Desenvolvimento de Driver para um Display LCD de comunicação I^{2}C em Sistema Embarcado

Matheus Luiz Anderle de Souza
Universidade Federal de Santa Catarina
matheusanderle@gmail.com

Paulo Victor Duarte
Universidade Federal de Santa Catarina
paulovictor237@gmail.com

I. Introdução

Um dos tópicos abordados na disciplina de Sistemas Operacionais é relacionado aos dispositivos de entrada e saída. Durante as aulas, vimos que toda vez que um usuário conecta um dispositivo em sua máquina, uma camada abstrata de código é responsável por fazer a comunicação entre o hardware do dispositivo e o sistema operacional da máquina, facilitando a utilização das funções desse dispositivo pelo usuário. Esse tipo de programa é chamado de driver.

Um dos exemplos mais comuns desse tipo de interação é a utilização de um teclado. Quando o teclado é conectado ao computador, o driver é responsável por inicializar as funções do dispositivo e ainda registrá-lo no computador através de um identificador, fazendo com que não seja necessária a instalação desse dispositivo toda vez que ele for conectado ao computador.

Para este trabalho, o objetivo será realizar a integração entre um display LCD de modelo 1602A com um Raspberry Pi 3 Model B+. Isso será feito através do I^{2}C, um barramento comum de baixa velocidade que utiliza um sistema de master e slave para realizar transferências de dados de/para uma aplicação de usuário.

II. Desenvolvimento

Para o desenvolvimento deste trabalho, os seguintes componentes e software foram utilizados:

- Display LCD 16x2 QAPASS 1602a
- Módulo Adaptador I2C PCF8574T
- Raspberry Pi 3 Model B+
- Sistema Operacional: Raspbian
- Versão do Kernel: 4.19.42-v7+

A. Configurações Iniciais

Para iniciar o desenvolvimento de drivers para o Linux, é necessário ter acesso aos arquivos headers do kernel. Para isso, foram realizados uma série de passos listados em [1], como a construção, atualização e configuração do kernel. Vale citar que, para compilar o kernel, o guia referenciado pede para que se utilize o seguinte comando:

```
make -j4 zImage modules dtbs
```

No entanto, o uso dos quatro núcleos acabou travando o Raspberry Pi, então foi reduzido o parâmetro para apenas dois núcleos.

```
make -j2 zImage modules dtbs
```

Por conta dessa alteração, o processo de compilação demorou um tempo maior para ser realizado – em torno de seis horas. Após a finalização dos passos de configuração, prosseguiu-se com o desenvolvimento do trabalho.

B. Conceitos Fundamentais

Para o início do entendimento dos conceitos relacionados à criação de drivers, foi desenvolvido um primeiro driver protótipo que imprimia uma simples mensagem de ”Hello World” no kernel. Esse driver foi baseado e adaptado de aplicações semelhantes feitas por [2] e [3].

```
// hello-kernel.c
#include <linux/init.h>
#include <linux/module.h> // Necessario para todo modulo
#include <linux/kernel.h> // Necessario para KERNEL_INFO

static int __init helloworld_init(void) {
  pr_info("Hello world!\n");
  //printk(KERN_ALERT "Hello, world\n");
return 0;
}
```

```
static void __exit helloworld_exit(void) {
  pr_info("End of the world\n");
  //printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(helloworld_init);
module_exit(helloworld_exit);

MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
MODULE_LICENSE("GPL");
```

Através desse código, foi constatado que os macros *module_init* e *module_exit* são essenciais para o funcionamento desse driver. O macro *module_init* define qual função deverá ser chamada na hora em que o driver for conectado ou chamado, enquanto o macro *module_exit* define qual função será chamada quando o driver for removido do sistema.

Um arquivo makefile foi escrito para executar esse driver, e sua chamada no terminal depende de dois comandos: *insmod* e *rmmod*. Segundo [2], para um módulo ser operacional, ele precisa ser carregado no kernel utilizando *insmod* e o diretório do módulo como argumento, ou então utilizando o comando *modprobe*, que não será detalhado neste relatório. Para a remoção do módulo no sistema, o comando a ser utilizado é o *rmmod*, que deverá ser chamado juntamente com o nome do módulo que o usuário deseja remover.

Com os conceitos fundamentais adquiridos, foi possível iniciar o processo de desenvolvimento de um driver I^2C para o display anteriormente mencionado.

C. I^2C

Criado em 1982 pela Phillips, o I^2C é caracterizado por ser um barramento half-duplex, assíncrono e de relação master e slave com suporte para múltiplos mestres. Ele possui duas ligações, uma intitulada Serial Data (SDA) e a outra Serial Clock (SCL). Segundo [2], a linha de clock é gerenciada pelo mestre e é responsável por sincronizar a transferência de dados, que por sua vez é comandada pelo SDA. Tanto a via de transferência de dados como o sinal de clock são bidirecionais, ou seja, tanto o mestre quanto o escravo podem enviar e receber sinais.

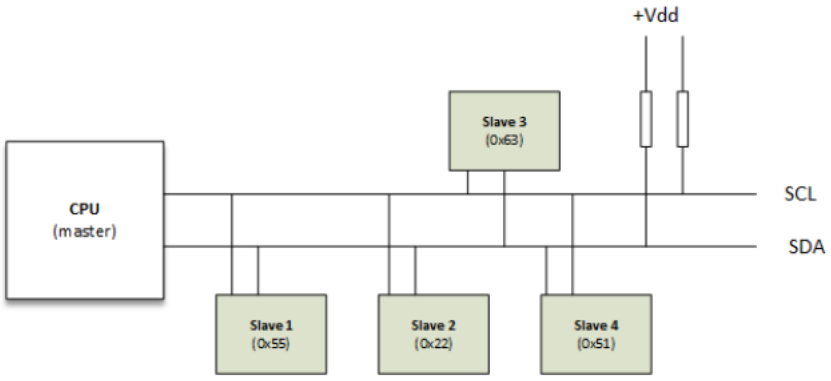

Fig. 1. Arquitetura de um barramento I^2C. [2]

O gerenciamento do driver I^2C é feito através de duas estruturas essenciais, denominadas *i2c_driver* e *i2c_client*. A estrutura *i2c_driver* é responsável pela criação do driver no sistema e o gerenciamento de suas funções, enquanto *i2c_client* é uma representação do próprio dispositivo que será utilizado.

D. Criação das Estruturas

O gerenciamento dos dispositivos I^2C que são utilizados em um sistema operacional é feito por uma estrutura denominada *i2c_driver*. Com base na teoria de [2] e aplicações feitas por [3] e [4], essa estrutura possui um formato genérico similar ao que é exibido à seguir:

```
struct i2c_driver {
/* Standard driver model interfaces */
int (*probe)(struct i2c_client *, const struct i2c_device_id *);
int (*remove)(struct i2c_client *);
/* driver model interfaces that don't relate to enumeration */
struct device_driver driver;
const struct i2c_device_id *id_table;
};
```

Para um melhor entendimento sobre a estrutura, será necessário entender o papel que cada variável e função desempenham. Primeiramente, a cada vez que se deseja inicializar um dispositivo I^2C, a função *probe* deverá ser chamada. Ela é responsável por realizar as rotinas de configuração do dispositivo, como:

- Verificar se o dispositivo a ser conectado é o dispositivo desejado;
- Inicializar o dispositivo;
- Verificar se o dispositivo já está registrado no sistema;
- Realizar as configurações necessárias para que o dispositivo funcione corretamente.

Um dos parâmetros recebidos pela função *probe* é um ponteiro que representa o próprio dispositivo I^2C representado pela estrutura *i2c_client*. Os dados dessa estrutura são preenchidos pelo próprio kernel e servem para registrar características específicas daquele dispositivo. O outro parâmetro é um ponteiro de identificação para o dispositivo em questão.

A função *remove* é responsável por remover o registro do driver no sistema que foi anteriormente realizado pela função *probe*. Como ela recebe o ponteiro para o dispositivo, é possível manipular ou armazenar informações pertinentes ao dispositivo antes de realizar a remoção.

Além disso, a estrutura do driver também possui um ponteiro para uma estrutura de tipo *i2c_device_id*, que é basicamente uma tabela com as identificações dos dispositivos de tipo I^2C. Conforme explorado por [2], essa estrutura possui o seguinte formato:

```
struct i2c_device_id {
char name[I2C_NAME_SIZE];
kernel_ulong_t driver_data; /* Data private to the driver */
};
```

O campo *name* representa uma string com o nome do dispositivo enquanto a variável *driver_data* possui informações privadas do próprio driver. Ainda é necessário o uso da macro *MODULE_DEVICE_TABLE(i2c, idtable)* que receberá o tipo de driver e a tabela que está sendo utilizada para o gerenciamento de dispositivos. A utilização dessa tabela em junção com as outras funções presentes na estrutura *i2c_driver* é essencial para que o kernel entenda qual função *probe* ou *remove* deverá ser chamada dependendo do dispositivo que está sendo utilizado.

Como anteriormente mencionado, a instalação de um determinado driver no sistema operacional depende de rotinas *init* e *exit* que serão chamadas no momento em que o driver é carregado para o kernel. Com isso em mente, a função init desenvolvida chama a macro abaixo, responsável por adicionar o dispositivo no sistema:

```
i2c_add_driver(struct i2c_driver *drv)
```

Como o display utilizado neste trabalho é considerado como um dispositivo de caractere, é importante saber que ele será registrado no kernel utilizando o par de números *major* e *minor*, como visto na literatura de [8]. *Major* representa o tipo de driver (neste caso, o I^2C) que gerencia um determinado número de dispositivos, que por sua vez são representados pelo número *minor*. Esses números podem ser alocados de forma estática ou dinâmica, dependendo do uso pretendido pelo usuário. Caso seja necessário o alocamento estático do número de *major*, deverá ser chamada a seguinte função:

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
```

Na alocação dinâmica, a função que deverá ser utilizada é:

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name);
```

Ela deverá ser seguida da macro *MAJOR(dev_t dev)*, que configura o número desejado. Como neste trabalho só será registrado um único dispositivo I2C no driver, o número *minor* foi configurado como zero.

Além das estruturas previamente mencionadas, o desenvolvimento do driver I^2C também depende da utilização de outras duas estruturas que já são definidas pelo sistema operacional: *struct i2c_adapter* e *struct i2c_board_info*. As duas são responsáveis por definir o barramento do sistema e armazenar informações importantes sobre o dispositivo, respectivamente. Embora não precisem ser alteradas diretamente pelo usuário, elas são de extrema importância para o funcionamento do driver. No caso de *struct i2c_board_info*, um último passo precisou ser realizado e a macro *I2C_BOARD_INFO()*, que preenche informações essenciais da estrutura, precisou ser chamada dentro do método *__initdata*.

Por fim, o driver ainda necessita de métodos para ler, escrever, abrir dados e realizar outras funções básicas para seu funcionamento. Com isso em mente, é necessária a aplicação de mais uma estrutura de tipo *file_operations*, que registra e gerencia esses métodos.

E. Funções do Display

Após o desenvolvimento das funções essenciais de instalação e execução do driver, iniciou-se o processo de criação de funções específicas para a utilização do display. Para isso, decidiu-se utilizar a linguagem C++, que possibilita a aplicação do conceito de classes para auxiliar no desenvolvimento do código.

A classe *lcdd* criada possui os seguintes métodos:

- **lcdd() e ~lcdd()**: construtor e destrutor da classe.
- **lcd_display_string()**: escreve uma string dada pelo usuário na tela.
- **typeFloat() e typeInt()**: escreve um float ou inteiro na tela.
- **back_Light_OFF() e back_Light_ON()**: apaga ou acende a luz de fundo da tela.
- **lcd_clear()**: limpa os dados da tela.

Além disso, ela também utiliza dois métodos privados:

- **home()**: retorna a posição inicial da tela.
- **mover_cursor()**: configura uma nova posição para o cursor.

```
class lcdd{
public:
    lcdd();
    ~lcdd();
    void lcd_display_string(const char* palavra,unsigned int line=1,unsigned int pos=0);
    void typeFloat(float myFloat,unsigned int line=1,unsigned int pos=0);
    void typeInt(int i,unsigned int line=1,unsigned int pos=0);
    void back_Light_OFF(void);
    void back_Light_ON(void);
    void lcd_clear(void);
private:
    void home(void);
    void mover_cursor(unsigned int line,unsigned int pos);
private:
    int fd;
    unsigned int cmd;
    unsigned long arg;
    unsigned long pos_new;
};
```

F. Makefile para o Driver

Como chegou a ser mencionado na explicação do driver ”Hello Kernel” no começo deste trabalho, a instalação de um driver no kernel do Linux depende de um comando essencial, *insmod*. Ele é o responsável por carregar os arquivos e códigos no sistema operacional.

Com isso em mente, foi desenvolvido um makefile adaptado de [3] que além de possuir os comandos principais (*load, unload, run*) também carrega alguns métodos extras como *info, modulos e app*. Esses métodos adicionais podem proporcionar informações sobre os desenvolvedores da aplicação ou também diretamente executar um exemplo de programa.

```
TARGET_MODULE := lcdisplay

BUILDSYSTEM_DIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

obj-m += $(TARGET_MODULE).o

# Run kernel build system to make module
all:
  $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) modules

# Run kernel build system to cleanup in current directory
clean:
  $(MAKE) -C $(BUILDSYSTEM_DIR) M=$(PWD) clean

# Load the module
load:
  insmod ./$(TARGET_MODULE).ko
  sudo chmod 777 load.sh
  sudo ./load.sh

# Unload the module
unload:
  rmmod ./$(TARGET_MODULE).ko
  sudo chmod 777 unload.sh
  sudo ./unload.sh

# Compile and run the application test
run:
  gcc $(APPLICATION).c -o $(APPLICATION) -pthread -Wall
  ./$(APPLICATION)

```

```
print:
  dmesg | tail

info:
  modinfo ./$(TARGET_MODULE).ko

modulos:
  lsmod

app:
  cd APPLICATION; \
    g++ *.cpp -o Executavel -pthread; \
    ./Executavel
```

G. Utilização do Driver

Para poder exemplificar o funcionamento do driver e de suas funções, foi criado uma main utilizando a linguagem C++ que realizasse uma rotina de inicialização e demonstração de cada método criado.

```
#include <iostream>
#include <string>

#include "I2C_LCD_INTERFACE.h"

using namespace std;

int main(int argc, char const *argv[])
{
  cout << "||| APLICACAO INICIADA |||" << endl;
  lcdd displaypv;
  sleep(5);
  displaypv.lcd_clear();
  displaypv.lcd_display_string("APLICACAO",1,4);
  displaypv.lcd_display_string("INICIADA",2,4);
  //--------------------------------
  sleep(5);
  displaypv.lcd_clear();
  displaypv.lcd_display_string("Linha 1",1);
  displaypv.lcd_display_string("Linha 2",2);
  sleep(5);
  displaypv.lcd_clear();
  displaypv.lcd_display_string("Escrever float");
  displaypv.typeFloat(13.45,2,5);
  sleep(5);
  displaypv.lcd_clear();
  displaypv.lcd_display_string("Escrever int");
  displaypv.typeInt(456,2,6);
  sleep(5);
  displaypv.lcd_clear();
  displaypv.lcd_display_string("Desligar luz");
  sleep(5);
  displaypv.back_Light_OFF();
  sleep(5);
  displaypv.back_Light_ON();
  sleep(5);
  displaypv.lcd_clear();
  displaypv.lcd_display_string("Ligar luz");
  sleep(5);
  displaypv.lcd_clear();
  displaypv.lcd_display_string("FIM DO PROGRAMA");
  sleep(5);
  //--------------------------------
  cout << "||| FIM DA APLICACAO |||" << endl;
  return 0;
}
```

III. Resultados

Após o carregamento do driver no sistema operacional, a main foi chamada e a rotina de inicialização foi executada com sucesso, conforme exibido na Figura abaixo.

Fig. 2. Inicialização do Display.

IV. Conclusões

O desenvolvimento do driver I^2C foi essencial para a absorção de conceitos aprendidos na disciplina, especialmente relacionado ao assunto de Entrada/Saída. Desde o desenvolvimento de um primeiro driver extremamente simples como o *hello kernel* até a aplicação para o display 1602A, a base para o desenvolvimento se manteve a mesma, mas o nível de complexidade aumentou. Isso é impulsionado pelo fato de existirem algumas limitações da programação em nível de kernel, que é bem mais específica e possui funcionalidades que ainda não haviam sido vistas em outras disciplinas relacionadas à programação.

A maior dificuldade encontrada foi justamente o entendimento da teoria relacionada à criação de drivers. Observar como construir as estruturas e as funções necessárias e adaptar a literatura para a aplicação específica deste trabalho foi extremamente desafiador, mas satisfatório.

Para trabalhos futuros, seria interessante a criação de uma interface gráfica que pudesse atuar a nível de usuário de forma clara e intuitiva. Essa interface comunicaria diretamente com o código fonte e poderia proporcionar funcionalidades como limpar o display após apertar um botão, além de disponibilizar um campo de texto para que o usuário pudesse digitar o que quisesse. Ferramentas como QT ou até mesmo a biblioteca TKinter, da linguagem Python, poderiam ser utilizadas juntamente com outras bibliotecas de programação que realizariam esse intermédio entre o dispositivo e o usuário.

References

[1] ”https://www.raspberrypi.org/documentation/linux/kernel/README.md”, (Accessed on 06/25/2018).
[2] J. Madieu, *Linux Device Drivers Development*. Packt, 2017.
[3] J. V. Filho, L. C. Souza, ”Driver para um display LCD utilizando comunicação I2C em uma Raspberry Pi”. 2018.
[4] ”Github - lucidm/lcdi2c: Linux kernel module for hd44780 with i2c expander”. https://github.com/lucidm/lcdi2c, (Accessed on 06/25/2018).
[5] M. Opdenacker, ”Introduction to Linux kernel driver programming: i2c drivers”. https://bootlin.com/pub/conferences/2018/elc/opdenacker-kernel-programming-device-model-i2c/kernel-programming-device-model-i2c.pdf, (Accessed on 06/25/2019).
[6] L. Loflin. http://www.bristolwatch.com/rpi/code/i2clcd.txt, (Accessed on 06/25/2019).
[7] R. C. Ganiga, ”Writing I2C Clients in Linux”. https://opensourceforu.com/2015/01/writing-i2c-clients-in-linux/, (Accessed on 06/25/2019).
[8] V. Maciel, ”Linux Device Drivers - Diferenças entre Drivers de Plataforma e de Dispositivo”. https://www.embarcados.com.br/linux-device-drivers/, (Accessed on 06/25/2018).
[9] D. Molloy, ”Writing a Linux Kernel Module — Part 2: A Character Device”. http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/, (Accessed on 06/25/2019).
[10] SparkFun, ”Raspberry Pi SPI and I2C Tutorial”. https://learn.sparkfun.com/tutorials/raspberry-pi-spi-and-i2c-tutorial/all, (Accessed on 06/25/2019).
[11] ”Como usar um display LCD I2C com Raspberry Pi”. https://www.arduinoecia.com.br/2016/12/como-usar-display-lcd-i2c-raspberry-pi.html, (Accessed on 06/25/2019).