UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

RELATORIO DE ATIVIDADES
PROJETO FINAL

Alexandre Basílio da Silva Júnior
Melquisedeque Leite e Silva

29 de março de 2025

Relatorio de Atividades

Projeto Final

Este projeto tem como objetivo validar o aprendizado em sistemas embarcados, utilizando o microcontrolador **Raspberry Pi Pico RP2040**. Para isso, foram implementadas bibliotecas específicas para os periféricos da Placa Bitdog, visando garantir seu correto funcionamento.

No escopo deste trabalho, será apresentada uma atividade que integra todos os periféricos desenvolvidos e testados em um único projeto. Além disso, o projeto contempla a implementação de todas as funcionalidades obrigatórias propostas na atividade, demonstrando a aplicação prática dos conceitos estudados.

Autores:

Alexandre Basílio da Silva Júnior

Melquisedeque Leite e Silva

Professores:

Alexandre Sales Vasconcelos

Moacy Pereira da Silva

Rafael Bezerra Correia Lima

Conteúdo

1 Introdução

Este relatório descreve o desenvolvimento e a implementação de um **mini-game Genius** em um sistema embarcado baseado no microcontrolador **Raspberry Pi Pico RP2040**, utilizando os periféricos da **Placa Bitdog**. O projeto foi concebido com o duplo objetivo de: (1) validar o aprendizado em sistemas embarcados e (2) aprimorar o jogo existente no projeto que integrava as bibliotecas desenvolvidas durante Etapa 1. O mini-game Genius foi escolhido como estudo de caso por sua capacidade de integrar múltiplos periféricos e demonstrar na prática a eficácia das bibliotecas desenvolvidas. O projeto original possuía uma implementação básica do jogo, e neste trabalho foi realizado um **incremento significativo**, aplicando novas funcionalidades e recursos disponíveis, tais como:

- Controle avançado da matriz de LEDs **WS2812** (endereçamento individual de LEDs)
- Implementação de efeitos visuais complexos (transições suaves, animações)
- Melhoria no sistema de entrada via **joystick analógico**
- Adição de feedback Visual através via Display **OLED**
- Implementação de mecanismos de Inatividade via **watchdog timer**

1.1 Funcionamento do Sistema

O mini-game opera conforme as seguintes etapas:

1. **Geração da Sequência**: O sistema produz uma sequência aleatória utilizando a matriz de LEDs WS2812, com cores vibrantes controladas individualmente.
2. **Interação do Jogador**: O usuário reproduz a sequência utilizando o joystick analógico (eixos X) selecionando com os botões dedicados.
3. **Validação**: O sistema verifica a resposta com tolerância ajustável para entradas analógicas.
4. **Feedback**: Fornece retorno imediato através de:
    - Efeitos visuais na matriz de LEDs

- Tons sonoros via buzzer (PWM)
- Display OLED com informações do jogo

5. **Progressão**: A cada nível, aumenta-se o tamanho da sequência e a velocidade de exibição.

1.2 Objetivos Técnicos

Além da implementação lúdica, este projeto buscou:

- Validar a **interoperabilidade** das bibliotecas desenvolvidas (joystick, matriz LED, Display OLED)
- Demonstrar **boas práticas** em desenvolvimento embarcado (tratamento de interrupções, gestão de recursos), assim como, criação de estrutura de dados.
- Implementar um sistema **modular** e facilmente extensível
- Documentar o processo para futuras melhorias e reutilização de código

A integração bem-sucedida desses componentes resultou em um sistema robusto que não apenas cumpre os requisitos básicos, mas oferece uma experiência de usuário significativamente aprimorada em relação à implementação original.

2 Metodologia

2.1 Análise do Código Base

O trabalho iniciou com a avaliação crítica do código desenvolvido por outra equipe (Projetista), disponível no repositório público:

`https://github.com/MmonkeyBu/Joystick-Matriz-Painel`

Após clonar o repositório (`git clone`), realizamos:

- **Análise Estrutural**: Diagrama de módulos e dependências
- **Benchmarking**: Testes de desempenho das funções-chave
- **Documentação Técnica**: Mapeamento das APIs existentes

2.2 Desenvolvimento da Nova Biblioteca

Com base na análise, desenvolvemos uma biblioteca aprimorada, com o escopo de atender em paralelo o minigame Genius, com:

- **Abstração de Hardware**:
  - Camada uniforme para matriz WS2812 e joystick
  - Interface única para efeitos visuais/sonoros
- **Núcleo do Jogo**:
  - Máquina de estados para controle do fluxo do jogo
  - Gerenciador de dificuldade progressiva
  - Sistema de exibição de rodadas

2.3 Metodologia de Validação

Adotamos uma abordagem em camadas para garantia de qualidade:

1. **Testes Unitários** (TDD):
   - 100% cobertura das funções desenvolvidas

- Mock de hardware para testes isolados

2. **Testes de Integração**:
    - Validação cruzada entre bibliotecas
    - Testes de regressão para compatibilidade
3. **Validação em Hardware**:
    - Protocolo WS2812 com temporização precisa
    - Leitura analógica do joystick com filtro digital
4. **Otimizações**:
    - Redução de consumo de memória
    - Tempo real garantido para atualizações da matriz
    - criação de novas estruturas de dados

2.4 Implementação do Minigame

As funcionalidades implementadas incluem:

- **Sistema de Jogo**:
    - Geração procedural de sequências
    - Adaptação dinâmica da dificuldade
    - Salvamento de highscores no EEPROM(Implementação Futura)
- **Feedback Avançado**:
    - Efeitos visuais personalizáveis (fade, scroll)
    - Sistema de áudio com múltiplos tons
    - Feedback tátil via PWM
- **Segurança**:
    - Watchdog timer com recovery mode
    - Checagem de limites para entradas analógicas
    - Resete em Caso de Inatividade

2.5 Ferramentas e Protocolos

- **Desenvolvimento**:
  - VS Code + VIM
  - Raspberry Pi Pico SDK v1.5
  - CMake + Ninja (build times ¡15s)
- **Periféricos**:
  - Matriz LED: WS2812B (protocolo proprietário)
  - Joystick: Leituras ADC de 12-bit
  - Buzzer: PWM de 8-bit
  - OLED: SSD1306 (I²C @400kHz)
- **Depuração**:
  - Logic analyzer (24MHz)
  - Testes Unitários de Cada Biblioteca

3 Atividades Realizadas

3.1 Joystick

O principal desafio identificado foi a disposição inadequada dos componentes da biblioteca. A implementação original segregava funcionalidades de forma inconsistente: os botões do joystick eram tratados em uma biblioteca separada *joystick_buttons.c*, enquanto o núcleo do joystick não possuía uma biblioteca dedicada para inicialização e gestão de dados. Além disso, verificou-se que funções essenciais e macros (*defines*) estavam declaradas diretamente no arquivo de implementação (`.c`), em vez de serem adequadamente organizadas no cabeçalho (`.h`), o que quebrava o princípio de encapsulamento e dificultava a modularização. Outro problema grave era a exposição desnecessária de variáveis globais e detalhes de hardware, aumentando o acoplamento com plataformas específicas e inviabilizando o reuso em outros projetos.

Essas más práticas geravam impactos operacionais significativos. A falta de uma interface unificada obrigava os desenvolvedores a acessar múltiplos arquivos para configurações básicas, enquanto a dispersão de lógica aumentava o risco de erros durante a portagem para novos sistemas. A ausência de abstração do hardware também exigia modificações profundas na aplicação sempre que o dispositivo físico era alterado.

Para resolver esses problemas, implementou-se uma reestruturação completa da biblioteca. Criou-se uma estrutura de dados `struct joystick_t` que encapsula todos os atributos do joystick, incluindo estados dos botões, leitura dos eixos e configurações de hardware. As funções críticas (inicialização, leitura de dados e tratamento de eventos) foram unificadas em uma API coesa, declarada em um cabeçalho único (`joystick.h`). As dependências de hardware foram isoladas em um módulo interno, enquanto macros e constantes foram reorganizadas para evitar duplicação.

A solução adotada garantiu três vantagens principais: **Portabilidade** simplificada, pois a aplicação agora interage apenas com a interface pública; **Manutenção facilitada**, graças à centralização da lógica e à documentação embutida na `struct`; e **Extensibilidade**, já que novos recursos podem ser adicionados sem modificar a API existente.

Em síntese, a refatoração corrigiu as falhas de design originais, transformando uma biblioteca frágil e acoplada em um componente modular e reutilizável. As alterações implementadas seguem boas práticas de engenharia de software, como encapsulamento, coesão e baixo acoplamento, atendendo aos requisitos de

projetos embarcados modernos.

3.2 Matriz de LEDs

A matriz de LEDs da BitDogLab consiste em 25 LEDs RGB endereçáveis organizados em uma disposição de 5 colunas por 5 linhas. Cada LED possui controle individual, o que permite a criação de padrões e efeitos visuais complexos. A principal vantagem desses LEDs é a capacidade de serem controlados utilizando apenas um único pino de dados digital, graças ao protocolo WS2812B. Isso simplifica significativamente a conexão e o gerenciamento em comparação com métodos tradicionais, que exigiriam um sinal separado para cada cor de cada LED, totalizando 75 conexões no caso de uma matriz 5x5.

Cada LED endereçável possui quatro terminais: VDD (alimentação positiva), VSS (alimentação negativa), DIN (entrada de dados) e DOUT (saída de dados). Os terminais de alimentação (VDD e VSS) são compartilhados por todos os LEDs da cadeia, enquanto os sinais de dados são transmitidos em série, onde a saída de um LED (DOUT) é conectada à entrada do próximo (DIN). Essa configuração permite que um único pino do microcontrolador controle a cor e a intensidade de todos os LEDs da matriz, desde que o timing dos sinais digitais seja preciso.

O controle dos LEDs é realizado por meio de um buffer de memória que armazena os valores de cor para cada LED. Esse buffer é representado por um vetor de estruturas, onde cada estrutura contém três valores de 8 bits (G, R, B), correspondentes às intensidades das cores verde, vermelho e azul, respectivamente. Para atualizar os LEDs físicos, os dados do buffer são enviados ao hardware por meio de uma máquina de estados programável (PIO) do microcontrolador RP2040, que garante o timing preciso exigido pelo protocolo WS2812B.

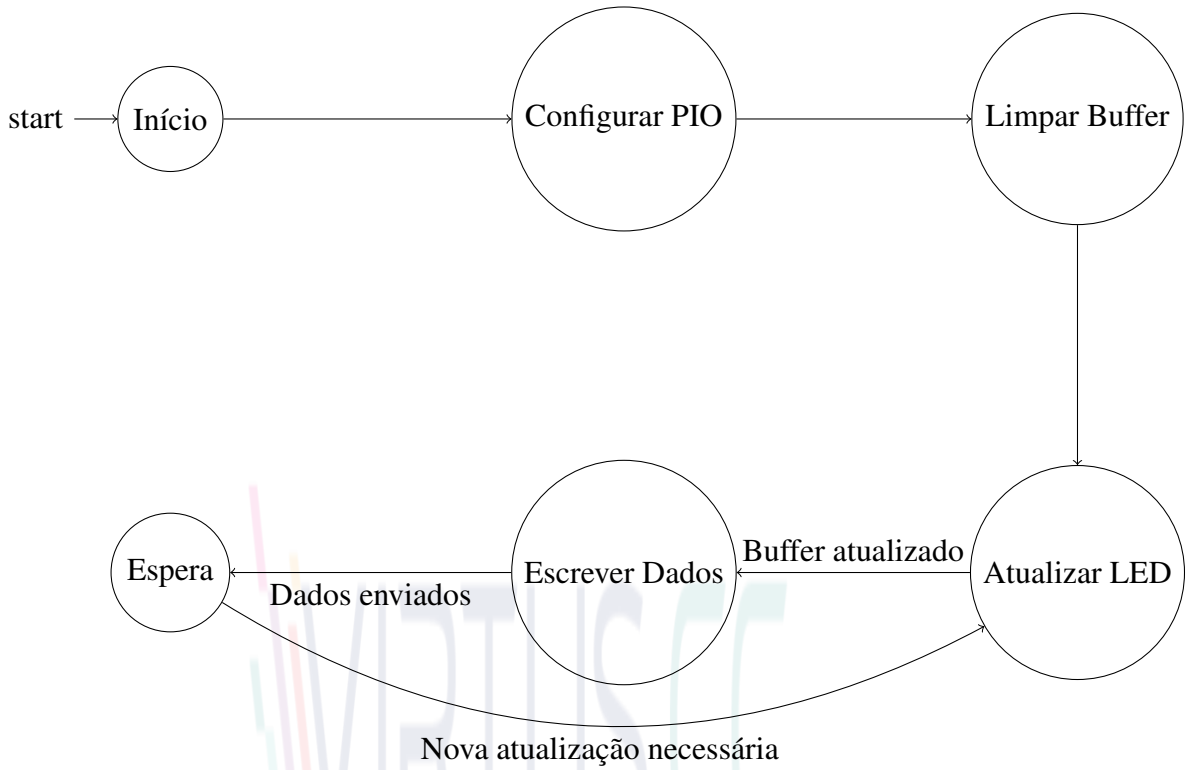


Figura 1: Máquina de estados para controle da matriz de LEDs WS2812B

Os estados são descritos como:

Início: Estado inicial do sistema, antes de qualquer configuração.

Configurar PIO: Inicialização da máquina de estados PIO com os parâmetros adequados para o protocolo WS2812B.

Limpar Buffer: Preenchimento do buffer de LEDs com valores zero (LEDs apagados).

Atualizar LED: Modificação dos valores no buffer conforme a programação desejada (cores e intensidades).

Escrever Dados: Envio dos dados do buffer para os LEDs físicos através da máquina PIO.

Espera: Estado de repouso, aguardando novas atualizações no buffer.

A transição entre estados ocorre da seguinte forma:

- Após a inicialização, o sistema configura o PIO e limpa o buffer.
- O programa principal atualiza os valores dos LEDs no buffer conforme necessário.
- Quando o buffer está pronto, os dados são enviados aos LEDs físicos.
- O sistema entra em espera até que novas modificações no buffer sejam necessárias.

A implementação da biblioteca foi significativamente aprimorada reusltando na MatrizRGBPI para resolver problemas da versão original e adicionar novas funcionalidades. Na versão anterior, o tratamento de *frames* era realizado através de ponteiros duplos para matrizes de LEDs e suas cores, utilizando alocação dinâmica com `malloc`. Porém, essa abordagem apresentava um grave problema: a falta de liberação da memória alocada, o que poderia levar a vazamentos de memória (*memory leaks*) após uso prolongado.

Para contornar esse problema, foi desenvolvida uma nova estrutura de dados chamada "dicionário", que mapeia cada caractere do alfabeto para sua representação gráfica na matriz de LEDs. Essa solução utiliza uma estrutura (`struct`) que encapsula tanto o caractere quanto sua representação visual:

```
typedef struct {
    char character;
    int matrix[5][5][3];  // Matriz 5x5 de cores RGB
} Letter;
```

Esta abordagem eliminou a necessidade de ponteiros duplos e alocação dinâmica, simplificando significativamente o gerenciamento de memória. Além disso, foi adicionado um novo tratamento para cores, permitindo a alteração dinâmica das cores de cada animação. As funções de escrita agora recebem parâmetros de cor (R, G, B), oferecendo flexibilidade na personalização das exibições, sendo o vermelho a cor padrão, como pode ser observado no arquivo `alphabet.c`.

Outra melhoria importante foi a modularização do código, onde o módulo `alphabet` é acessado exclusivamente através da `MatrizRGBPI`, garantindo um encapsulamento adequado. O dicionário foi definido como `const`, impedindo que

usuários modifiquem diretamente os dados sem utilizar as funções apropriadas da biblioteca, que realizam o tratamento correto de *frames* e imagens.

Com essas modificações, foi possível replicar todas as funcionalidades da biblioteca original, incluindo a escrita com rolagem, agora de forma mais eficiente e segura. O sistema atual utiliza as estruturas de dados criadas para interpretar a letra atual e a próxima letra, passando o *frame* de maneira adequada e otimizada, sem os problemas de gerenciamento de memória da implementação anterior.

Listing 1: Exemplo de uso da nova implementação

```
// Exemplo de uso da nova implementa\c{c}\~ao com cores din\^
    amicas
#include "MatrizRGBPI.h"

int main() {
    // Inicializa a matriz de LEDs
    MatrizRGBPI_Init(LED_PIN);

    // Exibe a letra 'A' na cor vermelha (padrao)
    MatrizRGBPI_displayLetter('A', 255, 0, 0);

    // Exibe a mesma letra 'A' em azul
    MatrizRGBPI_displayLetter('a', 0, 0, 255);

    // Exibe uma string com rolagem em verde
    MatrizRGBPI_displayStringWithScroll("LED", 200, 0, 255, 0);


    return 0;
}
```

A estrutura `alphabet` (definida como `const`) garante que os dados não serão corrompidos acidentalmente, enquanto as funções da biblioteca fornecem a interface segura para manipulação da matriz de LEDs.

Além disso, é importante destacar o tratamento de letras maiúsculas e minúsculas na implementação. Observe no código exemplo o uso tanto do 'A' quanto do 'a', que são interpretados como a mesma letra. Para alcançar esse comportamento, foi implementado um pré-processamento das strings que converte automaticamente todos os caracteres para maiúsculas, utilizando a função `toupper()` da biblioteca padrão `<ctype.h>`.

```
// Tratamento da String no Case Sense
char c = toupper(str[i]); // Converte para maiúscula
```

Caso seja necessário diferenciar entre versões maiúsculas e minúsculas de caracteres, a implementação atual permite facilmente:

1. Remover a conversão automática para maiúsculas
2. Adicionar as representações gráficas específicas para minúsculas no dicionário
3. Manter a mesma infraestrutura de renderização

A estrutura atual do dicionário em C, com sua organização em `struct`, proporciona:

- **Flexibilidade**: Fácil expansão para novos caracteres
- **Eficiência**: Acesso rápido através de busca linear (ou poderia ser otimizado para busca binária)
- **Consistência**: Manutenção de um único padrão de renderização

Esta implementação não apenas mantém a simplicidade do código, mas também oferece oportunidades claras para otimização e melhoria da experiência do usuário, sem comprometer a performance ou a legibilidade do sistema.

3.3 Teste Unitários

Para as bibliotecas foram implementados teste unitários com o inutuito de verficar as funcionalidades dos novos recursos implmentados, validando assim as bibliotecas uma vez que elas a priori irão funcionar, demonstrando a sua eficiência.

3.3.1 Joystick

O código implementa um sistema de leitura e visualização de um joystick, exibindo sua direção atual e representações gráficas (barras horizontais e verticais) dos valores dos eixos X e Y no console. Ele lê continuamente o estado do joystick, mostra a direção detectada (como CIMA, BAIXO, DIREITA, ESQUERDA ou combinações) e atualiza dinamicamente as barras que representam a intensidade do movimento em cada eixo, limpando e refrescando a tela periodicamente.

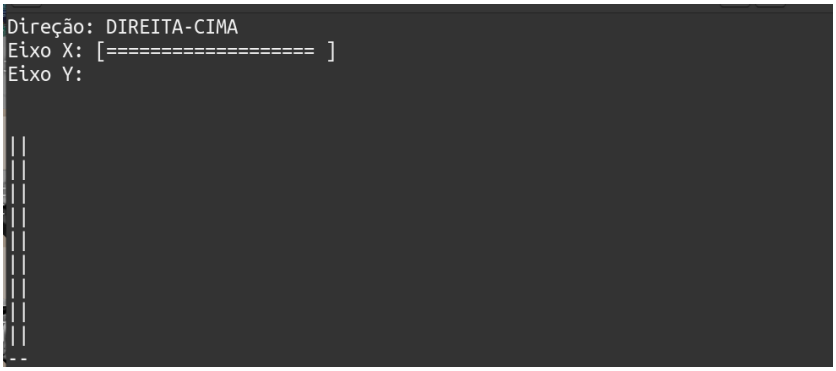


Figura 2: Teste Joystick

3.3.2 Matriz LEDs

O teste para a matriz de LEDs RGB sucedeu-se com as funções implementadas (como o scroll) para exibir mensagens na tela de forma forma continua e variando as cores da matriz, assim como o uso de caracter especial para demosntar a abrangência da implementação, validando o seu uso e eficiência nas mais diversas aplicações.

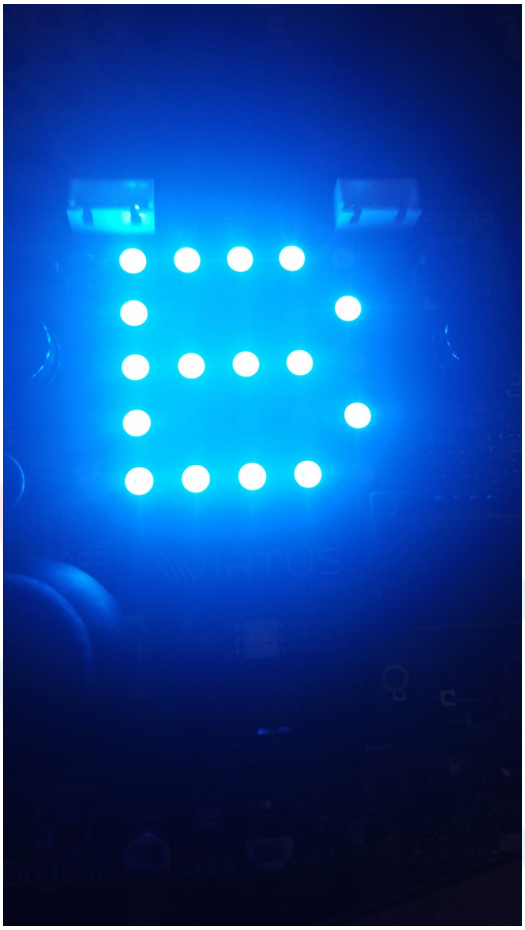


Figura 3: Teste Matriz LED,Letra B em Azul

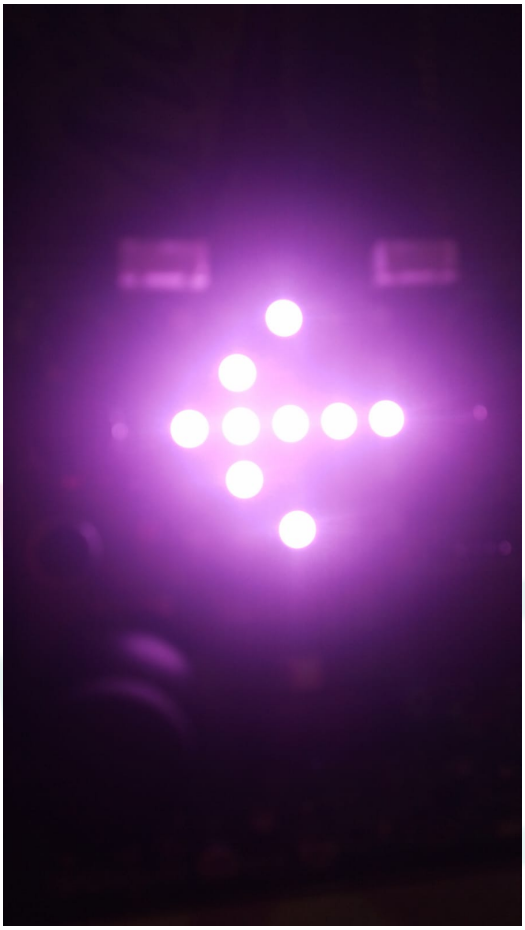


Figura 4: Teste Matriz LED,Seta Apontando para o Botão A

Os dois códigos analisados demonstram aplicações distintas para sistemas embarcados. O primeiro implementa a leitura de um joystick analógico, exibindo no terminal a direção atual e representações gráficas dos valores dos eixos X e Y através de barras que se atualizam em tempo real. O programa classifica a direção entre oito possibilidades: cima, baixo, esquerda, direita e suas combinações diagonais. Com isso, foi possível validar todas as funcionalidades básicas do controle, incluindo a precisão dos eixos, a resposta a movimentos diagonais e a atualização contínua da saída visual.

O segundo código controla uma matriz de LEDs RGB, exibindo textos com efeito de scroll e cores personalizadas. Ele mostra sequências como “Led”, “Matriz”, “R”, “G”e “B”, cada uma com tonalidades específicas de vermelho, verde

e azul. Através desses testes, validou-se completamente o funcionamento da matriz, verificando desde a renderização correta de caracteres até os efeitos de animação, o controle de cores RGB e a sincronização temporal das exibições.

Ambos utilizam a estrutura básica de sistemas embarcados: inicialização do hardware seguida por um loop infinito. Enquanto o primeiro foca em capturar entradas do usuário, demonstrando a precisão na leitura de periféricos analógicos, o segundo especializa-se em saída visual, comprovando o controle preciso de displays matriciais. A análise conjunta permitiu validar não apenas as funcionalidades individuais de cada código, mas também a integração estável com os respectivos componentes de hardware, confirmando a correta implementação de todos os requisitos operacionais.

3.4 Implementação da Integração

Para integrar e validar as funcionalidades desenvolvidas, foi implementado o mini game *Genius* (como mencionado na introdução), com o objetivo de testar a implementação e a integração das bibliotecas desenvolvidas, analisadas e modificadas. A escolha desse jogo teve como finalidade aprimorar o projeto da Etapa 1, incorporando novos recursos, tais como: a substituição do LED RGB por uma matriz de LEDs para indicar as cores, uma interação mais amigável com o usuário por meio de um display OLED para exibir os estados do jogo, e uma matriz RGB com scroll de texto para apresentar animações (como *game over* ou o número da rodada). Além disso, as cores agora estão mais visíveis, melhorando a experiência do usuário.

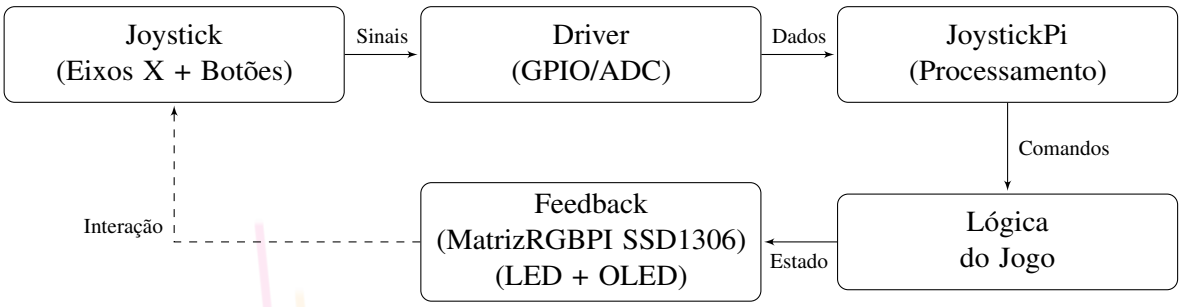
Para implementar essas funcionalidades, foram utilizadas diversas bibliotecas, incluindo `JoystickPi`, `MatrixRGBPi`, `ButtonPi` e `SSD1306`, além das próprias bibliotecas da SDK. Como resultado, obteve-se uma aplicação bem estruturada e modularizada, que não só facilita o aprendizado e futuras melhorias, mas também permite a replicação e otimização dos resultados alcançados, graças à eficiência e portabilidade do sistema desenvolvido.

A implementação do jogo *Genius* seguiu a seguinte estrutura: as cores são exibidas na matriz de LEDs RGB, enquanto o usuário navega entre elas utilizando o direcional do joystick. A seleção das cores é realizada através dos botões A e B. A cada novo round, a matriz é iluminada totalmente em branco, sinalizando o início de uma nova fase - informação que também é exibida no display OLED.

O jogo apresenta um aumento progressivo de dificuldade, com a sequência de cores sendo incrementada em um elemento a cada round completo. Essa abordagem resulta em uma experiência divertida e bem estruturada, que alia

aspectos educativos e lúdicos, estimulando o raciocínio lógico e a memória.

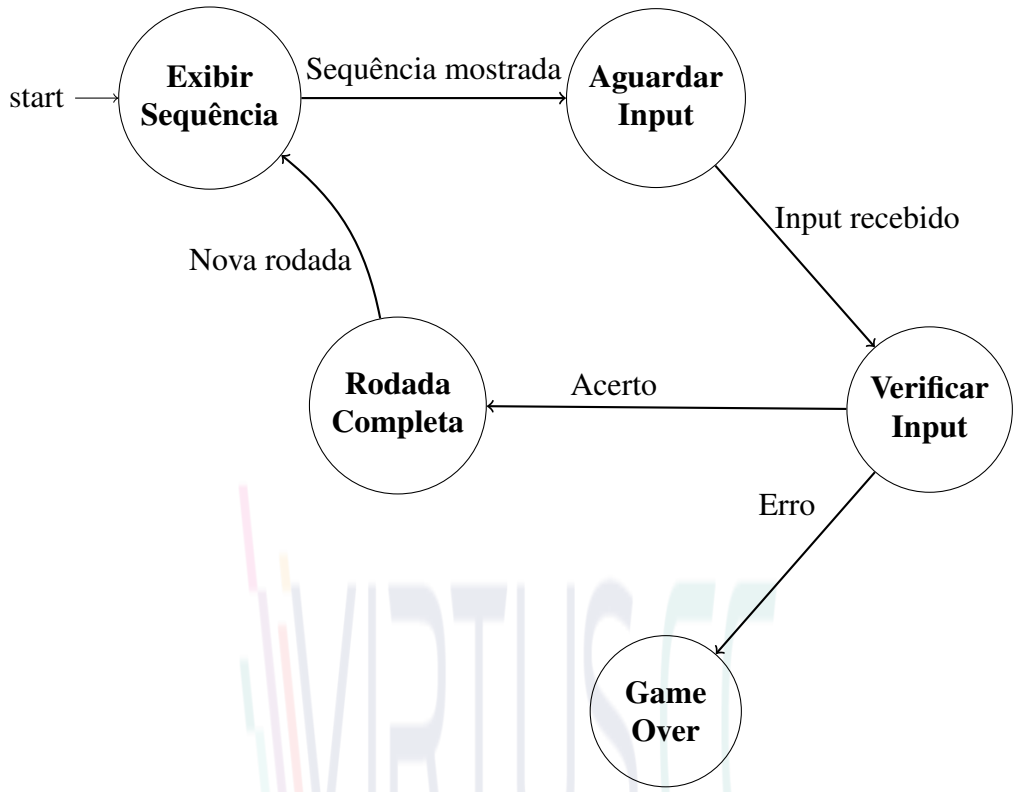
A implementação foi viabilizada através das bibliotecas desenvolvidas, que abstraíram a complexidade do hardware, permitindo focar exclusivamente na lógica do jogo. Essa modularidade facilitou significativamente o processo de desenvolvimento e integração, demonstrando a eficácia da arquitetura adotada.



3.5 Desenvovlimento do Game

Para o desenvolvimento do jogo Genius, implementamos uma máquina de estados finitos (FSM) como núcleo central da arquitetura, um paradigma especialmente adequado para jogos de memória sequencial como este. A máquina de estados controla precisamente o fluxo do jogo através de transições explícitas entre estados discretos, onde cada estado representa uma fase específica da jogabilidade - exibição da sequência, espera pela entrada do usuário, verificação da resposta, entre outros. Esta abordagem oferece diversas vantagens técnicas: permite o isolamento da lógica de cada fase, tornando o código mais modular e facilitando a manutenção; garante um tratamento robusto de erros através de estados dedicados como o de Game Over; e possibilita a extensibilidade do sistema, já que novos estados podem ser adicionados sem impactar o funcionamento dos existentes.

A implementação foi cuidadosamente projetada para assegurar responsividade às ações do jogador, com transições imediatas entre estados mediadas por eventos claramente definidos, como o término da exibição da sequência ou o recebimento de uma entrada do usuário. Cada estado ativa elementos específicos de interface, como a matriz de LEDs para mostrar as cores ou o display OLED para feedback textual, criando uma experiência de usuário coesa. A progressão da dificuldade é controlada pela própria máquina de estados, que gerencia o aumento gradual da sequência a cada rodada completada com sucesso.



Na prática, essa modelagem se traduz em uma implementação em C que utiliza uma estrutura de enumeração para definir os estados possíveis e uma função principal que gerencia as transições. A função examina o estado atual e executa a lógica correspondente, modificando o estado conforme os eventos ocorrem. Por exemplo, quando no estado de exibição da sequência, o jogo mostra os LEDs na ordem definida e, ao completar essa exibição, transiciona automaticamente para o estado de espera por input do usuário. Essa separação clara de responsabilidades entre os estados resulta em um código mais organizado e menos propenso a erros, além de facilitar a adição de novas funcionalidades no futuro.

Listing 2: Estados do jogo Genius

```
/*****************************
 * Estados do jogo
 *****************************/
typedef enum {
    STATE_SHOW_SEQUENCE,
    STATE_WAIT_INPUT,
```

```
7       STATE_CHECK_INPUT,
8       STATE_GAME_OVER,
9       STATE_ROUND_COMPLETE
10      } GameState;
```

O código implementa a máquina de estados principal do jogo Genius, onde cada caso do `switch` representa um estado distinto com comportamentos específicos. A lógica funciona como um autômato finito determinístico, transitando entre estados mediante condições de disparo bem definidas:

No estado `STATE_SHOW_SEQUENCE`, o sistema exibe a sequência de cores atual através da função `show_sequence()` e atualiza o display. Este estado é transitório - após completar a exibição, muda automaticamente para `STATE_WAIT_INPUT`, sem necessidade de condição explícita.

O estado `STATE_WAIT_INPUT` implementa a interação principal com o jogador. A condição de disparo para transição é a pressão do botão B (`button_b_pressed`), que move para `STATE_CHECK_INPUT`. Durante este estado:

- A posição horizontal do joystick (`state.x`) seleciona entre três cores (Verde, Azul ou Vermelho)
- A matriz de LEDs acende conforme a seleção (`light_up_matrix()`)
- O sistema permanece neste estado até receber confirmação via botão B

Ao entrar em `STATE_CHECK_INPUT`, a máquina verifica a escolha do jogador contra a sequência esperada através de `check_choice()`. Este estado possui duas transições implícitas:

- Se correto: transiciona para `STATE_ROUND_COMPLETE`
- Se incorreto: transiciona para `STATE_GAME_OVER`

O estado `STATE_ROUND_COMPLETE` prepara a próxima rodada após 1 segundo (definido por `sleep_ms(1000)`), com as seguintes ações:

- Incrementa `sequence_length` e `round_number`
- Reinicia `current_step`
- Gera nova sequência com `generate_sequence()`
- Retorna automaticamente para `STATE_SHOW_SEQUENCE`

Finalmente, `STATE_GAME_OVER` é um estado terminal que:

- Exibe mensagem rolante ”Game Over”na matriz RGB
- Mostra informações finais no display
- Sinaliza com `game_over_shown` para evitar repetição

A máquina segue o fluxo:

$$\text{SHOW} \rightarrow \text{WAIT} \rightarrow \text{CHECK} \rightarrow \begin{cases} \text{ROUND\_COMPLETE} \rightarrow \text{SHOW} & \text{(acerto)} \\ \text{GAME\_OVER} & \text{(erro)} \end{cases}$$

Esta estrutura, que foi detalhadamente representada no diagrama de estados apresentado anteriormente, pode ser diretamente implementada no seguinte código em C:

```c
/* Implementa o da m quina de estados principal */
    switch (game_state) {
        /* Estado 1: Exibe a sequłncia para o jogador */
        case STATE_SHOW_SEQUENCE:
            show_sequence();      /* Acende LEDs na sequłncia*/
            update_display();     /* Atualiza o display */
            break;

        /* Estado 2: Aguarda entrada do jogador */
        case STATE_WAIT_INPUT:
            /* Leitura da p o s i  o do joystick */
            joystick_state_t state = joystickPi_read();

            /* Mapeia posicao X para cores */
            if (state.x < 1365) {
                selected_color = GREEN;
            }
            else if (state.x < 2730) {
                selected_color = BLUE;
            }
            else {
                selected_color = RED;
            }

            light_up_matrix(selected_color);

            /* Transicao ao pressionar bot o B */
            if (button_b_pressed) {
```

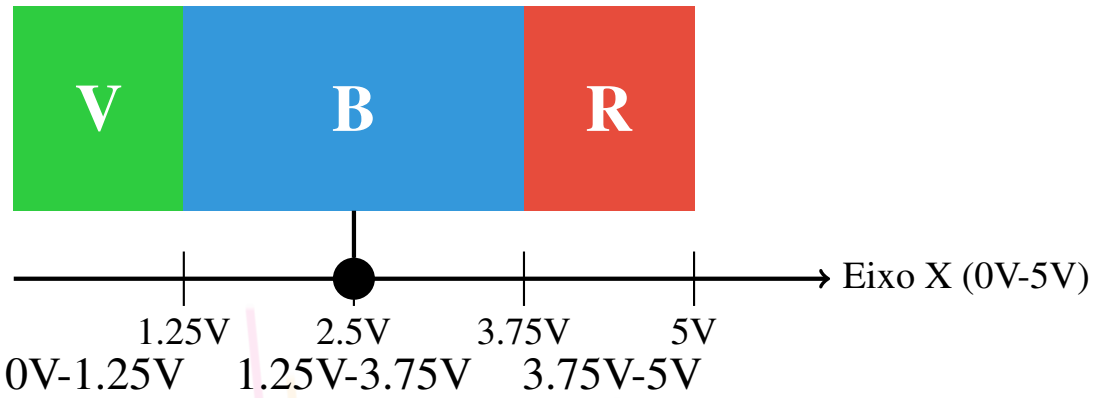
```
                button_b_pressed = false;
                game_state = STATE_CHECK_INPUT;
            }
            break;

        /* Estado 3: Verifica escolha do jogador */
        case STATE_CHECK_INPUT:
            check_choice();
            break;

        /* Estado 4: Rodada completada */
        case STATE_ROUND_COMPLETE:
            show_round_message();
            show_white_matrix();
            sleep_ms(1000);

            /* Prepara pr xima rodada */
            sequence_length++;
            current_step = 0;
            round_number++;
            game_state = STATE_SHOW_SEQUENCE;
            generate_sequence();
            update_display();
            break;

        /* Estado 5: Fim de jogo */
        case STATE_GAME_OVER:
            if (!game_over_shown) {
                MatrizRGBPI_displayStringWithScroll("Game Over",
                    ↪  150, COLOR_RED);
                show_game_over_display();
                game_over_shown = true;
            }
            break;
    }
```

Listing 3: Maquina de estados do jogo Genius

Para a seleção das cores durante o jogo, foi implementado um sistema intuitivo utilizando o joystick como periférico de entrada. O jogador movimenta o eixo horizontal (X) do joystick para escolher entre as três cores disponíveis, conforme o seguinte mapeamento (faixa de 0V a 5V):

Seleção de Cores

(Faixa 0V-5V no Eixo X)



4 Conclusão

O desenvolvimento do jogo **GENIUS** representou uma experiência técnica e acadêmica enriquecedora para os estudantes envolvidos. Durante a implementação, foi necessário utilizar e adaptar diversas bibliotecas de terceiros para atender aos requisitos do projeto. A biblioteca `JoystickPi`, originalmente desenvolvida para leitura básica de joysticks, foi significativamente modificada para incluir tratamento de ruído digital, calibração dinâmica de sensibilidade e um sistema de resposta tátil mais preciso. Da mesma forma, a biblioteca `MatrizRGBPi`, responsável pelo controle da matriz de LEDs, passou por otimizações que permitiram a implementação de efeitos de transição suave entre cores e o sistema de scroll para mensagens.

Além dessas adaptações, o projeto integrou a biblioteca `SSD1306` para controle do display OLED, que foi estendida com novas funções de renderização de texto e gráficos específicos para a interface do jogo. Todas essas modificações foram documentadas e compartilhadas com os mantenedores originais das bibliotecas, contribuindo para a comunidade de desenvolvimento embarcado. O processo de adaptação dessas bibliotecas proporcionou um aprendizado valioso sobre integração de sistemas, gestão de dependências e trabalho colaborativo em projetos de código aberto.

A experiência com o desenvolvimento do GENIUS demonstrou na prática como a engenharia pode transformar conceitos teóricos em aplicações interativas e divertidas. O jogo não apenas cumpriu seu objetivo de entretenimento, mas também serviu como plataforma para o aprimoramento de habilidades técnicas em programação embarcada, tratamento de sinais analógicos e desenvolvimento de interfaces homem-máquina. As lições aprendidas com a modificação e integração das bibliotecas de terceiros certamente serão úteis em futuros projetos profissionais dos estudantes envolvidos.

Ademais o desenvolvimento do jogo **GENIUS** representou uma experiência enriquecedora, permitindo a aplicação prática de conceitos técnicos adquiridos ao longo do curso, além de estimular o trabalho em equipe, a criatividade e a capacidade de resolução de problemas. A partir da ideia inicial, foi possível criar uma versão funcional e envolvente do jogo, proporcionando um desafio tanto do ponto de vista técnico quanto no que diz respeito à jogabilidade.

Durante o processo, foram exploradas diversas tecnologias de programação, bem como estratégias de design e desenvolvimento de interfaces. O uso de algoritmos para processamento de entrada do jogador, gerenciamento de sequência de

cores e o aumento progressivo da dificuldade ao longo do jogo foram elementos cruciais para tornar o jogo desafiador e interessante.

A implementação do **GENIUS** também possibilitou o aprimoramento das habilidades em desenvolvimento de software, especialmente no que diz respeito à criação de sistemas interativos e ao gerenciamento de estados de jogo. Além disso, o feedback obtido ao testar o jogo com diferentes usuários foi valioso para ajustar a dinâmica do jogo, garantindo que ele fosse acessível e divertido.

Em termos de aprendizado, o projeto contribuiu significativamente para o desenvolvimento de competências técnicas e interpessoais, proporcionando uma melhor compreensão das etapas de concepção, desenvolvimento e testes de um produto. Este projeto também reforçou a importância de trabalhar em equipe, de realizar um planejamento adequado e de considerar a experiência do usuário, elementos fundamentais em qualquer projeto de engenharia.

Por fim, o jogo **GENIUS** não é apenas um marco importante para os estudantes envolvidos, mas também uma representação do potencial da engenharia aplicada ao entretenimento e ao desenvolvimento de jogos. Este projeto abre portas para futuras iniciativas no campo dos jogos interativos e traz lições valiosas para os próximos desafios.

Referências

[1] Raspberry Pi Foundation. *Raspberry Pi Pico C/C++ SDK*. Raspberry Pi Trading Ltd, 2021. Documentao oficial do SDK para Raspberry Pi Pico.

[2] BitDogLab. Bitdoglab-c: C/c++ libraries for embedded systems. `https://github.com/BitDogLab/BitDogLab-C/tree/main`, 2023. Acesso em: 29 mar. 2025.