embarca tech | Residência Tecnológica em Sistemas Embarcados

Modulação por Largura de Pulso (PWM)

Unidade 4 | Capítulo 7

Otávio Alcântara de Lima Júnior

Executores:
INSTITUTO FEDERAL Piauí
INSTITUTO FEDERAL Rio Grande do Norte
INSTITUTO FEDERAL Maranhão
INSTITUTO FEDERAL Ceará
cepedi
INSTITUTO HARDWARE BR

Coordenação: Softex

Iniciativa: MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO
GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO

Sumário

1 Boas-Vindas e Apresentação

Olá, estudantes! Sejam bem-vindos ao material de apoio "Modulação por Largura de Pulso". O objetivo deste material é servir como apoio e conteúdo extra para o Capítulo 7 da Unidade 4 sobre Microcontroladores.

Os objetivos da nossa conversa de hoje são:

- Entender o processo de modulação por largura de pulso e seu funcionamento;
- Configurar a utilizar os diferentes modos de configuração do módulo PWM;
- Aplicar o módulo PWM para controle de intensidade de luminosa de LED.

2 Modulação por Largura de Pulso (PWM)

A Modulação por Largura de Pulso, mais conhecida pela sigla em inglês **PWM**, é uma das ferramentas essenciais no cinto de utilidades de um programador de sistemas embarcados. O PWM é uma técnica simples que permite controlar a potência entregue a um dispositivo eletrônico. Trata-se de um sinal digital de frequência constante, no qual variamos a porção do tempo que o sinal permanece em nível lógico alto. Esse tempo em nível alto é chamado de **ciclo ativo** (ou *duty cycle* em inglês).

Na Figura 1, mostramos um exemplo simples de como o PWM pode ser usado para controlar a intensidade de luminosidade de uma lâmpada ou LED. Observe que, ao aumentar o duty cycle, a quantidade de energia que a lâmpada recebe também aumenta, o que resulta em maior luminosidade. Dessa forma, o PWM não só controla a luminosidade, mas também pode influenciar o consumo de energia do sistema. Além de controlar LEDs, o PWM tem diversas outras aplicações, como: geração de sinais de áudio, controle de velocidade de motores, e regulação de conversores de potência, entre outras.

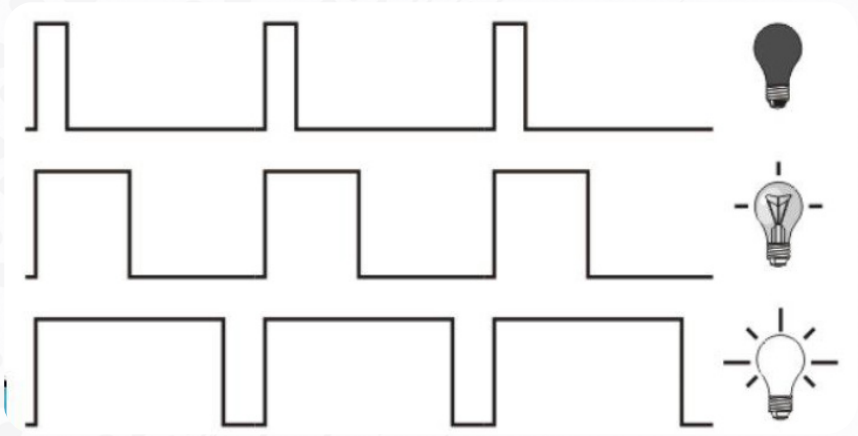

Figura 1: exemplo de Aplicação de PWM
Fonte: imagem do autor

A imagem mostra três ondas quadradas, uma sobre a outra. Cada onda representa um sinal elétrico.
A primeira onda, no topo, é um pulso único, alto e estreito. Ao lado dela, há um desenho de uma lâmpada apagada.
A segunda onda tem pulsos mais curtos e largos, em um padrão regular. Ao lado dela, há um desenho de uma lâmpada acesa, com um filamento fino.
A terceira onda tem pulsos ainda mais curtos e largos, em um padrão regular. Ao lado dela, há um desenho de uma lâmpada acesa, com um filamento mais grosso e mais brilhante, emitindo raios de luz.
As ondas e as lâmpadas sugerem uma relação entre a forma do sinal elétrico e a intensidade da luz da lâmpada. A primeira onda, com um pulso único, representa uma lâmpada desligada. A segunda onda, com pulsos mais curtos e largos, representa uma lâmpada acesa com pouca intensidade. A terceira onda, com pulsos ainda mais curtos e largos, representa uma lâmpada acesa com mais intensidade.

Graças à sua simplicidade, a maioria dos microcontroladores do mercado possui circuitos dedicados à geração de PWM. Por isso, é fundamental entendermos como essa técnica funciona e como podemos aplicá-la em nossos projetos. O PWM oferece um meio eficiente e preciso de controle em sistemas eletrônicos.

Neste material de apoio, exploraremos como o módulo PWM é implementado no microcontrolador **RP2040**, bem como os modos de configuração disponíveis. A seguir, apresentaremos exemplos detalhados de códigos, que ajudarão a entender como utilizar as funções do SDK [1] do Raspberry Pi Pico para implementar o PWM em seus projetos.

• IMPORTANTE

Por fim, não se esqueçam de revisar atentamente todo o conteúdo deste capítulo e de realizar as atividades propostas na plataforma. Esta é uma oportunidade para entender como os microcontroladores podem ser usados para transformar nosso cotidiano e para aplicar esse conhecimento em projetos reais. Contamos com sua participação ativa nessa jornada de aprendizado.

Vamos em frente!

3 Módulo PWM do RP2040

O **RP2040** é um microcontrolador moderno e versátil que oferece um conjunto de oito geradores de PWM, chamados de '**slices**'. Cada slice pode gerar dois sinais de PWM independentes, permitindo até 16 saídas PWM simultâneas [2]. Além de gerar sinais de PWM, esses slices podem ser usados para medir o período ou o ciclo ativo de sinais de entrada, aumentando a flexibilidade do sistema.

Uma outra vantagem do projeto do RP2040 é que **qualquer pino** pode ser configurado como uma saída PWM, facilitando o roteamento dos sinais conforme a necessidade do projeto. Com isso, o RP2040 oferece um total de **16 saídas PWM** operando simultaneamente, o que o torna uma excelente escolha para uma ampla gama de aplicações que demandam controle preciso de sinais.

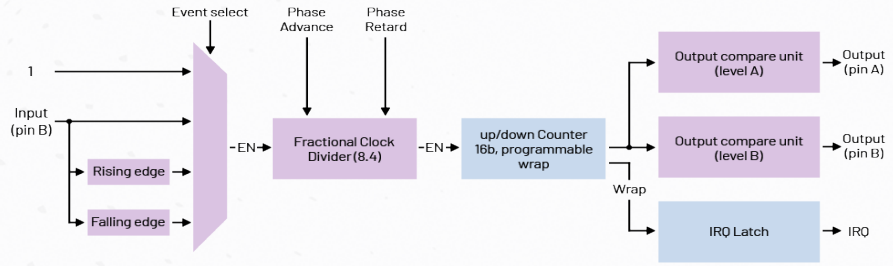

Figura 2: estrutura de um módulo PWM do RP2040 [2]
Fonte: Acesso direto a registradores com Micropython - Embarcados

A imagem mostra um diagrama de blocos que representa um sistema de contagem e comparação de tempo.
O diagrama possui sete blocos conectados por setas que indicam o fluxo de sinal. Os blocos são:
Evento select: Um bloco retangular com bordas trapezoidais, recebendo entradas de "1" e "Input (pin B)" e enviando um sinal para "Rising edge" e "Falling edge".
Rising edge: Um bloco retangular pequeno, que recebe sinal de "Evento select" e envia um sinal para "Fractional Clock Divider".
Falling edge: Um bloco retangular pequeno, que recebe sinal de "Evento select" e envia um sinal para "Fractional Clock Divider".
Fractional Clock Divider (8,4): Um bloco retangular, que recebe sinal de "Rising edge" e "Falling edge", além de um sinal de "EN" (enable) da direita. Este bloco envia um sinal para "up/down Counter".
up/down Counter 16b, programmable wrap: Um bloco retangular azul claro, que recebe sinal de "Fractional Clock Divider" e envia um sinal para "Output compare unit (level B)".
Output compare unit (level A): Um bloco retangular azul claro, que recebe sinal de "up/down Counter" e envia um sinal para "Output (pin A)".
Output compare unit (level B): Um bloco retangular azul claro, que recebe sinal de "up/down Counter" e envia um sinal para "Output (pin B)".
IRQ Latch: Um bloco retangular azul claro, que recebe sinal de "up/down Counter" por meio de "Wrap" e envia um sinal para "IRQ".
O diagrama indica que o sistema recebe um sinal de entrada (pin B) e, dependendo da configuração do evento select, o sinal será passado para o "Fractional Clock Divider". Este divide o sinal de entrada em uma frequência menor e o envia para o "up/down Counter", que incrementa ou decrementa o contador dependendo do sinal recebido. Os comparadores de saída (level A e level B) comparam o valor do contador com um valor de referência e geram um sinal de saída se houver uma correspondência. O sinal de saída é então armazenado no "IRQ Latch" e enviado para o "IRQ".
O diagrama mostra que o sistema é capaz de contar tempo e comparar o valor contado com um valor de referência, gerando um sinal de saída para uma interrupção (IRQ) se houver uma correspondência.
O texto "Phase Advance" e "Phase Retard" indica que o sistema também é capaz de ajustar a fase do sinal de entrada.
O diagrama não especifica o tipo de sinal de entrada, mas sugere que o sinal pode ser um sinal digital ou analógico.

Na **Figura 2**, apresentamos a estrutura de um módulo PWM do **RP2040**. Na parte esquerda do diagrama de blocos, há um **multiplexador** que permite especificar qual evento será utilizado para acionar o contador de 16 bits do PWM. Existem quatro opções: nível lógico de um pino, evento de borda de subida, evento de borda de descida, e o clock do sistema. As três primeiras opções são usadas no modo de medição de período ou ciclo ativo, enquanto a última opção é usada quando estamos gerando um sinal de PWM.

Após a seleção do evento que incrementa o contador, há um **divisor de frequência**, que ajusta quantos eventos de entrada são necessários para incrementar o contador de 16 bits. Esse contador pode operar em dois modos: **modo crescente** e **modo crescente/decrescente**. No modo crescente, o contador incrementa até atingir um limite, chamado de **wrap**, e então é reiniciado. Esse parâmetro define o período do sinal PWM. Na Figura 3, vemos que nesse modo de contagem, o registrador TOP é incrementado até um limite, em seguida é zerado, formando uma onda triangular. Na parte de baixo, vemos o sinal de PWM gerado, perceba que o duty cycle inicia sempre no começa da contagem.

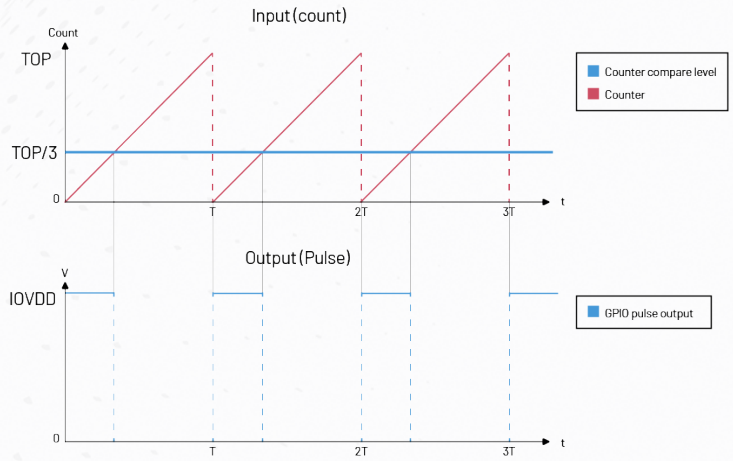

Figura 3: modo crescente do PWM [2]
Fonte: The Pico In C: Basic PWM

Este gráfico mostra o funcionamento de um contador.
O eixo vertical superior representa o valor de contagem, que vai de 0 até o valor máximo (TOP).
O eixo horizontal representa o tempo.
A linha vermelha representa o valor do contador.
A linha azul representa o nível de comparação do contador.
A linha azul no gráfico inferior representa a saída do pulso.
O gráfico mostra que o contador aumenta linearmente com o tempo e reinicia quando chega ao valor máximo (TOP).
A saída do pulso é gerada quando o valor do contador atinge o nível de comparação.
O gráfico mostra que a saída do pulso é gerada a cada terceiro ciclo do contador, pois o nível de comparação está definido em TOP/3.

No modo **crescente/decrescente**, o contador incrementa até o valor de **wrap** e, em seguida, decresce até zero, repetindo esse ciclo continuamente. A **Figura 4** ilustra esse comportamento. Note que, nesse modo, o **duty cycle** não começa mais a partir de zero, mas o seu centro permanece fixo, criando um sinal mais simétrico. Alguns dispositivos funcionam de maneira mais eficiente com esse tipo de PWM, pois ele oferece um controle mais equilibrado e suave em determinadas aplicações, como motores e sistemas de iluminação.

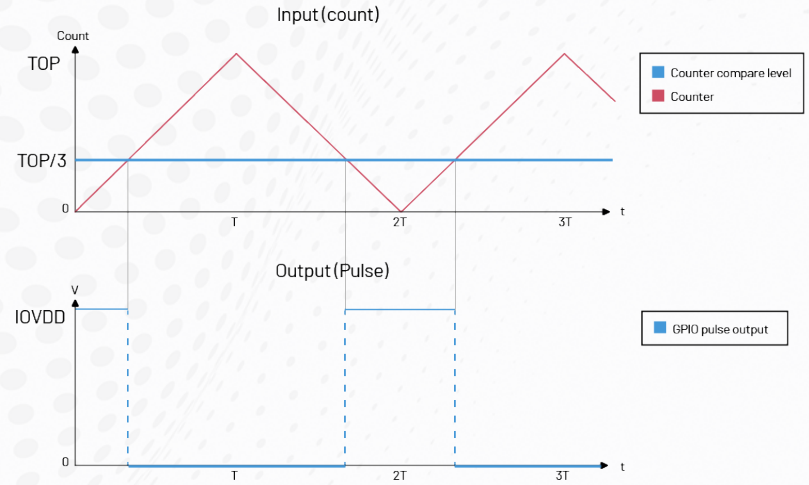

Figura 4: modo crescente/decrescente do PWM [2].
Fonte: The Pico In C: Basic PWM
Este gráfico mostra o funcionamento de um contador.
O eixo vertical superior representa o valor de contagem, que vai de 0 até o valor máximo (TOP).
O eixo horizontal representa o tempo.
A linha vermelha representa o valor do contador.
A linha azul representa o nível de comparação do contador.
A linha azul no gráfico inferior representa a saída do pulso.
O gráfico mostra que o contador aumenta linearmente com o tempo e reinicia quando chega ao valor máximo (TOP).
A saída do pulso é gerada quando o valor do contador atinge o nível de comparação.
O gráfico mostra que a saída do pulso é gerada a cada terceiro ciclo do contador, pois o nível de comparação está definido em TOP/3.

Os últimos três blocos do diagrama estão relacionados à geração do sinal nos canais **A** e **B**. Cada gerador de PWM possui dois canais que podem ser associados a pinos de saída do microcontrolador. O último bloco é responsável pela geração de **interrupções do PWM**, que podem ser usadas tanto para ajustar o valor do duty cycle quanto para obter medições dos sinais de entrada.
A Tabela 1 apresent
a o mapeamento dos pinos de GPIO que podem ser usados com os geradores de PWM do microcontrolador [2]. É importante notar que todos os trinta pinos podem ser usados como PWM.

GPIO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B	7A	7B
GPIO	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
PWM Channel	0A	0B	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	6A	6B		

Tabela 1: Mapeamento dos pinos de GPIO nos geradores de PWM [2]
Fonte: Imagem do autor.

É uma tabela que mostra a correspondência entre os pinos GPIO (General Purpose Input/Output) e os canais PWM (Pulse Width Modulation) de um dispositivo.
A tabela tem duas linhas e duas colunas, com os números dos pinos GPIO na primeira linha e os canais PWM correspondentes na segunda linha.
Os pinos GPIO variam de 0 a 29, enquanto os canais PWM variam de 0A a 7B.
Cada pino GPIO corresponde a um canal PWM específico.
Por exemplo, o pino GPIO 0 corresponde ao canal PWM 0A, o pino GPIO 1 corresponde ao canal PWM 0B, e assim por diante.
A tabela mostra que os pinos GPIO 0 a 15 e 16 a 29 são todos mapeados para canais PWM específicos.
Essa tabela pode ser útil para entender como os pinos GPIO podem ser usados para controlar os canais PWM.

4 Etapas de configuração do PWM usando o SDK

Vamos apresentar uma metodologia para configurar o **PWM** usando as

funções do **SDK do Raspberry Pi Pico** [1]. Essa metodologia é dividida em quatro etapas: configuração do GPIO, configuração do sinal de saída, habilitação da saída PWM e ajuste do duty cycle.

Na primeira etapa, precisamos configurar o pino que será usado como saída do PWM e identificar qual **slice** está associado a esse pino. Para isso, utilizaremos duas funções do SDK: **gpio_set_function** e **pwm_gpio_to_slice_num**. A primeira função recebe um inteiro correspondente ao número do pino. Por exemplo, para o pino **GP12**, usamos o valor 12. Além disso, essa função recebe um segundo parâmetro, que deve ser a constante **GPIO_FUNC_PWM**. Após a execução dessa função, o pino estará configurado para operar como uma saída PWM.

A segunda função, **pwm_gpio_to_slice_num**, recebe o número do pino e retorna o número do **slice** associado a esse pino, permitindo que você controle qual slice gerará o sinal PWM nesse pino.

A segunda etapa consiste na configuração do **período (wrap)**, do **divisor de clock** e do **duty cycle (level)**. Para isso, é importante compreender a relação entre esses parâmetros e a frequência do clock do sistema. A **Equação 1** apresenta essa relação.

Observe que o clock do sistema, por padrão, opera a 125 MHz. Dependendo da aplicação, será necessário definir a frequência de operação do PWM, que é determinada pelo divisor de clock (composto por uma parte inteira e uma parte fracionária) e pelo valor do período (wrap). Ajustando esses parâmetros, podemos controlar tanto a frequência quanto a precisão do sinal PWM gerado.

É importante ter cuidado ao ajustar esses valores. Um valor muito pequeno de wrap pode resultar em uma perda de resolução no PWM, o que levará a um controle menos preciso do sistema.

$$f_{PWM} = \frac{f_{clock}}{\left(\text{divisor inteiro} + \frac{\text{divisor fracional}}{16}\right).\text{wrap}}$$

Onde:
f_{PWM} *é a frequência do sinal de PWM.*
f_{clock} *é a frequência do sinal de clock base (125 Mhz).*
divisor inteiro é a parte inteiro do dividor de clock, possui 8 bits.
divisor fracionário é a parte fracionária do dividor de clock, possui 4 bits.
wrap é o valor máximo de contagem, possui 16 bits.

Figura 5: Equação 1- cálculo da frequência de operação do PWM [2]
Fonte: Imagem do autor.

A imagem mostra a fórmula para calcular a frequência de um sinal PWM.
A fórmula é: f_PWM = f_clock / (divisor_inteiro + divisor_fracional/16) * wrap.
Onde:
f_PWM é a frequência do sinal PWM.
f_clock é a frequência do sinal de clock base, que é 125 MHz.
divisor_inteiro é a parte inteira do divisor de clock, que possui 8 bits.
divisor_fracional é a parte fracionária do divisor de clock, que possui 4 bits.
wrap é o valor máximo de contagem, que possui 16 bits.

Para realizar a etapa 2, utilizamos as funções **pwm_set_clkdiv**, **pwm_set_wrap** e **pwm_set_gpio_level**. A primeira função, **pwm_set_clkdiv**, recebe uma variável de ponto flutuante que define o divisor do clock de entrada. A segunda função, **pwm_set_wrap**, recebe um valor inteiro de 16 bits, que representa o valor máximo de contagem do PWM (o valor de wrap). Por fim, a função **pwm_set_gpio_level** recebe o número do pino do PWM e o valor de **level**, que corresponde ao **duty cycle**.

Para realizar a **terceira etapa**, utilizamos a função **pwm_set_enabled**, passando como parâmetros o número do slice e o valor **1** para habilitar ou 0 para desabilitar o PWM. Por fim, na quarta etapa, fazemos o ajuste do **duty cycle** de forma periódica utilizando a função **pwm_set_gpio_level**, que já foi explicada anteriormente.

• NA PRÁTICA

Exemplo de Código 01

Nesta seção, apresentamos com usar o módulo PWM do RP2040 para controlar o brilho do LED da placa BitDogLab [3]. Para tal, você precisa criar um novo projeto no VS Code usando o plugin do Raspberry Pi Pico, como você fez nas aulas anteriores. Após criar o projeto, você precisará editar o conteúdo do arquivo CMakeLists.txt, que controla o build system do projeto. Você irá adicionar a biblioteca hardware_pwm. A Figura 5 ilustra essa modificação.

```
# Add the standard library to the build
target_link_libraries(PWM_LED_0
        pico_stdlib hardware_pwm)
```

Figura 6: adição da biblioteca hardware_pwm no CMakeLists.txt.
Fonte: Imagem do autor

A imagem mostra um código de programação em uma tela escura. O código inclui uma linha que adiciona bibliotecas padrão ao processo de compilação. A linha de código é: target_link_libraries(PWM_LED_0 pico_stdlib hardware_pwm).
A linha de código usa a função target_link_libraries() para adicionar as bibliotecas pico_stdlib e hardware_pwm ao alvo de compilação chamado PWM_LED_0.
O código é destacado em cores, com comentários em verde, nomes de funções e bibliotecas em azul e palavras-chave em amarelo.

Vamos agora apresentar o código da nossa aplicação, destacando os trechos principais. A **Figura 6** mostra o início do nosso programa. Podemos ver a inclusão das bibliotecas necessárias, bem como a definição das constantes que serão utilizadas ao longo do código, como o **endereço do LED**, o **valor do período do PWM**, o **divisor de clock** e o **passo de incremento do duty cycle**. Além disso, declaramos uma variável chamada **led_level**, que será responsável por armazenar o **duty cycle** do PWM.

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/pwm.h"

const uint LED = 12; // Pino do LED conectado
const uint16_t PERIOD = 2000; // Período do PWM (valor máximo do contador)
const float DIVIDER_PWM = 16.0; // Divisor fracional do clock para o PWM
const uint16_t LED_STEP = 100; // Passo de incremento/decremento para o duty cycle do LED
uint16_t led_level = 100; // Nível inicial do PWM (duty cycle)
```

Figura 7: inclusão de bibliotecas e declaração de constantes
Fonte: Imagem do autor

A imagem mostra um trecho de código em C, escrito em um editor de código com fundo escuro e cores que destacam os elementos do código.
O código define algumas variáveis constantes com seus respectivos valores e comentários explicativos.
As primeiras três linhas do código incluem arquivos de cabeçalho: stdio.h, pico/stdlib.h e hardware/pwm.h.
Em seguida, são definidas cinco variáveis constantes:
LED: um inteiro que representa o pino do LED conectado, com valor 12.
PERIOD: um inteiro de 16 bits que define o período do PWM (valor máximo do contador), com valor 2000.
DIVIDER_PWM: um número de ponto flutuante que representa o divisor fracional do clock para o PWM, com valor 16.0.
LED_STEP: um inteiro de 16 bits que define o passo de incremento/decremento para o duty cycle do LED, com valor 100.
led_level: um inteiro de 16 bits que representa o nível inicial do PWM (duty cycle), com valor 100.
Os comentários em português explicam a finalidade de cada variável.

Para simplificar a configuração do PWM, criamos uma função chamada **setup_pwm**, que segue a metodologia explicada neste documento. A **Figura 7** apresenta o código da função **setup_pwm**. Inicialmente, declaramos uma variável para armazenar o número do slice do PWM. Em seguida, configuramos o pino do LED como saída PWM utilizando a função **gpio_set_function**. Depois, identificamos

a qual slice o pino está associado, chamando a função **pwm_gpio_to_slice_num**. A configuração do divisor de clock, do wrap e do level (duty cycle) é feita por meio das funções **pwm_set_clkdiv**, **pwm_set_wrap** e **pwm_set_gpio_level**, respectivamente. Por fim, habilitamos a saída do PWM com a função **pwm_set_enabled**.

```
void setup_pwm()
{
    uint slice;
    gpio_set_function(LED, GPIO_FUNC_PWM); // Configura o pino do LED para função PWM
    slice = pwm_gpio_to_slice_num(LED); // Obtém o slice do PWM associado ao pino do LED
    pwm_set_clkdiv(slice, DIVIDER_PWM); // Define o divisor de clock do PWM
    pwm_set_wrap(slice, PERIOD); // Configura o valor máximo do contador (período do PWM)
    pwm_set_gpio_level(LED, led_level); // Define o nível inicial do PWM para o pino do LED
    pwm_set_enabled(slice, true); // Habilita o PWM no slice correspondente
}
```

Figura 8: Função setup_pwm
Fonte: Imagem do autor

A imagem mostra um trecho de código em C, escrito em um editor de código com fundo escuro e cores que destacam os elementos do código. O código define uma função chamada setup_pwm(), que configura o PWM (Pulse Width Modulation) para controlar um LED.
A função é composta de seis linhas:
1 Declara uma variável uint slice para armazenar o número do slice do PWM.
2 Chama a função gpio_set_function() para configurar o pino do LED para a função PWM.
3 Chama a função pwm_gpio_to_slice_num() para obter o número do slice do PWM associado ao pino do LED.
4 Chama a função pwm_set_clkdiv() para definir o divisor de clock do PWM.
5 Chama a função pwm_set_wrap() para configurar o valor máximo do contador (período do PWM).
6 Chama as funções pwm_set_gpio_level() e pwm_set_enabled() para definir o nível inicial do PWM e habilitar o PWM no slice correspondente.
Os comentários em português explicam a função de cada linha de código.

A função **main** será apresentada nas **Figuras 8** e **9**, devido à sua extensão. Na **Figura 8**, vemos a declaração de uma variável **up_down**, que controla se a luminosidade do LED aumenta ou diminui. Em seguida, chamamos a função **stdio_init_all** para inicializar o sistema padrão de entrada e saída. Após isso, chamamos nossa função **setup_pwm**.

Após as configurações iniciais, entramos no **superloop**, onde ajustamos o **duty cycle** utilizando a função **pwm_set_gpio_level**. Em seguida, aplicamos a lógica que controla a intensidade do LED. Se **up_down** for verdadeira, a variável **led_level** é incrementada até atingir seu valor máximo, com o incremento acontecendo a cada iteração do laço. Se **up_down** for falsa, a variável **led_level** é decrementada até atingir seu valor mínimo, como mostrado na **Figura 9**. Ao atingir um dos limites (máximo ou mínimo), a variável **up_down** é alternada, iniciando o próximo ciclo. Dessa forma, o LED aumenta sua intensidade até o máximo, depois diminui até o mínimo, repetindo o ciclo.

```c
int main()
{
    uint up_down = 1; // Variável para controlar se o nível do LED aumenta ou diminui
    stdio_init_all(); // Inicializa o sistema padrão de I/O
    setup_pwm(); // Configura o PWM
    while (true)
    {
        pwm_set_gpio_level(LED, led_level); // Define o nível atual do PWM (duty cycle)
        sleep_ms(1000); // Atraso de 1 segundo
        if (up_down)
        {
            led_level += LED_STEP; // Incrementa o nível do LED
            if (led_level >= PERIOD)
                up_down = 0; // Muda direção para diminuir quando atingir o período máximo
        }
        else
```

Figura 9: Trecho inicial da função main
Fonte: Imagem do autor.

A imagem mostra um trecho de código em C, escrito em um editor de código com fundo escuro e cores que destacam os elementos do código.
O código define a função main(), que é o ponto de entrada do programa.
A função main() é composta de sete linhas:
1 Define a variável uint up_down como 1, que controla se o nível do LED aumenta ou diminui.
2 Chama a função stdio_init_all() para inicializar o sistema padrão de I/O.
3 Chama a função setup_pwm() para configurar o PWM.
4 Inicia um loop infinito com while(true).
5 Dentro do loop, chama a função pwm_set_gpio_level() para definir o nível atual do PWM (duty cycle).
6 Utiliza a função sleep_ms() para atrasar o programa por 1 segundo.
7 Verifica se a variável up_down é verdadeira (1), e se for, incrementa o nível do LED e verifica se ele atingiu o período máximo, mudando a direção para diminuir caso positivo.
Caso contrário, decrementa o nível do LED e verifica se atingiu o valor mínimo, mudando a direção para aumentar caso positivo.
Os comentários em português explicam a função de cada linha de código.

Os ajustes são feitos a cada 1 segundo, graças à função **sleep_ms(1000)**.

Utilize sua **BitDogLab** para replicar este exemplo em casa. Em caso de dúvidas, entre em contato com nossa equipe pela plataforma de ensino.

```c
while (true)
{
    pwm_set_gpio_level(LED, led_level); // Define o nível atual do PWM (duty cycle)
    sleep_ms(1000); // Atraso de 1 segundo
    if (up_down)
    {
        led_level += LED_STEP; // Incrementa o nível do LED
        if (led_level >= PERIOD)
            up_down = 0; // Muda direção para diminuir quando atingir o período máximo
    }
    else
    {
        led_level -= LED_STEP; // Decrementa o nível do LED
        if (led_level <= LED_STEP)
            up_down = 1; // Muda direção para aumentar quando atingir o mínimo
    }
}
```

Figura 10: super loop da função main
Fonte: Imagem do autor.

A imagem mostra um trecho de código em C, escrito em um editor de código com fundo escuro e cores que destacam os elementos do código. O código implementa um loop infinito que controla o brilho de um LED conectado a um microcontrolador. O loop é responsável por aumentar e diminuir o brilho do LED de forma cíclica, criando um efeito de fade.
O código funciona da seguinte maneira:
1 while(true): Inicia um loop infinito, que será executado continuamente.
2 pwm_set_gpio_level(LED, led_level);: Define o nível atual do PWM (Pulse Width Modulation) para o pino do LED, controlando o brilho do LED.
3 sleep_ms(1000);: Aguarda por 1 segundo.
4 if (up_down): Se a variável up_down for verdadeira (1), significa que o LED está aumentando de brilho.
5 led_level += LED_STEP;: Incrementa o nível do LED em LED_STEP.
6 if (led_level >= PERIOD): Se o nível do LED atingiu o valor máximo (PERIOD), muda a direção do fade, definindo up_down como 0.
7 else: Caso contrário, se up_down for falsa (0), significa que o LED está diminuindo de brilho.
8 led_level -= LED_STEP;: Decrementa o nível do LED em LED_STEP.
9 if (led_level <= LED_STEP): Se o nível do LED atingiu o valor mínimo (LED_STEP), muda a direção do fade, definindo up_down como 1.
Os comentários no código explicam cada etapa do código em português, facilitando a compreensão do seu funcionamento.

Exemplo de Código 02

Nesta seção, vamos apresentar uma variação do exemplo de código anterior, agora utilizando interrupções para atualizar o *duty cycle*. O primeiro passo será criar um novo projeto no VS Code e editar o arquivo `CMakeLists.txt` para incluir a biblioteca do PWM. Após realizar esses ajustes, explicaremos o código do programa em detalhes.

Iniciamos o código com a inclusão das bibliotecas padrão e a definição das constantes e variáveis que serão usadas, como ilustrado na Figura 10. As configurações são semelhantes às do exemplo anterior, com uma adição: a constante `PWM_REFRESH_LEVEL`. Ela será utilizada na rotina de tratamento de interrupções, indicando quando será necessário atualizar o *duty cycle*.

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/pwm.h"

// Definições de constantes para o controle do PWM
const uint LED = 12;            // Pino do LED conectado
const uint16_t PERIOD = 2000;   // Período do PWM (valor máximo do contador)
const float DIVIDER_PWM = 16.0; // Divisor fracional do clock para o PWM
const uint16_t LED_STEP = 100;  // Passo de incremento/decremento para o duty cycle do LED
const uint32_t PWM_REFRESH_LEVEL = 40000; // Nível de contagem para suavizar o fade do LED
uint16_t led_level = 100;       // Nível inicial do PWM (duty cycle do LED)
```

Figura 11: Inclusão de bibliotecas e declaração de constantes e variáveis.
A imagem mostra um trecho de código em C, escrito em um editor de código com fundo escuro e cores que destacam os elementos do código.

O código inclui três linhas de código que adicionam arquivos de cabeçalho para usar funções de entrada e saída padrão (stdio.h), bibliotecas do sistema para o microcontrolador (pico/stdlib.h) e funções específicas de controle PWM (hardware/pwm.h).
Em seguida, o código define seis variáveis constantes com seus respectivos valores e comentários explicativos.
LED: um inteiro que representa o pino do LED conectado, com valor 12.
PERIOD: um inteiro de 16 bits que define o período do PWM (valor máximo do contador), com valor 2000.
DIVIDER_PWM: um número de ponto flutuante que representa o divisor fracional do clock para o PWM, com valor 16.0.
LED_STEP: um inteiro de 16 bits que define o passo de incremento/decremento para o duty cycle do LED, com valor 100.
PWM_REFRESH_LEVEL: um inteiro de 32 bits que define o nível de contagem para suavizar o fade do LED, com valor 40000.
led_level: um inteiro de 16 bits que representa o nível inicial do PWM (duty cycle do LED), com valor 100.
Os comentários em português explicam a finalidade de cada variável.

Agora, vamos analisar as alterações feitas na função `setup_pwm`, mostradas na Figura 11. O início da rotina permanece praticamente o mesmo, mas realizamos modificações importantes para configurar a interrupção do PWM.

Utilizamos a função `irq_set_exclusive_handler` para definir qual função será responsável pelo tratamento da interrupção do PWM — no nosso caso, a função `pwm_irq_handler`, que será apresentada em breve. Em seguida, limpamos a flag de interrupção usando `pwm_clear_irq`, o que é uma boa prática de projeto para garantir que a interrupção anterior foi resolvida corretamente. Logo após, habilitamos as interrupções para o canal do PWM com a chamada à função `pwm_set_irq_enabled`. Finalmente, ativamos as interrupções globais para esse periférico.

```c
// Função para configurar o PWM para o LED
void setup_pwm()
{
    uint slice;

    gpio_set_function(LED, GPIO_FUNC_PWM); // Configura o pino do LED para função PWM
    slice = pwm_gpio_to_slice_num(LED);    // Obtém o slice do PWM associado ao pino do
    pwm_set_clkdiv(slice, DIVIDER_PWM);    // Define o divisor de clock do PWM
    pwm_set_wrap(slice, PERIOD);           // Configura o valor máximo do contador (perí
    pwm_set_gpio_level(LED, led_level);    // Define o nível inicial do PWM para o pino
    pwm_set_enabled(slice, true);          // Habilita o PWM no slice correspondente

    // Configura e habilita a interrupção do PWM
    irq_set_exclusive_handler(PWM_IRQ_WRAP, pwm_irq_handler); // Define o handler da int
    pwm_clear_irq(slice);                 // Limpa interrupções pendentes do slice
    pwm_set_irq_enabled(slice, true);     // Habilita interrupções para o slice do PWM
    irq_set_enabled(PWM_IRQ_WRAP, true); // Habilita interrupções globais para o PWM
}
```

Figura 12: função setup_pwm.

A imagem mostra um trecho de código em C, escrito em um editor de código com fundo escuro e cores que destacam os elementos do código.
O código define uma função chamada setup_pwm(), que configura o PWM (Pulse Width Modulation) para controlar um LED.
A função é composta de nove linhas:
1 Declara uma variável uint slice para armazenar o número do slice do PWM.
2 Chama a função gpio_set_function() para configurar o pino do LED para a função PWM.
3 Chama a função pwm_gpio_to_slice_num() para obter o número do slice do PWM associado ao pino do LED.
4 Chama a função pwm_set_clkdiv() para definir o divisor de clock do PWM.
5 Chama a função pwm_set_wrap() para configurar o valor máximo do contador (período do PWM).
6 Chama a função pwm_set_gpio_level() para definir o nível inicial do PWM para o pino do LED.
7 Chama a função pwm_set_enabled() para habilitar o PWM no slice correspondente.
8 Chama a função irq_set_exclusive_handler() para definir o handler da interrupção do PWM e habilitá-lo.
9 Chama as funções pwm_clear_irq(), pwm_set_irq_enabled() e irq_set_enabled() para limpar as interrupções pendentes do slice, habilitar as interrupções para o slice do PWM e habilitar as interrupções globais para o PWM.
Os comentários em português explicam a função de cada linha de código.

A rotina `pwm_irq_handler` será chamada pelo SDK sempre que ocorrer a interrupção `PWM_IRQ_WRAP`, ou seja, quando o valor do contador do PWM atingir o valor máximo definido por wrap. A Figura 12 apresenta o código da função que trata essa interrupção.

No início da função, criamos duas variáveis estáticas: `up_down` e `count`. A variável `up_down` já era utilizada no exemplo anterior, porém sua lógica estava dentro da função `main`. Neste exemplo, transferimos a lógica de ajuste do *duty cycle* para a rotina de interrupção do PWM. A variável `count` é utilizada para contabilizar o número de interrupções ocorridas, sendo que o ajuste do PWM só será realizado após o número de interrupções definido por `PWM_REFRESH_LEVEL`.

Observe que chamamos a função `pwm_clear_irq` para limpar a *flag* de interrupção dentro da rotina de tratamento, garantindo que a interrupção foi devidamente processada. O código subsequente implementa a lógica de atualização do *duty cycle*, semelhante à utilizada no exemplo anterior.

```c
void pwm_irq_handler()
{
    static uint up_down = 1;    // Variável que controla se o LED está aumentando ou dimin
    static uint32_t count = 0;  // Contador para controlar a frequência de atualização do
    uint32_t slice = pwm_get_irq_status_mask(); // Obtém o status da interrupção do PWM
    pwm_clear_irq(slice); // Limpa a interrupção do slice correspondente
    if (count++ < PWM_REFRESH_LEVEL) return; // Verifica se o contador atingiu o nível de
    count = 0; // Reseta o contador para iniciar a próxima verificação
    if (up_down){// Ajusta o duty cycle do LED, alternando entre aumentar e diminuir
        led_level += LED_STEP; // Incrementa o nível do LED
        if (led_level >= PERIOD)
            up_down = 0; // Muda a direção para diminuir o brilho quando atinge o valor má
    }else{
        led_level -= LED_STEP; // Decrementa o nível do LED
        if (led_level <= LED_STEP)
            up_down = 1; // Muda a direção para aumentar o brilho quando atinge o valor mí
    }
    pwm_set_gpio_level(LED, led_level); // Atualiza o duty cycle do PWM para o pino do LED
}
```

Figura 13: rotina de tratamento de interrupção do PWM.

A imagem mostra o código de uma função em linguagem C chamada pwm_irq_handler, que é um handler para interrupções do PWM (Pulse Width Modulation). A função é responsável por atualizar o duty cycle do PWM, que controla o brilho de um LED conectado ao microcontrolador.
A função pwm_irq_handler é composta por várias linhas de código que executam as seguintes tarefas:
1 Define três variáveis:
static uint up_down = 1;: Uma variável que controla se o LED está aumentando ou diminuindo de brilho. Inicialmente, ela está definida como 1 (aumentando).
static uint32_t count = 0;: Um contador para controlar a frequência de atualização do duty cycle.
uint32_t slice = pwm_get_irq_status_mask();: Obtém o status da interrupção do PWM correspondente.
2 Chama a função pwm_clear_irq(slice); para limpar a interrupção do slice correspondente.
3 Incrementa o contador count e verifica se ele atingiu o nível de atualização definido pela constante PWM_REFRESH_LEVEL. Se sim, a função retorna.
4 Se o contador não atingiu o nível de atualização, ele é redefinido para 0 e o código verifica se a variável up_down é igual a 1.
5 Se up_down é igual a 1, o código aumenta o nível do LED (led_level) em LED_STEP e verifica se ele atingiu o valor máximo definido pela constante PERIOD. Se sim, a direção do brilho é invertida (up_down é definida como 0) para começar a diminuir o brilho.
6 Caso contrário, se up_down é igual a 0, o código diminui o nível do LED em LED_STEP e verifica se ele atingiu o valor mínimo definido pela constante LED_STEP. Se sim, a direção do brilho é invertida (up_down é definida como 1) para começar a aumentar o brilho.
7 Por fim, a função chama pwm_set_gpio_level(LED, led_level); para atualizar o duty cycle do PWM para o pino do LED, ajustando o brilho do LED de acordo com as alterações no nível.
Os comentários no código explicam cada etapa, facilitando a compreensão do seu funcionamento.

Por fim, temos a função main, apresentada na Figura 13, que é bem mais simples do que no primeiro exemplo. Apenas fazemos a configuração do microcontrolador e deixamos o super loop vazio, visto que a lógica do programa é tratada na interrupção.

```c
// Função principal do programa
int main()
{
    stdio_init_all(); // Inicializa o sistema padrão de I/O (UART, etc.)
    setup_pwm();      // Configura o PWM para o LED

    // Loop infinito - o controle do PWM é feito pela interrupção
    while (true)
    {
        // O loop principal pode ser usado para outras tarefas, se necessário
    }
}
```

Figura 14: função main.

A imagem mostra o código da função main em linguagem C, que é o ponto de entrada do programa. O código é escrito em um editor de código com fundo escuro e cores que destacam os elementos do código.
O código da função main é composto de quatro linhas principais:
1 stdio_init_all();: Essa linha inicializa o sistema padrão de entrada e saída (I/O), incluindo o UART (Universal Asynchronous Receiver/Transmitter) e outros recursos de comunicação.
2 setup_pwm();: Essa linha chama a função setup_pwm, que configura o PWM (Pulse Width Modulation) para controlar o LED.
3 while(true): Essa linha inicia um loop infinito, que continuará executando até que o programa seja interrompido. O loop é usado para manter o programa em execução e gerenciar a comunicação com o hardware.
4 // O loop principal pode ser usado para outras tarefas, se necessário: Essa linha é um comentário, indicando que o loop principal pode ser usado para outras tarefas, se necessário.
Os comentários no código em português explicam a função de cada linha, facilitando a compreensão do código.

5 CONCLUSÃO

O **PWM** é uma ferramenta extremamente útil, permitindo o controle da potência entregue a dispositivos eletrônicos, como o brilho de LEDs, a velocidade de motores e o funcionamento de conversores de potência, entre outras aplicações. O **RP2040** possui um módulo PWM completo, com suporte para até 16 sinais de saída, que podem ser conectados a qualquer um dos 30 pinos disponíveis. Além disso, o SDK da Raspberry Pi oferece funções simples e fáceis de utilizar para operar esse recurso de forma eficiente.

• ATENÇÃO

Aproveite esta oportunidade para replicar o exemplo de código apresentado na aula em sua placa. Espero que este material de apoio seja útil e enriquecedor para o seu aprendizado. Não deixe de aprofundar seus estudos consultando as referências deste texto, que servem como material suplementar. Vamos explorar juntos o fascinante mundo dos microcontroladores!

REFERÊNCIAS

[1] RASPBERRY PI LTD. Raspberry Pi Pico C/C++ SDK. 2024. Disponível em: https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf. Acesso em: 03 set. 2024.

[2] RASPBERRY PI LTD. RP2040 Datasheet. 2024. Disponível em: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf. Acesso em: 03 set. 2024.

[3] BitDogLab. Manual BitDogLab, Disponível em: https://github.com/BitDogLab/BitDogLab/tree/main/doc. Acesso em: 03 set. 2024.

7. RECURSOS

Exemplo de código 01:
https://www.dropbox.com/scl/fo/y9n6u0ssffyvjgzd3wxx9/AHsvflwN_N6kjWD_qiefH4E?rlkey=khv769gbbic7yk0puesgeosy4&st=eg1gkbrx&dl=0

Exemplo de código 02:
https://www.dropbox.com/scl/fo/oh96p5bu1p9m46m9bz4fw/ALXvZ2gf5FqgSVnfc0SZCW4?rlkey=sfu38wy0gf17etlrp9r7oj9wz&st=rbnsk8sk&dl=0


embarca
tech
Residência
Tecnológica
em Sistemas
Embarcados