interface Bits:
#define LED_RED_BIT   18  //PTB18
#define LED_GREEN_BIT 19  //PTB19
#define LED_BLUE_BIT   1  //PTD1

```

```
15 // Pin Control Bits:
16 #define MUX_GPIO_FUNCTION 8
17
18 // Clock Activation Bits:
19 #define CLOCK_PORTA_BIT 9
20 #define CLOCK_PORTB_BIT 10
21 #define CLOCK_PORTC_BIT 11
22 #define CLOCK_PORTD_BIT 12
23 #define CLOCK_PORTE_BIT 13
24
25 // Time Constant for the Blinking:
26 #define PULSE_WIDTH 250
27
28 // Initializes the pins used in the project.
29 extern void init_pins(void);
30
31 // Wait the given value in mili-seconds.
32 extern void delay_ms(int ms);
33
34 #endif
```

4.3 Blink RGB com SDK API

Abaixo está o mesmo código, só que usando a API que da NXP, o SDK. Ele facilita o desenvolvimento, abstraindo o uso de registradores. Obviamente ainda deve-se entender um pouco do microcontrolador para poder usá-lo.

Para usar o SDK devemos importar as bibliotecas, no formato "fsl_<periferico>.h", no caso, "fsl_clock.h", "fsl_gpio.h" e "fsl_port.h". Depois, devemos chamar as funções baseadas no periférico desejado, no formato "<PERIFERICO>_<Função>()". Constantes do SDK usadas no código sempre estão no formato "k<PERIFERICO>_<Descricao>', onde o "k" representa "constante.". O SDK se utiliza de structs para organizar o código e deixá-lo mais modular.

Na função de inicialização dos pinos usamos algumas funções importantes, sendo elas:

- CLOCK_EnableClock(clock_ip_name_t name): Habilita o clock para um periférico, identificado por uma constante. Ao chamar a função passando "kCLOCK_PortB" o clock para o periférico PORTB será ativado e ele poderá ser usado.
- PORT_SetPinMux(PORT_Type *base, uint32_t pin, port_mux_t mux): Configura o multiplexador do pino especificado. Passando "kPORT_MuxAsGpio" como o último argumento estaremos configurando o pino como entrada e saída digital.
- GPIO_PinInit(GPIO_Type *base, uint32_t pin, const gpio_pin_config_t *config): Inicializa um pino de GPIO como entrada ou saída, a partir de uma struct. Na struct definimos se é entrada, com "kGPIO_DigitalInput", ou saída, com "kGPIO_DigitalOutput", e o valor inicial do pino, se é 0 ou 1 (válido apenas para saída).

- GPIO_ClearPinsOutput(GPIO_Type *base, uint32_t mask): Limpar os bits do GPIO informado com base na máscara informada.
- GPIO_SetPinsOutput(GPIO_Type *base, uint32_t mask): Ativar os bits do GPIO informado com base na máscara informada.

```
#include "MKL25Z4.h"
#include "fsl_port.h"
#include "fsl_gpio.h"
#include "fsl_clock.h"

#define RED_LED    1 << 18
#define GREEN_LED    1 << 19
#define BLUE_LED  1 << 1
#define PULSE_WIDTH 50

/*
 * Delay the program in ms mili-seconds.
 */
void delay_ms(int ms);

/*
 * Initializes pins PTB18, PTB19 and PTD1 as digital output.
 */
void init_pins(void);

int main(void) {
  init_pins();
  while(1) {
    // Red Color:
    GPIO_ClearPinsOutput(GPIOB, RED_LED);
    GPIO_SetPinsOutput(GPIOB, GREEN_LED);
    GPIO_SetPinsOutput(GPIOD, BLUE_LED);
    delay_ms(PULSE_WIDTH);

    // Green Color:
    GPIO_SetPinsOutput(GPIOB, RED_LED);
    GPIO_ClearPinsOutput(GPIOB, GREEN_LED);
    GPIO_SetPinsOutput(GPIOD, BLUE_LED);
    delay_ms(PULSE_WIDTH);

    // Blue Color:
    GPIO_SetPinsOutput(GPIOB, RED_LED);
    GPIO_SetPinsOutput(GPIOB, GREEN_LED);
    GPIO_ClearPinsOutput(GPIOD, BLUE_LED);
    delay_ms(PULSE_WIDTH);

    // Yellow Color:
    GPIO_ClearPinsOutput(GPIOB, RED_LED);
    GPIO_ClearPinsOutput(GPIOB, GREEN_LED);
    GPIO_SetPinsOutput(GPIOD, BLUE_LED);
    delay_ms(PULSE_WIDTH);

```

```
48      // Magenta Color:
49      GPIO_ClearPinsOutput(GPIOB, RED_LED);
50      GPIO_SetPinsOutput(GPIOB, GREEN_LED);
51      GPIO_ClearPinsOutput(GPIOD, BLUE_LED);
52      delay_ms(PULSE_WIDTH);
53
54      // Cyan Color:
55      GPIO_SetPinsOutput(GPIOB, RED_LED);
56      GPIO_ClearPinsOutput(GPIOB, GREEN_LED);
57      GPIO_ClearPinsOutput(GPIOD, BLUE_LED);
58      delay_ms(PULSE_WIDTH);
59
60      // White Color:
61      GPIO_ClearPinsOutput(GPIOB, RED_LED);
62      GPIO_ClearPinsOutput(GPIOB, GREEN_LED);
63      GPIO_ClearPinsOutput(GPIOD, BLUE_LED);
64      delay_ms(PULSE_WIDTH);
65    }
66  }
67
68  void init_pins(void) {
69    // Enable clock for PORTB and PORTD peripherals.
70    CLOCK_EnableClock(kCLOCK_PortB);
71    CLOCK_EnableClock(kCLOCK_PortD);
72
73    // Select GPIO function for pins PTB18, PTB19 and PTD1.
74    PORT_SetPinMux(PORTB, 18, kPORT_MuxAsGpio);
75    PORT_SetPinMux(PORTB, 19, kPORT_MuxAsGpio);
76    PORT_SetPinMux(PORTD,  1, kPORT_MuxAsGpio);
77
78    // Config GPIO peripheral at PTB18 as OUTPUT.
79    const gpio_pin_config_t ptb18_config = {kGPIO_DigitalOutput, 0};
80    GPIO_PinInit(GPIOB, 18, &ptb18_config);
81
82    // Config GPIO peripheral at PTB19 as OUTPUT.
83    const gpio_pin_config_t ptb19_config = {kGPIO_DigitalOutput, 0};
84    GPIO_PinInit(GPIOB, 19, &ptb19_config);
85
86    // Config GPIO peripheral at PTD1 as OUTPUT.
87    const gpio_pin_config_t ptd1_config =  {kGPIO_DigitalOutput, 0};
88    GPIO_PinInit(GPIOD,  1, &ptd1_config);
89  }
90
91  void delay_ms(int ms) {
92    int i, j;
93    for(i = 0; i < ms; i++) {
94      for(j = 0; j < 7000; j++);
95    }
96  }
```

Abaixo, na figura 9, vemos o resultado dos dois códigos acima.

Figura 9: Blink RGB

5 Acendendo um LED ao Clique de um Botão

Dessa vez queremos ligar um LED ao clicarmos em um botão. Na prática temos dois botões na *protoboard* e queremos ligá-los ao pressionar um botão. Cada botão liga um LED. Iremos, então, obter uma tensão no pino, e, via software, ligar o LED.

5.1 Botão de Pull-down

Utilizaremos uma técnica bem simples para ligar o LED: o botão de pull-down. Devido à tecnologia CMOS não podemos deixar o um pino com entrada flutuante, isso quer dizer, deixar o pino desconectado de uma referência de potencial elétrico, como o Vcc ou o GND.

Quando conectamos o pino ao Vcc por meio de um botão, ou seja, uma chave, e a deixamos aberta, o pino fica desconectado de qualquer referência, tendo um nível lógico flutuante, ou seja, não temos como saber ao certo seu nível lógico. Dito isso precisamos dar um jeito de estabelecer o nível lógico do botão nos dois casos. Para isso usamos um botão de pull-down.

Ligar o botão ao Vcc e ao circuito não é o suficiente, devemos colocar um resistor, ligado ao GND, em paralelo com o circuito. Dessa forma estabelecemos um nível lógico baixo quando a chave está aberta, como mostra a figura 10.

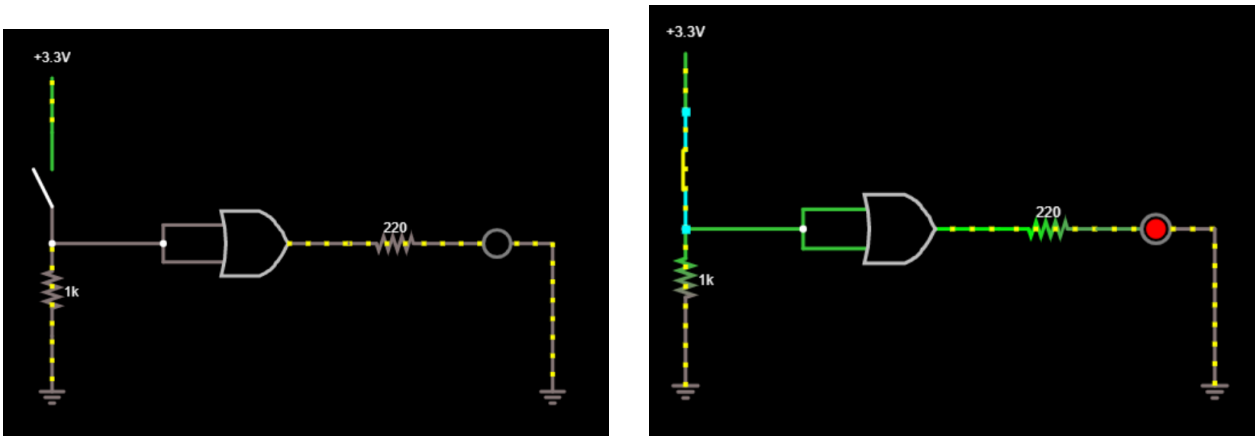

Figura 10: Botão de Pull-down

A porta lógica OR foi usada para simular a alta-impedância do pino do microcontrolador. A tensão do resistor, quando a chave está aberta, é emitida ao circuito (no nosso caso o pino), sendo 0V. Quando a chave é fechada é emitido a tensão Vcc ao resistor e ao circuito.

5.2 Verificando os Botões com Registradores

A figura 11 ilustra o circuito utilizado na atividade prática:

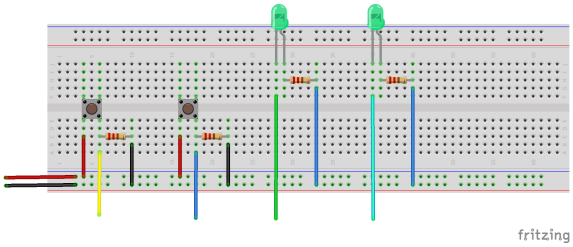

Figura 11: Circuito usado para ligar os LEDs ao toque dos Botões

Usamos resistores aqui para alguns fins: resistir à corrente e não queimar os LEDs e no botão, para estabelecer e uma tensão entre o pino e o terra (mandando 0V para o pino), logo, um resistor de **pull-down**.

Segue o código criado para fornecer o funcionamento do sistema que controla o acionamento dos LEDs via sinal externo - no caso, o pressionamento de um botão.

A única diferença desse código com o anterior é que no laço infinito ficamos verificando constantemente o nível do pino ligado ao botão. Se o pino ligado ao botão estiver em nível lógico alto (realizamos essa verificação no registrador PDIR), trocamos o nível do LED (usando o registrador PTOR). A verificação é feita usando uma operação AND bit-a-bit. Definimos, então, o pino do botão como entrada digital, colocando zero em seu respectivo bit no registrador PDDR.

```c
#include "MKL25Z4.h"
#include "Switch_Led_Couple.h"

int main(void) {
  init_pins();
  while(1) {
    if(TestBit(GPIOD->PDIR, SW1_BIT)) {
      // If SW1 is pressed, toggle the level of LED D1.
      SetBit(GPIOC->PTOR, LED_GREEN_D1_BIT);
    }
    if(TestBit(GPIOA->PDIR, SW2_BIT)) {
      // If SW2 is pressed, toggle the level of LED D3.
      SetBit(GPIOA->PTOR, LED_GREEN_D3_BIT);
    }
  }
}

void init_pins(void) {
  // Enable the clock to the used ports.
  SetBit(SIM->SCGC5, CLOCK_PORTA_BIT);
  SetBit(SIM->SCGC5, CLOCK_PORTC_BIT);
  SetBit(SIM->SCGC5, CLOCK_PORTD_BIT);

  // Configures the pins as GPIO.
  SetBit(PORTC->PCR[LED_GREEN_D1_BIT], MUX_GPIO_FUNCTION);
  SetBit(PORTA->PCR[LED_GREEN_D3_BIT], MUX_GPIO_FUNCTION);
  SetBit(PORTD->PCR[SW1_BIT], MUX_GPIO_FUNCTION);
  SetBit(PORTA->PCR[SW2_BIT], MUX_GPIO_FUNCTION);

  // Configure the LED pins as OUTPUT.
  SetBit(GPIOC->PDDR, LED_GREEN_D1_BIT);
  SetBit(GPIOA->PDDR, LED_GREEN_D3_BIT);

  // Configure the SWITCH pins as INPUT.
  ClearBit(GPIOD->PDDR, SW1_BIT);
  ClearBit(GPIOA->PDDR, SW2_BIT);
}

```

Na função *init_pins()*, usamos o registrador SCGC5 para habilitar o clock para as portas A, C e D. Os bits responsáveis por emitir uma tensão (PTC8 e PTA13), e os bits responsáveis por receber sinais externos (PTD4 e PTA12) são configurados para a função GPIO, onde os dois primeiros terão função de saída - os LED's são conectados neles - e os outros dois terão função de entrada - receberão o sinal do botão.

Na função *main()*, se o valor no pino PTD4 for 1, significa que o botão foi pressionado e então a operação **toggle** é efetuada no pino do LED (PTC8), ou seja, sendo o LED aceso se seu nível atual for baixo, e apagado se seu nível for alto). O mesmo ocorre para o outro botão e LED.

Abaixo se encontra o arquivo cabeçalho, com os macros:

```
#ifndef SWITCH_LED_COUPLE_H
#define SWITCH_LED_COUPLE_H

// Bit Manipulation Macros:
#define SetBit(value, bit)    (value |=  (1 << bit))
#define ClearBit(value, bit)  (value &= ~(1 << bit))
#define ToggleBit(value, bit) (value ^=  (1 << bit))
#define TestBit(value, bit)   (value &   (1 << bit))

// RGB LEDs Interface Bits:
#define LED_RED_BIT   18  //PTB18
#define LED_GREEN_BIT 19  //PTB19
#define LED_BLUE_BIT   1  //PTD1

// Green LEDs Interface Bits:
#define LED_GREEN_D1_BIT  8   //PTC8
#define LED_GREEN_D2_BIT  9   //PTC9
#define LED_GREEN_D3_BIT 13   //PTA13
#define LED_GREEN_D4_BIT  5   //PTD5

// Switch Interface Bits:
#define SW1_BIT  4    //PTD4
#define SW2_BIT 12    //PTA12
#define SW3_BIT  4    //PTA4
#define SW4_BIT  5    //PTA5

// Pin Control Bits:
#define MUX_GPIO_FUNCTION 8

// Clock Activation Bits:
#define CLOCK_PORTA_BIT 9
#define CLOCK_PORTB_BIT 10
#define CLOCK_PORTC_BIT 11
#define CLOCK_PORTD_BIT 12
#define CLOCK_PORTE_BIT 13

// Initializes the pins used in the project.
extern void init_pins(void);

// Wait the given value in mili-seconds.
extern void delay_ms(int ms);

#endif

```

5.3 Verificando Botões com a SDK API

Abaixo temos o mesmo código, só que usando a SDK API, da NXP. Dessa forma temos um código mais legível e modularizado. Temos poucas diferenças do código passado (com SDK):

- Configuramos o pino dos botões com **kGPIO_DigitalInput**, para que possamos ler a tensão.

- Verificamos se o botão foi, ou não, apertado por meio da função GPIO_PinRead (GPIO_Type *base, uint32_t mask). Se sim, retorna 1 e altera o valor do LED.

```
#include "Switch_Led_Couple.h"

#include "MKL25Z4.h"

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

int main(void) {

  init_pins();
  while(1) {
    if(GPIO_ReadPinInput(GPIOD, SW1_BIT)) {
      GPIO_TogglePinsOutput(GPIOC, 1 << LED_GREEN_D1_BIT);
    }
    if(GPIO_ReadPinInput(GPIOA, SW2_BIT)) {
      GPIO_TogglePinsOutput(GPIOA, 1 << LED_GREEN_D3_BIT);
    }
  }
}

void init_pins(void) {
  // Enable the clock to the used ports.
  CLOCK_EnableClock(kCLOCK_PortA);
  CLOCK_EnableClock(kCLOCK_PortC);
  CLOCK_EnableClock(kCLOCK_PortD);

  // Configures the pins as GPIO.
  PORT_SetPinMux(PORTC, LED_GREEN_D1_BIT, kPORT_MuxAsGpio);
  PORT_SetPinMux(PORTA, LED_GREEN_D3_BIT, kPORT_MuxAsGpio);
  PORT_SetPinMux(PORTD, SW1_BIT, kPORT_MuxAsGpio);
  PORT_SetPinMux(PORTA, SW2_BIT, kPORT_MuxAsGpio);

  // Configure the LED pins as OUTPUT.
  gpio_pin_config_t led_green_d1 = {kGPIO_DigitalOutput, 0};
  GPIO_PinInit(GPIOC, LED_GREEN_D1_BIT, &led_green_d1);
  gpio_pin_config_t led_green_d3 = {kGPIO_DigitalOutput, 0};
  GPIO_PinInit(GPIOA, LED_GREEN_D3_BIT, &led_green_d3);

  // Configure the SWITCH pins as INPUT.
  gpio_pin_config_t sw1 = {kGPIO_DigitalInput, 0};
  GPIO_PinInit(GPIOD, SW1_BIT, &sw1);
  gpio_pin_config_t sw2 = {kGPIO_DigitalInput, 0};
  GPIO_PinInit(GPIOA, SW2_BIT, &sw2);
}
```

5.4 Circuito com Filtro RC

O circuito mostrado acima tem um problema sério, que é a não absorção de um sinal indesejado no momento do pressionamento do botão. Esse sinal é caracterizado por uma espécie de ruído no sinal, também conhecido como trepidação, como é mostrado na figura 12:

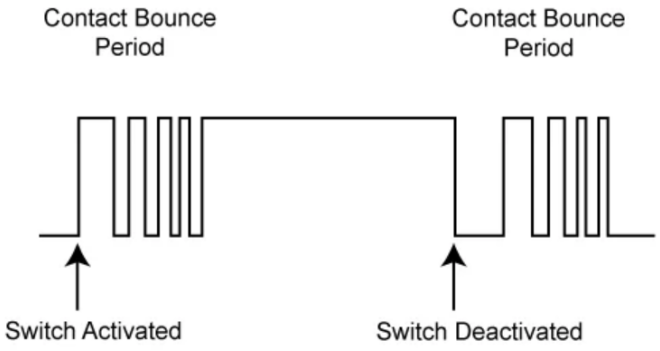

Figura 12: Efeito de Trepidação no Sinal do Botão

Esse problema pode fazer com quem o sistema verifique várias vezes que o botão foi pressionado. Esse problema é ocasionado pela limitação mecânica do botão, e agravado quando o próprio é muito usado. Esse efeito é conhecido como *bounce*. Para resolver esse problema, empregamos alguma técnica de *debounce*, seja por *sofwtare*, como um delay, que irá esperar o sinal estabilizar, ou *hardware*, como um filtro.

Para contornar esse problema, aplicamos uma técnica de *hardware*: o filtro passa-baixa, ou filtro RC. Essa solução consiste na implementação de capacitores no circuito em paralelo com as chaves, como mostra a figura 13, os quais absorvem o sinal indesejado no momento do pressionamento do botão.

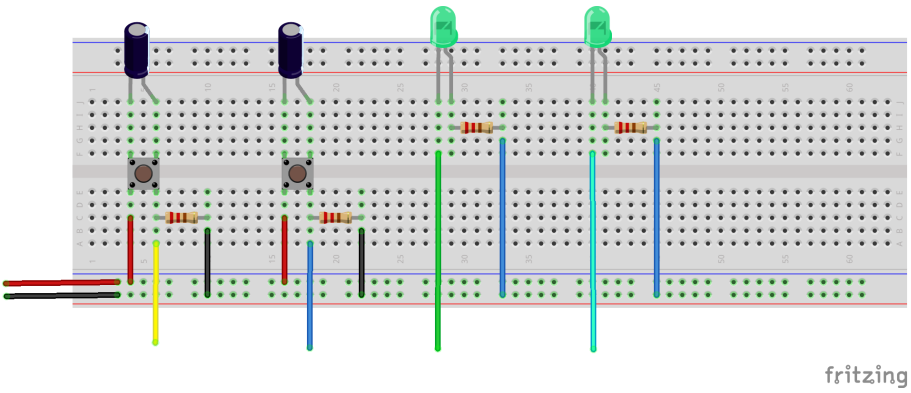

Figura 13: Circuito com o Filtro Passa-Baixa

6 Conclusão

Ao realizarmos os itens da atividade prática podemos ter uma noção melhor de como funciona a programação para microcontroladores. A configuração de GPIO é uma das etapas mais básicas na programação de sistemas embarcados, utilizada para atuar e ler níveis digitais.

Outra habilidade aprendida pela equipe foi a leitura do manual do processador e o datasheet do microcontrolador, além da aquisição de aptidão na codificação de *firmware* utilizando lógica binária, e ferramentas adicionais, como a API da NXP, o SDK.

Ao final podemos concluir que a realização das práticas fixou e melhorou os ensinamentos passados nas aulas teóricas, o que é essencial na formação de um engenheiro.

Referências

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