POLI-USP

PCS3732 - Documentação do projeto

SpotiPi

Autores
Bárbara Bueno
Carlos Engler
Erick Sousa

Implementação bare-metal de áudio PWM inspirada em consoles 8-bit.

2025

Conteúdo

1 Introdução

1.1 Motivação e objetivos do projeto

A popularização de computadores de baixo custo como o Raspberry Pi abriu novas possibilidades para a experimentação de conceitos de eletrônica, sistemas embarcados e processamento de sinais. Nosso projeto, denominado *SpotiPi*, tem como objetivo implementar a geração de áudio diretamente a partir dos periféricos de hardware do Raspberry Pi, sem depender de sistemas operacionais ou bibliotecas de alto nível. A ideia central é explorar o módulo de PWM (*Pulse Width Modulation*) integrado, operando em modo *bare-metal*, para produzir sons e músicas de forma semelhante aos videogames clássicos da era 8-bit.

Entre os objetivos específicos, destacam-se:

- Estudar o funcionamento do periférico PWM do Raspberry Pi e seus registradores de controle.
- Implementar funções em C para inicializar, configurar e manipular os canais de PWM.
- Reproduzir sons musicais por meio de um buzzer passivo, aproveitando o controle preciso de frequência e *duty cycle*.

1.2 Contexto histórico: drivers de áudio em videogames clássicos

Nos videogames clássicos da década de 1980, como o Nintendo Entertainment System (NES) e o Atari 2600, a geração de áudio era realizada por meio de circuitos dedicados, capazes de produzir formas de onda simples e controladas por registradores. O NES, por exemplo, dispunha de:

- Dois canais de onda quadrada (*square wave*) com controle de frequência e volume.
- Um canal de onda triangular (*triangle wave*) usado principalmente para baixos e efeitos suaves.
- Um canal de ruído (*noise*) para percussão e efeitos sonoros.

Apesar de suas limitações, esses sistemas permitiam composições musicais ricas e icônicas, explorando ao máximo os poucos canais disponíveis.

Neste projeto, propomos utilizar os dois canais independentes de PWM do Raspberry Pi como os canais de onda quadrada do NES. Ao conectar cada canal a um

buzzer passivo, podemos produzir sons com frequências distintas simultaneamente, simulando a polifonia básica dos consoles 8-bit. Essa abordagem mantém a essência do design original desses sistemas, mas com a vantagem de aproveitar a flexibilidade de um microcomputador moderno em modo *bare-metal*.

1.3 Desafios na documentação e extensão da biblioteca *libpi*

Ao longo do desenvolvimento, constatamos que a documentação oficial sobre os periféricos PWM do Raspberry Pi é bastante limitada e fragmentada, o que torna o entendimento e a aplicação prática do módulo um desafio. Além disso, não existem bibliotecas consolidadas que ofereçam controle bare-metal do PWM com a flexibilidade necessária para nossas finalidades.

Diante disso, decidimos estender a biblioteca libpi, originalmente desenvolvida pelo professor Bruno Bassetto para manipulação de GPIOs (além de outras funcionalidades básicas, como UART), adicionando funcionalidades específicas para o controle do PWM. Esse trabalho envolveu uma extensa investigação, testes práticos e pesquisa em múltiplas fontes (incluindo documentação técnica oficial, fóruns especializados e experimentação direta) para construir uma documentação própria, confiável e que atenda às necessidades do projeto.

Essa abordagem permitiu não só o desenvolvimento do firmware para geração de áudio, mas também criou uma base reutilizável para futuros projetos que precisem de controle bare-metal dos periféricos do Raspberry Pi, tendo em vista a falta desse tipo de conteúdo na internet, onde predominam bibliotecas de alto nível (como WiringPi e pigpio), que não entram em detalhes de implementação

Referências para áudio 8-bit e chiptune:

- How to write 8 bit chip tune music (Rescoring Super Mario Bros)
- The Basics of Chiptune Music

2 Fundamentos de PWM

2.1 O que é PWM (Pulse Width Modulation)

O PWM (*Pulse Width Modulation*) é uma técnica que controla o tempo em que um sinal digital permanece em nível alto dentro de um período fixo, chamado *período* T_{total}. A fração desse tempo em nível alto é chamada *duty cycle DC*, e é dada por:

$$DC = \frac{T_{\text{high}}}{T_{\text{total}}}$$

onde:

- T_{high}: tempo em que o sinal está em nível alto;
- T_{total}: período total do sinal.

O sinal resultante é uma onda quadrada cuja largura do pulso alto varia para controlar a potência média entregue ao dispositivo.

2.2 PWM padrão do Raspberry Pi: pulsos espaçados

No Raspberry Pi, o modo PWM padrão (default) não gera um único pulso contínuo em nível alto no início do período. Em vez disso, o duty cycle é distribuído em múltiplos pulsos curtos, uniformemente espaçados ao longo de todo o período.

Suponha que o período seja dividido em N intervalos iguais e que o duty cycle seja definido como $D = \frac{M}{N}$, com $M \leq N$. O sinal PWM resultante apresenta M pulsos curtos em nível alto, intercalados com pulsos em nível baixo, espaçados ao longo do período, conforme a sequência:

$$\text{High} \quad \text{Low} \quad \text{High} \quad \text{Low} \quad \ldots$$

Utilidades dessa distribuição

- **Redução de harmônicos:** Espalhar os pulsos ao longo do período ajuda a distribuir a energia dos harmônicos em frequências mais altas, reduzindo ruídos concentrados em frequências específicas.
- **Menor ruído mecânico e elétrico:** Em motores e outros dispositivos eletromecânicos, a distribuição uniforme dos pulsos evita vibrações bruscas.
- **Melhoria na qualidade do sinal:** Para aplicações de áudio, essa técnica reduz distorções e melhora a fidelidade sonora.

2.3 Modo Mark-Space

O modo Mark-Space é uma configuração alternativa do PWM no Raspberry Pi, na qual o sinal permanece em nível alto por um número fixo e consecutivo de ciclos de clock, seguido por um período em nível baixo, formando um único pulso contínuo dentro do período total.

Neste modo, o duty cycle é dado por:

$$DC = \frac{\texttt{DAT}}{\texttt{RNG}}$$

onde:

- `DAT`: número de ciclos consecutivos em nível alto;
- `RNG`: total de ciclos no período.

Visualmente, o pulso *High* fica concentrado no início do período, semelhante ao PWM tradicional encontrado em muitos microcontroladores.

Vantagens do Mark-Space

- **Linearidade e previsibilidade:** O duty cycle está diretamente relacionado ao valor programado, facilitando o controle.
- **Redução de distorções:** Ideal para aplicações sensíveis, como geração de áudio, pois minimiza variações de fase e ruído.
- **Simplicidade operacional:** O hardware gerencia a geração do sinal, evitando complicações com buffers ou FIFO.

2.4 Resumo comparativo entre os modos *default* e Mark-Space

Aspecto	PWM default (pulsos espaçados)	PWM Mark-Space
Formação do pulso	Múltiplos pulsos curtos espaçados	Pulso contínuo no início do período
Controle do duty cycle	Duty cycle dividido em intervalos	Duty cycle contínuo e direto
Harmônicos e ruído	Harmônicos espalhados, ruído menor	Sinal estável e linear
Complexidade	Mais complexo (FIFO e espaçamento)	Mais simples, gerenciado pelo hardware
Aplicações típicas	Controle geral, iluminação, motores	Áudio, geração de tons estáveis

Um guia do blog *youngkin* explica as diferenças entre os dois modos de maneira detalhada.

3 Estudo do Periférico de PWM do Raspberry Pi 2

3.1 Arquitetura do chip BCM2835/BCM2836

A implementação do PWM no Raspberry Pi é centrada no periférico descrito no *Broadcom BCM2835 ARM Peripherals Manual* (são os mesmos periféricos do BCM2836), que detalha a arquitetura interna, endereçamento e funcionamento dos registradores de controle. O módulo PWM é capaz de gerar até dois canais independentes de modulação, cada um com registradores próprios de configuração de *range*, dados e modos de operação.

O PWM interage diretamente com:

- O subsistema de *clocks* (para definição da frequência base).
- O módulo de GPIO (para roteamento do sinal PWM aos pinos físicos).
- O barramento APB (*Advanced Peripheral Bus*), responsável pela escrita/leitura dos registradores.

Infelizmente, o manual da Broadcom não informa acerca de um módulo **essencial** para o funcionamento da PWM: O *clock manager*, ou "gerenciador de clock", que é responsável por disponibilizar frequências de *clock* a partir de diversas fontes para periféricos como o de PWM.

Todas as informações acerca do gerenciador de clock da PWM foram obtidas através do adendo *BCM2835 Audio and PWM clocks*, referente a uma seção faltante do manual. Pode-se notar uma semelhança evidente entre esse módulo e o módulo de controle de *clock* para GPIO, que está presente no manual original.

3.2 Controle do *Clock Manager*

3.2.1 Registradores do *Clock Manager*

Seguem as funcionalidades dos registradores do *Clock Manager*. A base dos endereços antes do *offset* está em **0x7E1010A0**

Tabela 1: Registradores do *Clock Manager* para a *PWM*

Nome	Offset	Bits	Função
`CM_PWM_CTL`	0xA0-0xA3	32	Controle da fonte de clock do PWM.
`CM_PWM_DIV`	0xA4-0xA7	32	Configuração do divisor de clock para o PWM.

Tabela 2: Bits do registrador `CM_PWM_CTL`

Bits	**Nome**	**Função**
0-3	SRC	Define a fonte do clock.
4	ENAB	Habilita o clock e liga o bit BUSY.
5	KILL	Desabilita o clock e desliga o bit BUSY.
6	-	Inutilizado.
7	BUSY	Informa que o clock está sendo gerado.
8	FLIP	Inverte o bit de saída do clock.
9-10	MASH	Filtro que impõe limites de tensão a depende do divisor.
11-23	-	Inutilizado.
24-31	PASSWD	Senha fixa para alterar os outros bits.

Tabela 3: Bits do registrador `CM_PWM_CTL`

Bits	**Nome**	**Função**
0-11	DIVF	Parte fracionária do divisor de clock.
12-23	DIVI	Parte inteira do divisor de clock.
24-31	PASSWD	Senha fixa para alterar os outros bits.

Segue a relação de valor numérico de cada fonte de clock possível:

Tabela 4: Fontes possíveis para os bits `SRC` do registrador `CM_PWM_CTL`

Valor numérico decimal	**Nome**
0	GND
1	oscillator
2	testdebug0
3	testdebug1
4	PLLA per
5	PLLC per
6	PLLD per
7	HDMI auxiliary
8-15	GND

3.2.2 Permissionamento para modificações e geração do clock

Em ambos os registradores expostos, nota-se que há uma porção de 8 bits denominada **PASSWD**. Para se alterar o valor de quaisquer bits de cada registrador, essa

porção deve ser escrita **simultaneamente** com a senha padrão, que é **0x5A000000**. Vale ressaltar que essa porção de bits permite apenas leitura.

O clock é habilitado a partir do bit **ENAB**, que requisita a componentes específicos de hardware que a geração do clock seja iniciada. Uma vez que isso é feito, após alguns ciclos de execução, o bit **BUSY** é ativado, indicando sucesso na operação.

Para abortar a geração do clock, basta ativar o bit **KILL**, que irá fazer o inverso do bit **ENAB**, parando a operação e desativando o bit **BUSY**.

É importante notar que o bit **BUSY** também aceita somente leitura, ou seja, é apenas um indicativo do estado atual de geração do clock, que deve ser controlado de fato pelos bits **ENAB** e **KILL**, cujas ações de ativação são consideradas "*one shot operations*". Isto é, são equivalentes a **solicitações** para a execução das ações especificadas. A leitura desses dois sinais não tem um significado a ser abastraído.

O adendo também traz dicas importantes como nunca alterar os bits **SRC** enquanto **BUSY** estiver ativado.

3.2.3 Definição da frequência de clock da PWM

O clock do PWM é derivado do pode ser derivado de diversas fontes, portanto **não pode ser confundido com o clock padrão de 900Hz do processador**.

Normalmente, é utilizada a fonte *oscillator*, que funciona tipicamente a **19.2MHz**.

A frequência final do clock da PWM é determinada pela equação:

$$f_{\text{clock}} = \frac{f_{\text{clock base}}}{\text{divisor}} \tag{1}$$

Onde:

- $f_{\text{clock base}}$ — Frequência base fornecida pelo *Clock Manager* (através da fonte pré-definida através do registrador `CM_PWM_CTL`).
- `divisor` — Divisão do clock base, configurada no registrador `CM_PWM_DIV`. Tem parte fracionária e parte inteira.

Essa flexibilidade permite ajustar o PWM tanto para aplicações de baixa frequência (controle de motores) quanto para aplicações de alta frequência (geração de áudio de alta fidelidade).

Os modos de operação utilizando **MASH** e **FLIP** não foram explorados no desenvolvimento da lib.

3.3 Controle da *PWM*

3.3.1 Registradores da *PWM*

Seguem as funcionalidades dos registradores do periférico de *PWM*. A base dos endereços antes do offset está em **0x7E20C000** (essa informação não está presente no manual, mas sim em uma errata referente à página 141).

Atenção aos espaços inutilizados.

Tabela 5: Registradores do módulo PWM

Nome	Offset	Bits	Função
PWM_CTL	0x00-0x03	32	Controle geral dos canais e modos do PWM.
PWM_STA	0x04-0x07	32	Controle geral dos canais e modos do PWM.
PWM_DMAC	0x08-0x0B	32	Controle geral dos canais e modos do PWM.
-	0x0C-0x0F	32	Inutilizado.
PWM_RNG1	0x10-0x13	32	*Range* (período em alto) do canal 1.
PWM_DAT1	0x14-0x17	32	*Data* (período completo) do canal 1.
PWM_FIF1	0x18-0x1B	32	FIFO de dados para transmissão serializada no PWM.
-	0x1B-0x1F	32	Inutilizado.
PWM_RNG2	0x20-0x23	32	*Range* (período em alto) do canal 2.
PWM_DAT2	0x24-0x27	32	*Data* (período completo) do canal 2.

Tabela 6: Bits do registrador `PWM_CTL`

Bit	Nome	Função
0	PWEN1	Habilita canal 1.
1	MODE1	Seleciona modo (0 = PWM tradicional, 1 = M/S Enable).
2	RPTL1	Repetição do último dado no FIFO para canal 1.
3	SBIT1	Estado fixo de saída quando o canal 1 está parado.
4	POLA1	Inverte polaridade do canal 1.
5	USEF1	Habilita uso do FIFO para canal 1.
6	CLRF1	Limpa a FIFO (única para ambos os canais). Operação one shot.
7	MSEN1	Habilita uso do modo Mark-Space para canal 1.
8	PWEN2	Habilita canal 2.
9	MODE2	Seleciona modo (0 = PWM tradicional, 1 = M/S Enable) para canal 2.
10	RPTL2	Repetição do último dado no FIFO para canal 2.
11	SBIT2	Estado fixo de saída quando o canal 2 está parado.
12	POLA2	Inverte polaridade do canal 2.
13	USEF2	Habilita uso do FIFO para canal 2.
14	-	Inutilizado.
15	MSEN2	Habilita uso do modo Mark-Space para canal 2.
16-31	-	Inutilizados.

O registrador `PWM_FIF1` trata da entrada da FIFO, que será mais detalhada adiante.

Os registradores `PWM_STA` e `PWM_DMAC` não foram utilizados no desenvolvimento da lib.

3.3.2 Frequência e período finais

$$f_{\text{pwm}} = \frac{1}{T_{\text{total}}} = \frac{f_{\text{clock}}}{\text{range}} \tag{2}$$

Onde:

- f_{clock} — Definido na equação 1.
- T_{total} — Período completo de cada pulso.
- f_{pwm} — Frequência de cada pulso.
- `range` — Configurado diretamente em `PWM_RNG1` ou `PWM_RNG2`.

3.3.3 *Duty Cycle*

Os registradores referentes ao período da PWM (`PWM_RNG` e `PWM_DAT`) são inteiros de 32 bits. O *duty cycle* é definido a partir da seguinte equação:

$$DC = \frac{\text{range}}{\text{data}} \tag{3}$$

Onde:

- `DC` — *Duty Cycle*.
- `data` — Configurado diretamente em `PWM_DAT1` ou `PWM_DAT2`.
- `range` — Configurado diretamente em `PWM_RNG1` ou `PWM_RNG2`.

3.3.4 FIFO de dados para a PWM

O registrador `PWM_FIF1` corresponde à entrada da *FIFO* (First In, First Out), utilizada para transmissão serializada de dados ao periférico de PWM. Essa estrutura funciona como um buffer intermediário de 32 bits, que armazena múltiplos valores de `DATA` de forma sequencial. Assim, em vez de atualizar constantemente os registradores `PWM_DAT1` ou `PWM_DAT2`, o programador pode carregar uma série de valores na *FIFO*, que serão consumidos automaticamente pelo módulo de PWM.

O uso da *FIFO* é habilitado pelos bits `USEF1` e `USEF2` no registrador `PWM_CTL`, de forma independente para cada canal. Quando essa funcionalidade está ativa, cada amostra retirada da *FIFO* substitui o valor de `DATA` na geração do próximo ciclo de PWM. Isso permite, por exemplo, implementar saídas de áudio digital (streaming de amostras) ou geração de formas de onda arbitrárias.

O controle e monitoramento da *FIFO* é feito pelo registrador `PWM_STA`, que contém indicadores de estado, tais como:

- **FULL** — indica que a *FIFO* está cheia, impedindo a escrita de novos valores.
- **EMPTY** — indica que a *FIFO* foi completamente consumida.
- **ERR** — sinaliza erro de escrita quando ocorre tentativa de carregar a *FIFO* já cheia.

Por fim, o bit `CLRF1` no registrador `PWM_CTL` pode ser usado como operação *one shot* para esvaziar a *FIFO*, garantindo que não haja resíduos de dados antigos antes de uma nova transmissão.

Essa arquitetura de buffer em fila, aliada ao uso de DMA através do registrador `PWM_DMAC`, possibilita transmissões contínuas e com baixa sobrecarga da CPU, essencial para aplicações de áudio e geração de sinais complexos.

A FIFO não foi muito explorada, porém fforam criadas funções para um uso básico na lib.

4 Implementação Bare-Metal

4.1 Integração com *libpi* e módulo de GPIO existente

A base do nosso código foi o módulo de GPIO da *libpi*, criado pelo professor Bruno, que permite acesso direto aos registradores da Raspberry Pi. A partir dele, desenvolvemos funções específicas para configurar e manipular o periférico PWM, habilitando controle bare-metal sem intermediários.

4.2 Funções desenvolvidas

`pwm_start` Inicializa o clock do PWM com um divisor que define a frequência. Primeiro, mata o clock atual para resetá-lo, depois configura o divisor (parte inteira e fracionária) e reativa o clock usando o oscilador interno como fonte.

```
void pwm_start(float div) {
  if (div <= 0 || div >= 4096) {
    div = DEFAULT_DIV; // divisor padrão
  }
  uint32_t divi = (uint32_t)div;
  float frac_part = div - (float)divi;
  uint32_t divf = (uint32_t)(frac_part * 4096);

  CM_PWM_REG(ctl) = CM_PWM_CTL_KILL | CM_PWM_PASSWORD; // Mata clock
  CM_PWM_REG(div) = CM_PWM_PASSWORD | (divi << 12) | divf; // Configura
  ↪  divisor
  CM_PWM_REG(ctl) = CM_PWM_CTL_ENABLE | CM_PWM_CTL_SRC_OSC |
  ↪  CM_PWM_PASSWORD; // Ativa clock
}
```

`pwm_config` Configura o canal PWM escolhido (0 ou 1), habilitando o modo Mark-Space (MSEN) e o canal. Também permite configurar se será usado FIFO e se o canal deve repetir dados quando o FIFO estiver vazio.

```
void pwm_config(unsigned int channel, bool use_fifo, bool repeat_on_empty)
↪  {
  if (channel != 0 && channel != 1) return;

  uint32_t settings = 0;
  if (channel == 0) {
    settings = (1 << 7) | (1 << 0); // MSEN1 e PWEN1 habilitados
    if (use_fifo) {
      settings |= (1 << 5); // USEF1
      if (repeat_on_empty) settings |= (1 << 2); // RPTL1
    }
  } else {
    settings = (1 << 15) | (1 << 8); // MSEN2 e PWEN2 habilitados
    if (use_fifo) {
      settings |= (1 << 13); // USEF2
      if (repeat_on_empty) settings |= (1 << 10); // RPTL2
    }
  }
  PWM_REG(ctl) |= settings;
}
```

`pwm_set_duty_cycle` Define o ciclo ativo do PWM ajustando o `range` (período) e o valor de `data` (tempo ativo). Para cada canal, são usados registradores diferentes.

```
void pwm_set_duty_cycle(unsigned channel, int range, int data) {
  if (channel != 0 && channel != 1) return;

  if (channel == 0) {
    PWM_REG(rng1) = range;
    PWM_REG(dat1) = data;
  } else {
    PWM_REG(rng2) = range;
    PWM_REG(dat2) = data;
  }
}
```

`pwm_set_polarity` Controla a polaridade do sinal PWM: se ativo em nível alto (0) ou invertido (1).

```
void pwm_set_polarity(int channel, int polarity) {
  if (channel != 0 && channel != 1) return;

  if (channel == 0) {
```

```
    PWM_REG(ctl) = (PWM_REG(ctl) & ~0b000100000) | (polarity << 4);
  } else {
    PWM_REG(ctl) = (PWM_REG(ctl) & ~0b0001000000000000) | (polarity <<
    ↪  12);
  }
}
```

Manipulação da FIFO Permite limpar a fila de dados FIFO e escrever novos dados, além de verificar se a fila está cheia.

```
void pwm_clear_queue(void) {
  PWM_REG(ctl) |= (1 << 6);
}

int pwm_write_queue(uint32_t data) {
  if (pwm_full_queue()) return -1;
  PWM_REG(fif1) = data;
  return 0;
}

int pwm_full_queue() {
  return (PWM_REG(sta) & (1 << 0)) ? 1 : 0;
}
```

4.3 Arquivos adicionais e makefile

Para conseguirmos fazer o acréscimo dessas funcionalidade do módulo da PWM sobre a libpi, precisamos fazer algumas configurações adicionais. O primeiro desses arquivos foi o boot.s, que faz a preparação inicial adequada para execução do conteúdo da library. Esse arquivo foi desenvolvido com base no arquivo de boot disponibilizado durante as aulas, mas mantendo apenas os procedimentos essenciais para execução do código no processador (preparando o segmento .bss). Além disso, precisamos ajustar o makefile original da libpi para podermos executar. O makefile se trata de um arquivo com o conjunto de comandos necessários para compilar e associar os códigos da library com o linker. No makefile, adicionamos instruções de compilação dos arquivos adicionais que geramos (pwm.c e boot.s), além de limpar os resultados de compilação. Ao utilizar o comando "make", todos os códigos .c são compilados e seus objetos são associados ao linker.ld, por meio das funções do gcc. Ao usar o comando make-gdb, o debugger gdb é aberto e a comunicação serial com a placa é inicalizada, definida pela porta configurada no makefile. As seções do makefile estão demarcadas por comentários em seu próprio corpo.

5 Testes

Para validar o funcionamento, durante a implementação do módulo da PWM, realizamos múltiplos testes práticos. O primeiro foi com a visualização do formato da onda quadrada no osciloscópio. Durante o processo de compreensão do funcionamento do módulo PWM, tentamos validar com o osciloscópio, e, portanto, efetuamos diversas tentativas com a intenção de validar os diferentes canais e modos:

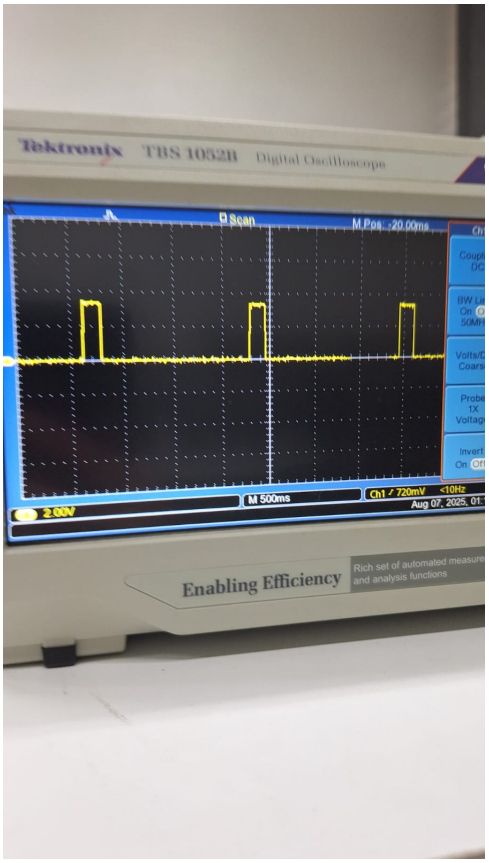

Figura 1: Osciloscópio mostrando PWM

Para validar o modo FIFO, salvamos múltiplas PWMs utilizando as funções que desenvolvemos, e verificamos também no osciloscópio. Após a validação da funcionalidade pelo osciloscópio, passamos a fazer testes em dispositivos controláveis pela PWM. Para isso, escolhemos um Led, e testamos tanto no modo Mark-Space quanto no modo padrão. Ao testar no modo mark-space, com um período razoável, pudemos ver o LED piscando. Enquanto no modo MSEN = 0, o led mantém um nível de luminosidade estático. Por fim, após os testes do Led, fizemos os testes utilizando

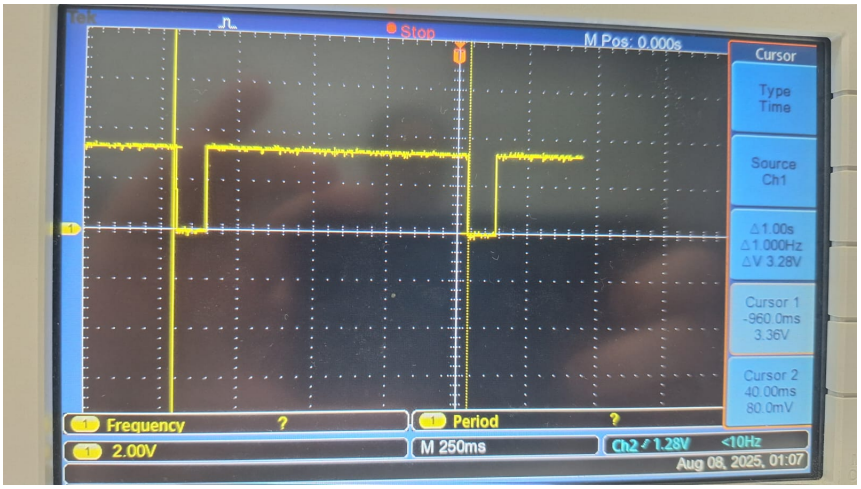

Figura 2: Osciloscópio mostrando teste com polaridade invertida

um buzzer, como forma de valiadção final da ideia de utilizar o projeto para síntese de música. Para tal, fizemos o seguinte trecho de código na função main:

```
int main() {
    gpio_init(12, GPIO_FUNC_ALT0);
    gpio_init(13, GPIO_FUNC_ALT0);

    pwm_start(19.2); // Clock base

    pwm_set_duty_cycle(0, 1000000, 100000);
    pwm_set_duty_cycle(1, 1000000, 100000);

    pwm_config(0, false, false);
    pwm_config(1, false, false);
    // pwm_set_polarity(1, 1);

    int time = 2;
    while (1) {

        // ======= Trecho 1 =======
        pwm_set_duty_cycle(0, 1516, 379); delay(250000*time); // E5
        pwm_set_duty_cycle(0, 1516, 189); delay(125000*time); // E5 12.5%
        pwm_set_duty_cycle(0, 1516, 0);   delay(125000*time); // silêncio
        pwm_set_duty_cycle(0, 1516, 379); delay(250000*time); // E5
        pwm_set_duty_cycle(0, 1516, 0);   delay(125000*time); // silêncio
        pwm_set_duty_cycle(0, 1911, 478); delay(250000*time); // C5
```

```
pwm_set_duty_cycle(0, 1516, 379); delay(250000*time); // E5
pwm_set_duty_cycle(0, 1275, 319); delay(500000*time); // G5
pwm_set_duty_cycle(0, 1516, 0);   delay(250000*time); // silêncio
pwm_set_duty_cycle(0, 3822, 956); delay(250000*time); // G4
pwm_set_duty_cycle(0, 3822, 0);   delay(125000*time); // silêncio
pwm_set_duty_cycle(0, 1911, 478); delay(250000*time); // C5
pwm_set_duty_cycle(0, 1911, 0);   delay(125000*time); // silêncio
pwm_set_duty_cycle(0, 3822, 956); delay(250000*time); // G4
pwm_set_duty_cycle(0, 3822, 0);   delay(125000*time); // silêncio
pwm_set_duty_cycle(0, 4545, 1136);delay(250000*time); // E4
pwm_set_duty_cycle(0, 3413, 853); delay(250000*time); // A4
pwm_set_duty_cycle(0, 3061, 765); delay(250000*time); // B4
pwm_set_duty_cycle(0, 3232, 161); delay(125000*time); // Bb4 12.5%
pwm_set_duty_cycle(0, 3413, 853); delay(500000*time); // A4
```

Os tons são determinados pela PWM configurada, que determina a frequência das notas, e o tempo que elas soam são determinados pelo tamanho do delay aplicado. Os testes com led e o buzzer podem ser vistos no seguinte vídeo.

6 Conclusão

O desenvolvimento do projeto *SpotiPi* possibilitou compreender de forma aprofundada o funcionamento do periférico de PWM do Raspberry Pi em modo bare-metal. Através da implementação e testes realizados, foi possível não apenas reproduzir sinais básicos, mas também explorar a geração de áudio e a síntese de tons musicais em um buzzer, aproximando-se da lógica utilizada por consoles clássicos de 8-bit.

Além do aprendizado técnico em manipulação direta de registradores e configuração do *clock manager*, o projeto evidenciou os desafios da ausência de documentação oficial completa e a importância de construir uma base própria de bibliotecas e funções reutilizáveis.

Em suma, o trabalho cumpriu seus objetivos de estudo e prática em sistemas embarcados de baixo nível, oferecendo uma base sólida para futuras extensões, como a reprodução de melodias completas ou a sincronização de áudio com outros periféricos.