Relatório Técnico

Trabalho Prático — Técnicas de Programação para Sistemas Embarcados II

Implementação de Driver Linux para Dispositivo USB Genérico usando Raspberry Pi Pico

15 de fevereiro de 2026

1 Introdução

A integração entre sistemas operacionais e dispositivos embarcados é um dos pilares do desenvolvimento de sistemas computacionais modernos. Em particular, a comunicação via USB permite que dispositivos externos sejam integrados de forma padronizada ao kernel do sistema, possibilitando a criação de interfaces de alto nível acessíveis por aplicações de usuário.

Este trabalho apresenta o desenvolvimento completo de um sistema composto por:

- Um firmware embarcado em microcontrolador;
- Um driver implementado no kernel Linux;
- Uma interface de comunicação acessível em *user space*.

O dispositivo utilizado foi o **RP2040**, presente na placa Raspberry Pi Pico, operando como um dispositivo USB genérico utilizando *endpoints vendor-specific*. O objetivo principal foi implementar um fluxo completo de comunicação USB capaz de transmitir comandos binários desde uma aplicação em espaço de usuário até o hardware embarcado, permitindo o controle do LED *onboard* da placa.

Além do aspecto funcional, o projeto busca demonstrar na prática conceitos fundamentais como:

- Arquitetura em camadas de sistemas computacionais;
- Comunicação entre *user space* e *kernel space*;
- Desenvolvimento de drivers no Linux;
- Programação de firmware com pilha USB.

2 Arquitetura do Sistema

O sistema foi estruturado em três camadas bem definidas, cada uma com responsabilidades específicas.

2.1 Camadas da Arquitetura

1. **Camada de Aplicação (User Space):** Responsável por gerar comandos e enviá-los ao dispositivo por meio de operações de escrita em arquivo de dispositivo.
2. **Camada de Driver (Kernel Space):** Responsável por intermediar a comunicação entre o sistema operacional e o hardware USB, encapsulando detalhes de baixo nível.
3. **Camada de Dispositivo (Firmware):** Executada no microcontrolador, interpreta os comandos recebidos e executa ações físicas no hardware.

2.2 Fluxo de Comunicação Detalhado

O fluxo evidencia a abstração fornecida pelo kernel Linux, onde o hardware é acessado por meio de um modelo de arquivos:

Aplicação → `syscall write()`
↓
VFS (Virtual File System)
↓
Driver USB (`file_operations`)
↓
USB Core do Linux
↓
Controlador USB Host
↓
Dispositivo USB (RP2040)
↓
TinyUSB Stack
↓
Lógica da Aplicação Embarcada

3 Implementação do Driver Linux

3.1 Registro do Driver no Kernel

O driver foi implementado como um módulo carregável, registrando uma estrutura `usb_driver` contendo a tabela de IDs suportados, função de *probe* e função de *disconnect*. A função *probe* é executada quando o dispositivo com VID/PID correspondente é conectado, sendo responsável por:

- Obter a interface USB;
- Localizar *endpoints*;
- Registrar o dispositivo de caractere.

3.2 Identificação do Dispositivo

O dispositivo é identificado pelos seguintes parâmetros:

- **VID:** `0xCAFE`
- **PID:** `0x4001`

Esses valores são definidos no firmware e utilizados pelo kernel para associar o dispositivo ao driver correto.

3.3 Criação do Dispositivo de Caractere

Após a detecção, o driver registra um dispositivo de caractere e cria a entrada em:

`/dev/pico_usb`

Isso permite que qualquer aplicação interaja com o hardware usando chamadas padrão do sistema (`open`, `read`, `write`).

3.4 Implementação da Escrita

A função `write()` constitui o núcleo da comunicação. Seu funcionamento interno segue os passos:

1. Recebe ponteiro de buffer do *user space*;
2. Valida tamanho da mensagem;
3. Aloca memória no kernel;
4. Copia dados com proteção de acesso;
5. Envia via *endpoint bulk OUT* através da função `usb_bulk_msg()`;
6. Retorna bytes transmitidos.

3.5 Depuração no Kernel

Mensagens foram inseridas com `printk()`, permitindo acompanhar o funcionamento do driver, facilitando a validação do protocolo, diagnóstico de falhas e verificação do fluxo de dados.

4 Firmware Embarcado

4.1 Pilha USB

O firmware utiliza a biblioteca **TinyUSB**, que abstrai o funcionamento do controlador USB do RP2040. O dispositivo foi configurado como *Vendor Class*, permitindo um protocolo proprietário simples.

4.2 Estrutura do Firmware

O firmware é dividido em dois contextos principais:

- **Contexto de Interrupção USB:** Responsável por receber dados e colocá-los em buffer.
- **Contexto de Aplicação:** Responsável por processar comandos e controlar o hardware.

4.3 Protocolo de Comunicação

Foi projetado um protocolo binário minimalista para reduzir *overhead* e simplificar o *parsing*.

Código	Operação	Parâmetros
`0x01`	Ligar LED	—
`0x02`	Desligar LED	—
`0x03`	Piscar LED	número de ciclos

Tabela 1: Protocolo de comandos do firmware.

4.4 Processamento de Comandos

Os comandos são inseridos em uma fila circular e processados no *loop* principal, garantindo o desacoplamento entre a USB e a lógica, execução determinística e ausência de bloqueios.

5 Aplicação em Espaço de Usuário

A interação inicial foi feita via terminal, demonstrando a transparência da interface:

```
1 printf "\x01\x00" | sudo tee /dev/pico_usb
```

Esse modelo reforça o conceito Unix de que dispositivos são tratados como arquivos.

6 Testes e Validação

- **Testes de Conectividade:** Reconhecimento automático e carregamento do driver.
- **Testes de Comunicação:** Envio de comandos e verificação de *logs* (`dmesg`).
- **Testes de Hardware:** Resposta imediata do LED e execução de piscadas.

7 Discussão

O projeto demonstra como o modelo de drivers do Linux abstrai a complexidade do hardware. A divisão em camadas garante modularidade e o uso de um protocolo simples reduz a complexidade sem comprometer a funcionalidade.

8 Conclusão

O trabalho atingiu os objetivos propostos, integrando firmware, driver Linux e interface de usuário. Proporcionou experiência prática em comunicação USB e interação entre diferentes níveis do sistema.

9 Trabalhos Futuros

- Implementação de comunicação bidirecional (*endpoint IN*);
- Criação de biblioteca de usuário;
- Medição de latência e *throughput*.