

Relatório do Experimento 2

Ferramentas de Desenvolvimento de Software

Henrique Aires Silva
RA: 169574 - Turma T
21 de Março de 2017

1 Introdução

Uma das etapas mais importantes do desenvolvimento de softwares embarcados é a rotina de testes (debug) junto ao Hardware selecionado. Esta tarefa nem sempre é a mais trivial, dado que o microcontrolador em que se deseja executar o software quase sempre não é o mesmo no qual se desenvolve. Além disso, por ser um ambiente de muito baixo nível lógico no sentido de abstrações, é praticamente inviável testar o software desejado analisando as transações entre registradores internos do microcontrolador e mudanças lógicas de seus pinos.

Para amenizar alguns deste problemas e tornar a tarefa de desenvolvimento de software embarcado mais simples e acessível, foram criados softwares chamados de **IDEs** (Integrated Development Environments - Ambientes de Desenvolvimento Unificado) que tem por premissa abstrair o acesso aos registradores e informações de baixo nível do microcontrolador de modo que uma pessoa com mínimos conhecimentos técnicos na área consiga com pouco esforço testar e corrigir, se necessário, seu software, focando seu esforço no desenvolvimento do algoritmo.

O IDE utilizado no curso de EA869 é conhecido como **CodeWarrior**[1], desenvolvido e mantido pela NXP (antiga Freescale). Nele estão presentes ferramentas de gerenciamento de projetos de software, editor de código, compilador e debugger, tudo em um mesmo ambiente.

Esta ferramenta é capaz de interfacear com o hardware da placa **FRDM-KL25**[2], que possui como componente principal o microcontrolador **MKL25Z128**[3], além de diversos periféricos, como conversor USB-Serial, sensor capacitivo de toque, diversos pinos de propósito geral (GPIO) e um LED RGB, que será o foco deste experimento.

2 Objetivo

Durante este experimento, procurou-se a familiarização do aluno com a ferramenta de desenvolvimento de software embarcado CodeWarrior.

Para alcançar tal objetivo, foi proposto que fosse criado um simples programa que controle um LED RGB e pisque na cor branca (todas as cores acesas) em um intervalo de tempo fixo.

Com o objetivo de estimular boas práticas de programação e incentivar o reuso de código, foi proposto que fosse desenvolvido um código modular, ou seja, com arquivos separados por função objetivo, de tal modo que este possa ser utilizado em outros projetos da disciplina no futuro caso necessário.

3 Metodologia

3.1 LED RGB

O LED RGB é composto por 3 LEDs de cores distintas (Azul, Verde e Vermelho) no mesmo encapsulamento com um anodo comum como mostrado na Figura 1. Desta forma é possível acioná-los independentemente controlando o nível de tensão em seu catodo. Por estarem no mesmo encapsulamento é possível gerar diversas combinações de cores acionando os LEDs em conjunto, dado que sua luz emitida será difundida e misturada.

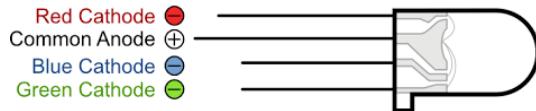


Figura 1: Esquema elétrico de um LED RGB com anodo comum

No escopo deste experimento, não será explorado o conceito de modulação de pulso, que pode ser utilizado para gerar diversos tons de cores com apenas as 3 cores primárias, apenas o acionamento lógico (ligado ou desligado) dos catodos. Assim, temos um total de 7 cores distintas possíveis.

Para gerar uma luz branca por exemplo, é possível acionar todos os 3 LEDs.

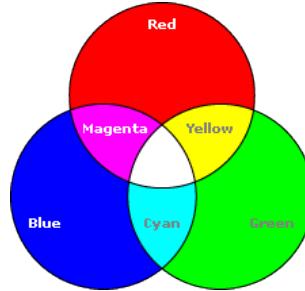


Figura 2: Diagrama mostrando as possíveis combinações de cores sólidas de um LED RGB

3.1.1 Acionamento do LED RGB

A conexão elétrica do LED RGB de anodo comum na placa MLK25Z128 segue um padrão bem conhecido, onde tem o pino comum ligado na alimentação positiva, e cada anodo sendo ligado em um pino independente do microcontrolador, com um resistor em série para limitar a corrente.

Este esquema é particularmente proveitoso pois ele contorna uma das dificuldades de todos os microcontroladores da atualidade, que é a capacidade baixa de fornecimento de corrente direto em pinos lógicos. Ou seja, se ligássemos o LED com o anodo ligado no pino lógico e o catodo no GND, dependendo do valor do resistor limitador, o controlador não conseguiria fornecer corrente suficiente para fazer o LED brilhar.

Por outro lado, os pinos lógicos têm uma grande capacidade de absorção de corrente, por conta de sua construção interna. Assim, podemos fazer a ligação mostrada na figura 3

Desta forma, teremos uma lógica invertida no software, ou seja, para acionar o LED devemos escrever o nível lógico "0" no pino correspondente. Analogamente, para apagar o LED, basta escrever "1" em seu pino.

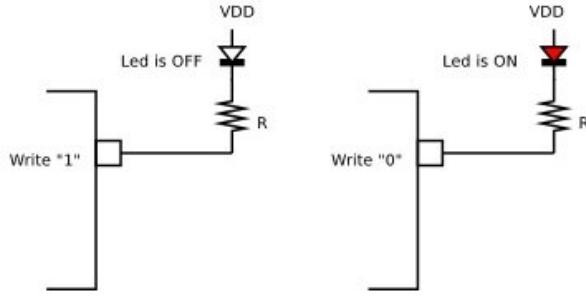


Figura 3: Esquemático de uma ligação de um LED em um controlador usando lógica invertida

3.2 Atrasos

Como microcontroladores possuem períodos de relógio extremamente rápidos, da ordem de dezenas de nanosegundos, estes executam seus algoritmos de maneira quase imperceptível aos humanos.

Para que consigamos interpretar e interagir com o hardware, de forma a percebemos eventuais problemas, devemos fazer com o microcontrolador execute funções de atraso entre certos comandos que tem por objetivo a percepção humana, como o acionamento de um LED ou escrita de uma mensagem em um display.

Tais rotinas apenas fazem com que o processador fique parado num laço finito por um determinado tempo, sem realizar mais nenhum comando (não entram no escopo deste experimento as interrupções assíncronas), assim é possível que uma pessoa distingua o piscar de um LED por exemplo.

4 Proposta de Solução

Para implementar o software que pisca um LED RGB na cor branca, é necessário realizar uma série de inicializações e configurações do hardware, dado que após o Reset, o microcontrolador não tem como saber em que estado o hardware se encontra.

Neste desenvolvimento foi necessário utilizar funções apenas do módulo **GPIO** do controlador, dado que é apenas necessário alterar o nível lógico dos pinos conectados ao LED.

4.1 Inicialização

Visando a redução do consumo de energia do microcontrolador, este foi fabricado de tal forma que fosse possível desabilitar o sinal de clock para cada módulo que não estivesse em uso. Este controle é feito através do conjunto de registradores **SIM_SCGC** (System Integration Module - System Clock Gating Control).

Assim, após o ínicio do programa, precisamos habilitar o clock para o periférico **GPBIO_B** e **GPBIO_D** setando, respectivamente, os bits 10 e 12 através do registrador **SIM_SCGC5**, como mostrado na figura 4.

Como cada pino do microcontrolador pode ter muitas funções, devemos especificar que usaremos os pinos ligados ao LED como GPIO. Para isso, devemos escrever o valor "b001" nos bits 10, 9 e 8 do registrador **PORTx_PCR** (Pin Configuration Register), que configuram o "MUX" das funções, mostrado na figura 5

Também é necessário configurar a direção dos pinos, pois este podem ser tanto Entrada (Input) como Saída (Output). Para isso, é preciso escrever no registrador **GPIOx_PDDR** (Port Data Direction Register) o valor "1" na posição correspondente do pino, seguindo a formatação mostrada na figura 6.

Para facilitar o reuso deste código de inicialização, foi criada uma função genérica que inicializa um pino por vez do LED RGB, recebendo então um valor inteiro e traduzindo-o em uma tabela da verdade para identificar os registradores que serão acessados.

Também foi criada uma função que inicializa todos os 3 pinos do LED RGB, sendo apenas um encapsulamento da função de inicialização citada anteriormente dentro de um laço.

Address: 4004_7000h base + 1038h offset = 4004_8038h																	
Bit	31	30	29	28	27	26	25	24		23	22	21	20	19	18	17	16
R									0					0		0	
W																	
Reset	0	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		PORTE	PORTD	PORTC	PORTB	PORTA		1	0		TSI		0	0		LPTMR
W			0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 4: Formatação do registrador SIM_SCGC5, mostrando os bits de ativação de clock para todos os bancos GPIO

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				0				
W										w1c				IRQC			
Reset	0	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					x*	x*	x*		0	x*	0	x*	0	x*	x*	x*	
Reset	0	0	0	0	0	0	x*	x*	x*	0	x*	0	x*	x*	x*	x*	

* Notes:
• x = Undefined at reset.

Figura 5: Formatação do registrador PORTx_PCR, onde pode-se ver os bits que configuram a função do pino (MUX)

Address: Base address + 14h offset																	
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
R														PDD			
W																	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
GPIOx_PDDR field descriptions																	
Field	Description																
31–0 PDD	Port Data Direction Configures individual port pins for input or output. 0 Pin is configured as general-purpose input, for the GPIO function. 1 Pin is configured as general-purpose output, for the GPIO function.																

Figura 6: Formatação do registrador GPIOx_PDDR, mostrando a configuração da direção de cada pino GPIO

4.1.1 Pseudo-código

Um possível pseudo-código para a rotina de inicialização dos pinos do LED RGB é a mostrada a seguir.

INICIO (inicializa_led)

Entrada: cor - Identificador (número inteiro) da cor do LED

Saída: Pino do LED inicializado

CASO cor

vermelho:

 Seta em 1 o bit 10 do registrador SIM_SCGC5

 Seta em 1 o bit 8 do registrador PORTB_PCR18

 Seta em 0 os bits 9 e 10 do registrador PORTB_PCR18

 Seta em 1 o bit 18 do registrador GPIOB_PDDR

```

verde:
    Seta em 1 o bit 10 do registrador SIM_SCGC5
    Seta em 1 o bit 8 do registrador PORTB_PCR19
    Seta em 0 os bits 9 e 10 do registrador PORTB_PCR19
    Seta em 1 o bit 19 do registrador GPIOB_PDDR

azul:
    Seta em 1 o bit 12 do registrador SIM_SCGC5
    Seta em 1 o bit 8 do registrador PORTD_PCR1
    Seta em 0 os bits 9 e 10 do registrador PORTD_PCR1
    Seta em 1 o bit 1 do registrador GPIOD_PDDR

FIM CASO
FIM

```

Temos também o pseudo-código da função que inicializa todos os pinos do LED RGB:

```

INICIO (inicializa_led_rgb)
    Entrada: Nada
    Saída: Todos os pino do LED RGB inicializados

    PARA( cor MENOR QUE 3 )
        CHAMA incializa_led COM cor
        INCREMENTA cor
    FIM PARA
FIM

```

4.2 Atuação no LED

Para atuar no LED RGB, ou seja, fazê-lo piscar precisamos escrever em registradores específicos que controlam o estado dos GPIOs.

Neste caso temos 3 registradores diferentes para cada porta GPIO:

- **GPIOx_PSOR (Port Set Output Register)** mostrado na figura 7, usado para colocar o pino selecionado em nível alto ("1"), ou seja, apagar o LED

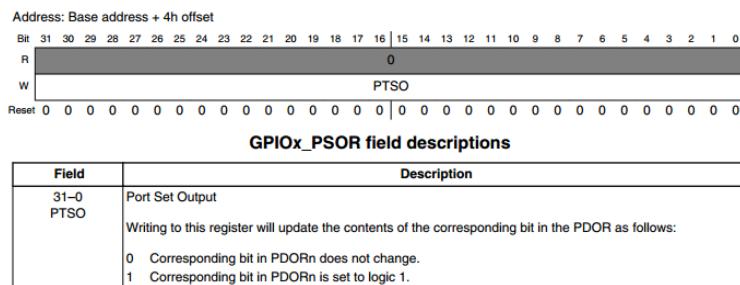


Figura 7: Formatação do registrador GPIOx_PSOR

- **GPIOx_PCOR (Port Clear Output Register)** mostrado na figura 8, usado para colocar o pino selecionado em nível baixo ("0"), ou seja, acender o LED
- **GPIOx_PTOR (Port Toggle Output Register)** mostrado na figura 9, usado para inverter o estado lógico do pino selecionado

Novamente, para poder utilizar este módulo que controla o LED RGB no futuro, podem ser criadas 3 funções distintas: acender o LED, apagá-lo e inverter seu estado.

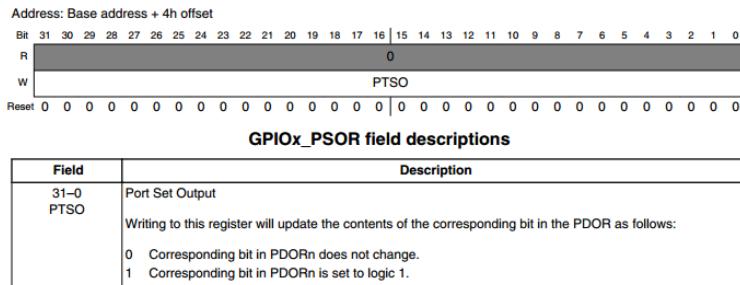


Figura 8: Formatação do registrador GPIOx_PCOR

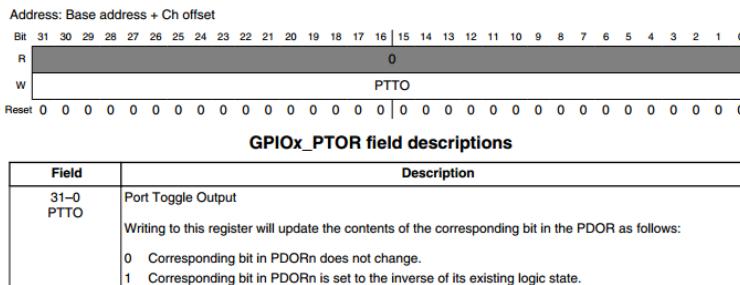


Figura 9: Formatação do registrador GPIOx_PTOR

4.2.1 Pseudo-código

O Pseudo-código para a função que acende o LED pode ser escrito como a seguir. Vale lembrar que os registradores específicos e os pinos respectivos ao LED RGB serão definidos em uma estrutura separada no código em C.

- Função para acender o LED:

INICIO (acende_led)

Entrada: Identificador (número inteiro) da cor do LED

Saída: Pino do LED em estado lógico baixo ("0")

Seta em 1 o bit correspondente ao pino do LED no registrador GPIOx_PCOR
FIM

- Função para apagar o LED:

INICIO (apaga_led)

Entrada: Identificador (número inteiro) da cor do LED

Saída: Pino do LED em estado lógico alto ("1")

Seta em 1 o bit correspondente ao pino do LED no registrador GPIOx_PSOR

- Função para inverter o LED:

INICIO (inverte_led)

Entrada: Identificador (número inteiro) da cor do LED
Saída: Pino do LED em estado lógico alto invertido

Seta em 1 o bit correspondente ao pino do LED no registrador GPIOx_PTOR
FIM

4.3 Atrasos

Há várias formas de implementar rotinas de atraso em um ambiente embarcado. A mais simples e a que será explorada neste experimento é feita por um laço no qual uma variável é decrementada até 0, retornando desta rotina apenas no fim deste laço.

Neste experimento o número de iterações do laço decrementativo foi obtido empiricamente, mas é possível realizar o cálculo do tempo exato em que a rotina ficará em execução a partir do valor da frequência de clock do microcontrolador.

4.3.1 Pseudo-código

O Pseudo-código da função de atraso é como a seguir:

```
INICIO (Delay)
    Entrada: i - Número de iterações
    Saída: Nada

    ENQUANTO( i DIFERENTE DE 0 )
        DECREMENTA i
    FIM ENQUANTO
FIM
```

4.4 Laço principal - Main

A função main contém o laço principal do programa e é chamada logo após a inicialização de baixo nível da memória.

Como todo o acesso aos registradores foi abstraído no módulo do LED RGB, a função main apenas faz chamadas às suas subrotinas.

4.4.1 Pseudo-Código

Um pseudo-código para a função main é o seguinte:

```
INICIO
    CHAMA inicializa_led_rgb
    ENQUANTO( verdadeiro )
        CHAMA inverte_led com 'vermelho'
        CHAMA inverte_led com 'verde'
        CHAMA inverte_led com 'azul'
        CHAMA delay com 500000
    FIM ENQUANTO
FIM
```

5 Testes

Utilizando o IDE do CodeWarrior para realizar os testes, foi possível acompanhar a execução do programa passo-a-passo e resolver eventuais problemas de implementação do código em C.

5.1 Registradores

Uma das ferramentas do CodeWarrior utilizada para debug foi a aba "Registers", onde é possível ver e editar o valor em tempo real de todos os registradores do microcontrolador. Agrupando todos os que foram utilizados neste experimento, temos na figura 10 seus valores antes da chamada da função de inicialização dos pinos do LED RGB. Já na figura 11 vemos destacado em amarelo os valores alterados dos registradores de configuração (SIM_SCGC5, PORTx_PCRy e GPIOx_PDDR).

Name	Value	Location
> Serial Peripheral Interface (SPI0)		
> Serial Peripheral Interface (SPI1)		
✓ Debug_RGB_LED		
SIM_SCGC5	0x00000180	0x40048038
PORTB_PCR18	0x00000000	0x4004a048
PORTB_PCR19	0x00000000	0x4004a04c
PORTD_PCR1	0x00000000	0x4004c004
GPIOB_PSOR	non-readable	0x400ff044
GPIOB_PCOR	non-readable	0x400ff048
GPIOB_PTOR	non-readable	0x400ff04c
GPIOB_PDDR	0x00000000	0x400ff054
GPIOD_PSOR	non-readable	0x400ff0c4
GPIOD_PCOR	non-readable	0x400ff0c8
GPIOD_PTOR	non-readable	0x400ff0cc
GPIOD_PDDR	0x00000000	0x400ff0d4

Figura 10: Estado dos registradores de configuração do LED RGB antes da chama da função de inicialização

5.2 Osciloscópio

Também foi utilizado um osciloscópio comum para analisar a forma de onda gerada pelo pino do LED RGB.

Como o pino do Catodo Azul do LED (PTD1) está roteado na placa também para um conector externo (J2 - pino 12) foi possível medir a tensão neste ponto com a ponta de prova do osciloscópio, mostrada na figura 12.

Na imagem do osciloscópio é possível ver também a medida do período da onda, que está relacionado diretamente ao valor inserido na função de atraso. No caso temos um período total de 480 ms, ou seja, o LED fica 240 ms ligado e o mesmo tempo desligado.

Como foi utilizada na implementação do programa a função de "toggle" do pino para termos um código mais enxuto, temos aqui um Duty Cycle de 50% no LED.

Name	Value	Location
> Serial Peripheral Interface (SPI0)		
> Serial Peripheral Interface (SPI1)		
✓ Debug_RGB_LED		
SIM_SCGC5	0x00001580	0x40048038
PORTB_PCR18	0x000000105	0x4004a048
PORTB_PCR19	0x000000105	0x4004a04c
PORTD_PCR1	0x000000105	0x4004c004
GPIOB_PSOR	non-readable	0x400ff044
GPIOB_PCOR	non-readable	0x400ff048
GPIOB_PTOR	non-readable	0x400ff04c
GPIOB_PDDR	0x000c0000	0x400ff054
GPIOD_PSOR	non-readable	0x400ff0c4
GPIOD_PCOR	non-readable	0x400ff0c8
GPIOD_PTOR	non-readable	0x400ff0cc
GPIOD_PDDR	0x00000002	0x400ff0d4

Figura 11: Estado dos registradores de configuração do LED RGB depois da chama da função de inicialização

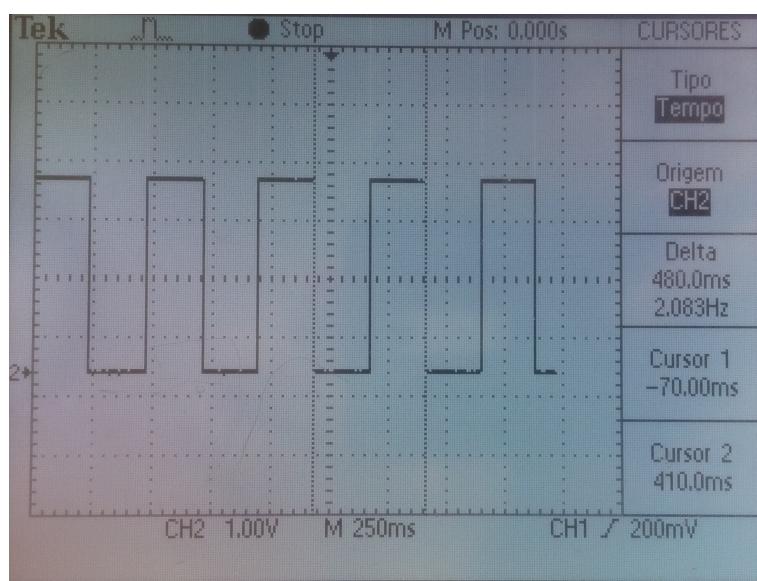


Figura 12: Forma de onda da tensão do pino 12 do conector J2 da placa FRDM-KL25Z que está ligado ao catodo azul do LED RGB

6 Conclusões

A solução proposta para o experimento foi mais que suficiente para alcançar o objetivo final de piscar o LED RGB na cor branca. Tal atividade foi útil particularmente para embasar a criação de um módulo de controle para o LED RGB e de um módulo de funções de atraso que possam ser utilizados em futuros projetos da disciplina.

O módulo do LED RGB foi criado de modo a não serem precisos futuros desenvolvimentos, já que todas as funções necessárias a ele já foram implementadas (inicialização por cor ou de todo o led, acender, apagar e inverter), assim, ele está pronto para ser apenas incluído em outros projetos.

Um possível desenvolvimento do módulo de atraso consiste em relacionar o valor de Clock do relógio do microcontrolador com o número de iterações necessárias, ou seja, criar uma função que receba como argumento um valor de tempo e não de iterações.

O código foi todo documentado no estilo Doxygen [5] para que fosse possível gerar uma documentação em estilo profissional do projeto sem muito esforço.

Todos os códigos desenvolvidos nesta disciplina foram hospedados no repositório do aluno no GitHub [4].

Referências

- [1] Página inicial do CodeWarrior - Acessado em 21/03/2017
http://www.nxp.com/products/software-and-tools/software-development-tools/codewarrior-development-tools:CW_HOME
- [2] Manual da placa FRMD-KL25Z - Acessado em 21/03/2017
<http://www.seeedstudio.com/document/pdf/FRMD-KL25Z.pdf>
- [3] Manual do microcontrolador Kinetis KL25Z128 - Acessado em 21/03/2017
http://cache.freescale.com/files/32bit/doc/ref_manual/KL25P80M48SF0RM.pdf
- [4] Re却tório do GitHub do aluno com os códigos da disciplina - Acessado em 21/03/2017
<https://github.com/henrique-silva/ea871>
- [5] Guia de documentação de código no estilo Doxygen - Acessado em 21/03/2017
<https://www.stack.nl/~dimitri/doxygen/manual/docblocks.html>

A Código Fonte - main.c

```
/*
 * @file main.c
 * @brief Pisca-pisca de um led rgb na cor branca com codigo modular
 *
 * @mainpage Projeto de Pisca-Pisca RGB Branco com codigo modular
 * Projeto referente ao Roteiro 2 da disciplina EA871, baseado no modelo fornecido por Wu,
 *
 * @author Henrique Aires Silva
 * @date 22/03/2017
 *
 */

/*
 */
#include "led_rgb.h"
#include "delay.h"

/*
 * @brief Led RGB na cor branca piscante
 * @return 1 somente para satisfazer a sintaxe C
 */
int main( void )
{
    /*! Inicializa os pinos dos LEDs RGB e apaga todos */
    led_init_rgb();

    /*!
     * Laçamento para produzir efeito piscante
     */
    for (;;)
    {
        delay( 500000 );           /*! \li Espera um tempo */
        /*! Inverte o estado dos bits do LED RGB */
        led_toggle( LED_RED );
        led_toggle( LED_GREEN );
        led_toggle( LED_BLUE );
    }

    return 1;
}
```

B Código Fonte - led_rgb.c

```
/*
 * @file led_rgb.c
 * @brief M oacute;dulo de controle do LED RGB
 *
 * @author Henrique Aires Silva
 * @date 22/03/2017
 *
 */

#include "led_rgb.h"

led_t led_rgb[3] = {
    [LED_RED] = {
        .psor = GPIOB_PSOR,
        .pcor = GPIOB_PCOR,
        .ptor = GPIOB_PTOR,
        .pin = LED_RED_PIN
    },
    [LED_GREEN] = {
        .psor = GPIOB_PSOR,
        .pcor = GPIOB_PCOR,
        .ptor = GPIOB_PTOR,
        .pin = LED_GREEN_PIN
    },
    [LED_BLUE] = {
        .psor = GPIOD_PSOR,
        .pcor = GPIOD_PCOR,
        .ptor = GPIOD_PTOR,
        .pin = LED_BLUE_PIN
    }
};

void led_on( led_color_t color )
{
    *(led_rgb[color].pcor) = led_rgb[color].pin;
}

void led_off( led_color_t color )
{
    *(led_rgb[color].psor) = led_rgb[color].pin;
}

void led_toggle( led_color_t color )
{
    *(led_rgb[color].ptor) = led_rgb[color].pin;
}

void led_init( led_color_t color )
{
```

```

switch ( color ) {

    case LED_RED:
        SIM_SCGC5 |= ( 1 << 10 ); /*! Habilita clock GPIO do PORTB (bit 10) */
        PORTB_PCR18 &= 0xFFFFF8FF; /*! Zera bits 10, 9 e 8 (MUX) de PTB18 */
        PORTB_PCR18 |= 0x00000100; /*! Seta bit 8 do MUX de PTB18, assim os 3 bits de MUX */
        GPIOB_PDDR |= (LED_RED_PIN); /*! Seta direccao do pino 18 de PORTB */
        break;

    case LED_GREEN:
        SIM_SCGC5 |= ( 1 << 10 ); /*! Habilita clock GPIO do PORTB (bit 10) */
        PORTB_PCR19 &= 0xFFFFF8FF; /*! Zera bits 10, 9 e 8 (MUX) de PTB19 */
        PORTB_PCR19 |= 0x00000100; /*! Seta bit 8 do MUX de PTB19, assim os 3 bits de MUX */
        GPIOB_PDDR |= (LED_GREEN_PIN); /*! Seta direccao do pino 19 de PORTB */
        break;

    case LED_BLUE:
        SIM_SCGC5 |= ( 1 << 12 ); /*! Habilita clock GPIO do PORTD (bit 12) */
        PORTD_PCR1 &= 0xFFFFF8FF; /*! Zera bits 10, 9 e 8 (MUX) de PTD1 */
        PORTD_PCR1 |= 0x00000100; /*! Seta bit 8 do MUX de PTD1, assim os 3 bits de MUX */
        GPIOD_PDDR |= (LED_BLUE_PIN); /*! Seta direccao do pino 1 de PORTD */
        break;

    default:
        return;
}

/*! Inicializa o LED apagado (n&iacute;vel l&oacute;gico 1) */
led_off( color );
}

void led_init_rgb( void )
{
    uint8_t led_count;
    for( led_count = 0; led_count < MAX_LED_RGB ; led_count++) {
        led_init( led_count );
    }
}

```

C Código Fonte - led_rgb.h

```
/*
 * @file led_rgb.h
 * @brief Abstracção do módulo de controle do LED RGB
 *
 * @author Henrique Aires Silva
 *
 */

#ifndef LED_RGB_H_
#define LED_RGB_H_

#include <stdint.h>

/*! @brief Registrador que habilita as portas do GPIO (Reg. SIM_SCGC5) */
#define SIM_SCGC5    (*(uint32_t *) 0x40048038)

/*! @brief MUX do pino PTB18 (Reg. PORTB_PCR18) */
#define PORTB_PCR18  (*(uint32_t *) 0x4004A048)
/*! @brief MUX do pino PTB18 (Reg. PORTB_PCR18) */
#define PORTB_PCR19  (*(uint32_t *) 0x4004A04C)
/*! @brief MUX do pino PTD1 (Reg. PORTD_PCR1) */
#define PORTD_PCR1   (*(uint32_t *) 0x4004C004)

/*! @brief Diretório dos dados nos pinos da porta PORTB (Reg. GPIOB_PDDR) */
#define GPIOB_PDDR   (*(uint32_t *) 0x400FF054)
/*! @brief Diretório dos dados nos pinos da porta PORTD (Reg. GPIOD_PDDR) */
#define GPIOD_PDDR   (*(uint32_t *) 0x400FF0D4)

/*! @brief Inverte o estado bit nos pinos da porta PORTB (Reg. GPIOB_PTOR) */
#define GPIOB_PTOR   ((uint32_t *) 0x400FF04C)
/*! @brief Inverte o estado bit nos pinos da porta PORTD (Reg. GPIOD_PTOR) */
#define GPIOD_PTOR   ((uint32_t *) 0x400FF0CC)

/*! @brief Seta o bit nos pinos da porta PORTB (Reg. GPIOB_PSOR) */
#define GPIOB_PSOR   ((uint32_t *) 0x400FF044)
/*! @brief Seta o bit nos pinos da porta PORTD (Reg. GPIOD_PSOR) */
#define GPIOD_PSOR   ((uint32_t *) 0x400FF0C4)

/*! @brief Limpa o bit nos pinos da porta PORTB (Reg. GPIOB_PCOR) */
#define GPIOB_PCOR   ((uint32_t *) 0x400FF048)
/*! @brief Limpa o bit nos pinos da porta PORTD (Reg. GPIOD_PCOR) */
#define GPIOD_PCOR   ((uint32_t *) 0x400FF0C8)

/*! @brief Define o bit correspondente ao pino do LED vermelho (bit 18 da porta)
#define LED_RED_PIN  (1<<18)
/*! @brief Define o bit correspondente ao pino do LED verde (bit 19 da porta)
#define LED_GREEN_PIN (1<<19)
/*! @brief Define o bit correspondente ao pino do LED azul (bit 1 da porta)
#define LED_BLUE_PIN (1<<1)
```

```

/*
 * @brief Estrutura que guarda as informações dos registradores que controlam
 */
typedef struct led {
    uint32_t *psor;      /*!< Registrador PSOR (seta o bit) */
    uint32_t *pcor;      /*!< Registrador PCOR (limpa o bit) */
    uint32_t *ptor;      /*!< Registrador PTOR (inverte o bit) */
    uint32_t pin;        /*!< Número do pino dentro do registrador */
} led_t;

/*
 * @brief Enumerações das cores possíveis de LED RGB
 */
typedef enum led_color {
    LED_RED = 0,
    LED_GREEN,
    LED_BLUE,
    MAX_LED_RGB
} led_color_t;

/*
 * @brief Configura e inicializa uma das cores do LED RGB (deve ser chamada uma vez)
 * @param [in] color Cor do LED a ser inicializada
 */
void led_init( led_color_t color );

/*
 * @brief Configura e inicializa apagada todas as cores do LED RGB (deve ser chamada uma vez)
 */
void led_init_rgb( void );

/*
 * @brief Acende uma das cores do LED RGB
 * @param [in] color Cor do LED a ser acesa
 */
void led_on( led_color_t color );

/*
 * @brief Apaga uma das cores do LED RGB
 * @param [in] color Cor do LED a ser apagada
 */
void led_off( led_color_t color );

/*
 * @brief Inverte o estado de uma das cores do LED RGB
 * @param [in] color Cor do LED a ser invertida
 */
void led_toggle( led_color_t color );

#endif

```

D Código Fonte - delay.c

```
/*
 * @file delay.c
 * @brief Função que faz delay por tempo
 *
 * @author Henrique Aires Silva
 * @date 22/03/2017
 *
 */
#include "delay.h"

void delay( uint32_t ticks )
{
    /*! Decrementa o número de iterações até zero */
    while ( ticks ) {
        ticks--;
    }
}
```

E Código Fonte - delay.h

```
/*!  
 * @file delay.h  
 * @brief Header das fun&ccedil;es de delay por la&ccedil;o  
 *  
 * @author Henrique Aires Silva  
 * @date 22/03/2017  
 *  
 */  
  
#ifndef DELAY_H_  
#define DELAY_H_  
  
#include <stdint.h>  
/*!  
 * @brief gera um atraso correspondente a i itera&ccedil;&otilde;es  
 * @param [in] ticks N&uacute;mero de itera&ccedil;&otilde;es  
 */  
void delay( uint32_t ticks );  
  
#endif
```