Driver para um display LCD utilizando comunicação I^{2}C em uma Raspberry Pi

Lucas de Camargo Souza e Jesuino Vieira Filho
UFSC, Centro Tecnológico de Joinville

I. Introdução

A disciplina de Sistemas Operacionais aborda o funcionamento de dispositivos de entrada e saída, os quais necessitam de uma camada de software especial dentro do sistema, que fornece ao usuário uma forma simplificada de acessar o equipamento. Como exemplo desta abstração, considera-se uma impressora, em que o usuário não deve se preocupar com o funcionamento do processo de impressão. A camada de software em questão é chamada de *drivers de dispositivos*, que é essencial para o funcionamento de um sistema operacional.
O objetivo deste trabalho foi utilizar os conceitos aprendidos na disciplina para criar um módulo de extensão para um sistema operacional de um dispositivo ARM, como o Raspberry Pi. O módulo escolhido foi o driver para um display I^{2}C, que caracteriza um dispositivo do tipo caractere. Tal dispositivo é caracterizado por transferir dados para/de uma aplicação de usuário, se comportando como *pipes* ou portas seriais. Tais dispositivos possuem, ao menos, chamadas de sistema para abrir, fechar, escrever e ler do dispositivo em um fluxo de dados feito em caractere por caractere.
Após a implementação do módulo, foi possível executar o driver em um programa em nível de usuário e escrever uma *string* no display.

II. Desenvolvimento

Abaixo estão listados os dispositivos utilizados e informações de software:

- Raspberry Pi 3+
- Sistema Operacional Raspbian GNU/Linux 9 (stretch)
- Versão do Kernel: 4.14.79-v7+
- Display LCD 16x2 QAPASS 1602a
- Módulo Adaptador I^{2}C PCF8574T

É importante ter em mente que o módulo adaptador I^{2}C geralmente possui um potenciômetro responsável por ajustar o nível de luminosidade do display – como o de cor azul na figura 2. Caso este potenciômetro não esteja ajustado corretamente, o display pode aparentar não estar apresentando saída.

A. Configurando o Raspbian

Seguindo os passos de [1], para compilar um driver de dispositivo no Linux necessita-se dos arquivos *header* do kernel. Estes arquivos não estão disponíveis nas versões padrões do sistema Raspbian, porém há uma forma de obtê-los: é necessário compilar e instalar um novo kernel. As informações de como isso pode ser feito estão documentadas no próprio site do Raspberry Pi[1]. A outra etapa de configuração do Raspberry é habilitar a comunicação I^{2}C do dispositivo, que geralmente vem desabilitada. O site didático do SparkFun possui um passo a passo de como isso pode ser feito utilizando o comando *raspi-config*, como feito na configuração do sistema operacional deste trabalho[2]. Após configurar o Raspberry devidamente, iniciou-se a etapa de implementação do driver.

B. Conceitos de Driver

A implementação de um driver para um sistema Linux exige, no mínimo, duas funções de extrema importância:

```
1 int init_module(void);
2 void cleanup_module(void);
```

A primeira é responsável por inicializar o driver e a segunda é utilizada para desmontar o módulo do sistema operacional. O entendimento destes dois métodos serviu como ponto de partida para a implementação de um primeiro driver, chamado de "*Hello Kernel*":

[1]https://www.raspberrypi.org/documentation/linux/kernel/building.md
[2]https://learn.sparkfun.com/tutorials/raspberry-pi-spi-and-i2c-tutorial/all#i2c-on-pi

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

int init_module(void)
{
  printk(KERN_INFO "Hello Kernel!\n");

  return 0;
}

void cleanup_module(void)
{
  printk(KERN_INFO "Goodbye Kernel!\n");
}
```

Para compilar o módulo, foi necessário utilizar um arquivo Makefile:

```
# Makefile
obj-m += hello-kernel.o

all:
  make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:
  make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

De acordo com [2], cada módulo é feito de um código objeto que pode ser dinamicamente conectado com o kernel em execução através da chamada *insmod* e pode ser desconectado utilizando *rmmod*. Logo, para carregar o módulo no kernel, é necessário executar a sequência de comandos "`$ make; sudo insmod hello-kernel`" e, para descarregar, executa-se "`$ sudo rmmod hello-kernel`". As saídas podem ser visualizadas executando a chamada "`$ dmesg`" no terminal, que gera uma saída com todos as últimas mensagens geradas pelos módulos do kernel. Isto é um resultado da chamada de `printk()`, uma função similar ao conhecido `printf()`, porém em baixo nível.

Quando o adaptador I^2C é registrado usando a API *i2c_register_adapter*, ele procura por dispositivos que possuem o mesmo número de barramento que o adaptador e um novo objeto *i2c_client* é criado usando a API *i2c_new_device*. Esta última cria uma nova estrutura *i2c_client* e seus campos são inicializados com a estrutura *i2c_board_info*. Durante o registro, o kernel corresponde ao nome de todos os drivers I^2C com o nome do cliente I^2C criado. Se qualquer nome corresponder ao cliente, a rotina de análise do driver (*probe*) será chamada.

Durante a rotina de análise, verifica-se que o dispositivo representado por *i2c_client* passado para o driver é o dispositivo real que o driver suporta. Isso é feito tentando se comunicar com o dispositivo representado por *i2c_client* usando o endereço presente nesta estrutura. Se isso falhar, a rotina irá retornar um erro de análise informando ao subsistema do driver de dispositivo do Linux, que ambos não são compatíveis. Caso contrário, a estrutura que representa o dispositivo é alocada e inicializada, realizando a primeira transferência de dados para o display. Depois que a rotina de análise é chamada e toda a configuração necessária é feita, o dispositivo fica ativo e o usuário pode realizar as devidas chamadas de sistema para utilizar a aplicação.

Uma vez sucedida a implementação do programa simples acima, pode-se dar continuação à implementação de um driver para um display I^2C.

C. Registro do Dispositivo de Caractere

Os dispositivos de caracteres são acessados por meio de nomes no sistema de arquivos. Eles ficam localizados no diretório */dev* e são identificados por um "c"na primeira coluna da saída do comando "`$ ls -l /dev`". Se este comando for utilizado, também será visto dois números (separados por vírgula), antes da data da última modificação. Estes números representam o *Major* e o *Minor* do dispositivo em questão.

Tradicionalmente, o número principal, *Major*, identifica o driver associado ao dispositivo. O Linux permite que vários drivers compartilhem números principais, mas a maioria dos dispositivos ainda estão organizados no princípio do "*one-major-one-driver*". O *Minor* é usado pelo kernel para determinar exatamente qual dispositivo está sendo referido.

Dentro do kernel, o tipo *dev_t* (definido em *include/linux/types.h*) é usado para armazenar números de dispositivos. Pode-se utilizar um conjunto de macros encontradas em *include/linux/kdev_t.h* para obter as partes principais ou secundárias de um *dev_t*:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

O kernel consegue alocar o número de *Major* do dispositivo dinamicamente através da chamada da função:

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

Com esta função, o *dev* é um parâmetro de saída que, com a conclusão bem sucedida, manterá o primeiro número no intervalo alocado. O parâmetro *firstminor* deve ser o primeiro *Minor* solicitado a ser usado, que geralmente é 0. O parâmetro *count* representa o número total de números de dispositivos que está sendo solicitado e *name* é o nome do dispositivo que deve ser associado a esse número. Deve-se liberar a região alocada quando o dispositivo não estiver mais em uso com:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Usualmente, a função *unregister_chrdev_region()* é chamada no método *cleanup()*.

A desvantagem da atribuição dinâmica é que não se pode criar os nós do dispositivo antecipadamente, pois o número principal atribuído ao seu módulo pode variar. Para o uso normal do driver, isso dificilmente é um problema, porque uma vez que o número tenha sido atribuído, pode ser obtido em */proc/devices*.

D. I^2C

Proposto e desenvolvido pela PHILIPS (atual NXP) como solução aos requisitos de interconexão de CIs em uma placa PCB, o I^2Cé um protocolo de comunicação em que os dados são transferidos serialmente e cada mensagem possui o tamanho de 8 bits. Este modelo é caracterizado como mestre/escravo, em que há um ou mais dispositivos (controladores), que possuem controle unidirecional sobre outros dispositivos, como sensores e atuadores. O barramento consiste em duas vias de comunicação: *Serial Data Line (SDA)*, utilizada para envio de dados, e *Serial Clock Line (SCL)*, para envio de pulsos de clock. Ambas as vias são bidirecionais, isto é, podem transmitir e receber sinais em ambas as direções,simultaneamente ou não. Ainda, cada uma das linhas está conectada a uma fonte de alimentação via um resistor de pull-up. A Figura 1 exemplifica uma conexão I^2C.

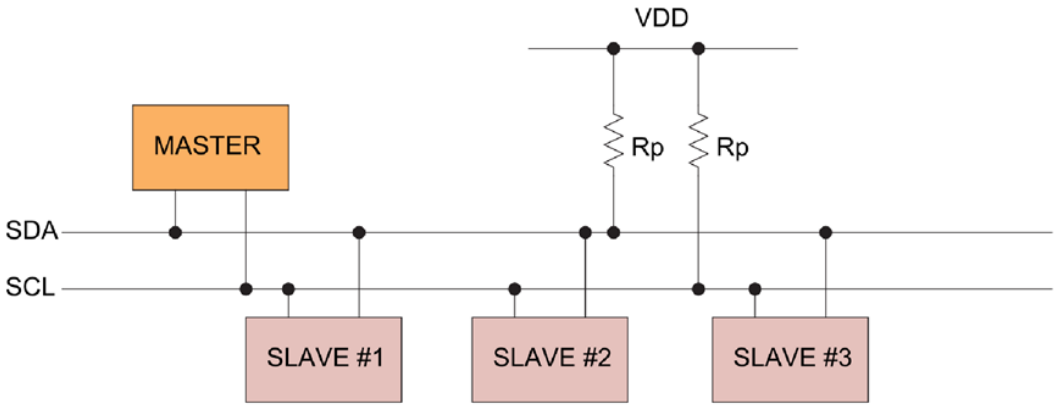

Figura 1. Barramento do protocolo de comunicação I^2C.

Para que os dispositivos possam ser encontrados na rede pelo mestre, cada um deles possui seu endereço próprio, geralmente pré-definido pelo fabricante – alguns dispositivos disponibilizam um mecanismo de escolha do endereço.

E. Implementando o Driver

De acordo com [3], o Linux o possui um subsistema I^2C, caracterizado por uma interface pela qual o sistema operacional pode interagir com os dispositivos conectados no barramento I^2C. A sua implementação é feita de forma que o sistema Linux será sempre o dispositivo mestre. Este subsistema é de extrema importância durante a implementação de um driver para um dispositivo I^2C, consistindo-se das seguintes estruturas (disponíveis em *include/linux/i2c.h*):

- **`struct i2c_adapter`**: define o barramento I^2C utilizado pelo sistema, podendo haver mais que um. Cada barramento é representado por um número (*bus number*).
- **`struct i2c_client`**: define cada dispositivo conectado ao barramento I^2C. O campo "*address*"consiste do endereço do dispositivo na rede e é de extrema importância, pois é utilizado pelo driver para comunicar com o dispositivo. Ainda, o campo "*adapter*"recebe a estrutura que representa o barramento em que o dispositivo foi conectado.
- **`struct i2c_driver`**: para cada dispositivo no sistema operacional, deve haver um driver que o controla. Esta estrutura representa o driver de um dispositivo I^2C e possui dois campos de alta importância: *driver.name*, que define o nome do driver que é utilizado para conectar com o dispositivo na rede, e *probe*, que é um ponteiro de função para um método do driver que é executado quando o dispositivo e o driver forem ambos encontrados no sistema.
- **`struct i2c_board_info`**: esta estrutura é o que realmente define a estrutura `i2c_client`, pois ela contém as informações da placa do dispositivo e os campos deste objeto são copiados para a estrutura `i2c_client`. Possui dois campos importantes: "*type*", que contém o nome do dispositivo para qual a estrutura é definida, e "*addr*", que contém o endereço do dispositivo na rede.

O módulo deste trabalho foi nomeado de *lcdisplay*. A implementação do módulo proposto foi feita baseando-se no driver disponível em [4]. Desta forma, a declaração do driver na estrutura *i2c_driver* foi feita da seguinte forma:

```
static struct i2c_driver lcdriver = {
  .driver = {
    .owner  = THIS_MODULE,
    .name = lcdisplay,
  },
    .probe    = lcdisplay_probe,
    .remove   = lcdisplay_remove,
    .id_table = lcdisplay_id,
};
```

Aqui, a macro `THIS_MODULE`, declarada em *linux/module.h*, será convertida para um ponteiro apontando para a estrutura do módulo que corresponde ao módulo atual, ou seja, do driver do display. Em seguida, o driver precisou ser registrado no subsistema I^{2}C do Linux, o que é feito na execução do método de inicialização do driver, aqui referenciado como:

```
static int __init lcdisplay_init(void)
```

dentro desta função, é feita a chamada de:

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

que recebe como parâmetro a estrutura declarada acima. Este método irá registrar o driver passado como argumento e combinar o nome do driver com todos os nomes em *i2c_client*. Se houver combinação com os nomes, o método `lcdisplay_probe` do driver será chamado e a estrutura *i2c_client* será passada como argumento para o método *probe*.

Além dos métodos comentados até aqui, ainda são necessárias formas para abrir, fechar, escrever e ler dados do dispositivo. Para isso, deve-se implementar uma função para cada um destes procedimentos e registrá-las na estrutura *file_operations*, que recebe uma referência pré-definida para cada uma das funções:

```
static struct file_operations lcdisplay_fops = {
  .owner      = THIS_MODULE,
  .open       = lcdisplay_open,
  .release    = lcdisplay_release,
  .write      = lcdisplay_write,
  .read       = lcdisplay_read,
  .unlocked_ioctl = lcdisplay_ioctl,
};
```

F. Escrevendo no Display

Uma vez que implementadas as funções básicas do driver, responsáveis por inicializá-lo e carregá-lo no subsistema I^{2}C do Linux, pôde-se dar continuidade ao processo de escrita no display. Este processo exige o conhecimento das características e configurações do display, disponíveis no datasheet [5]. Entretanto, a leitura do datasheet é demorada e trabalhosa, logo optou-se por buscar algum exemplo base na internet que possa ser adaptado para o código deste trabalho. O exemplo foi retirado da página [6], que utiliza a biblioteca *WiringPi* para escrever na rede I^{2}C. Ainda, interpretação do exemplo facilitou o entendimento do datasheet. Desta forma, foram retiradas duas informações importantes deste exemplo: como escrever dados no display e como inicializá-lo corretamente. Para escrever os dados, é necessário configurar os bits, para que sejam enviados de 8 em 8, escrevê-los na rede I^{2}C e habilitar um *toggle* a cada escrita. Entretanto, ao invés de utilizar as funções da biblioteca WiringPi, é necessário utilizar as funções de driver do subsistema I^{2}C do Linux, já discutido acima e disponível em *include/linux/i2c.h*. Esta biblioteca disponibiliza o método *i2c_smbus_read_byte_data()*, que pode substituir a fução *wiringPiI2CReadReg8()* da biblioteca WiringPi. Ainda, a função *delayMicroseconds()* da WiringPi pode ser substituída por *udelay()* da biblioteca disponível em *include/linux/delay.h*.

Após adaptar os métodos disponíveis no exemplo, foram implementadas funções para os seguintes objetivos:

- **`lcdinit()`**: inicializa o display e faz a configuração inicial, como setar a posição do cursor, o tamanho de dados, número de linhas e o tamanho da fonte;
- **`lcdfinalize()`**: apaga os dados do display e desliga a tela, deixando-o inativo;
- **`lcdclear()`**: limpa os dados da tela e coloca o cursor na primeira posição;
- **`lcdhome()`**: coloca o cursor na primeira posição;
- **`lcdrestart()`**: idem *lcdclear()*;
- **`lcdwrite()`**: escreve um texto no display a partir da posição atual do cursor;
- **`lcdsetbacklight()`**: liga ou desliga a luz de fundo do display.

G. Compilando e Carregando o Driver: Makefile

Depois de implementadas as funções operações do display, pôde-se dar início ao processo de compilar e carregar o driver no kernel. Para compilar o driver, o método é o mesmo que o utilizado no módulo *Hello Kernel*, apresentado inicialmente. Para

inserir o módulo no kernel, o método é simples, basta executar "`$ insmod ./lcdisplay.ko`" e então o módulo pode ser visto carregado utilizando "`$ lsmod`". Entretanto, apenas estes passos não são suficientes para tornar o driver utilizável, pois é necessário criar um laço de dispositivo no diretório */dev* do sistema operacional e configurar as permissões corretas de acesso para tornar-lo carregável e acessível por uma aplicação.
Para criar o laço, é necessário obter o número *Major* do dispositivo, que é usado para localizar o driver apropriado no kernel. Um método para obter este número seria mantê-lo fixo na implementação, porém isto causaria perda de generalidade no uso do driver. Logo, optou-se por uma forma mais genérica, descrita em [1]. O comando *insmod* faz a chamada da função *init_module* e solicita um número *Major* do kernel. Este é listado imediatamente no arquivo de dispositivo */proc/devices* e pode ser obtido utilizando a ferramenta *AWK* do Linux. Para tanto, utilizou-se um script de shell:

```
# load.sh
#!/bin/sh

device="lcdisplay"

major=`cat /proc/devices | awk "{if(\\$2==\"$device\")print \\$1}"`
minor=0

rm -f /dev/${device} c $major $minor

# Cria o arquivo em /dev para acessar e utilizar o display
mknod /dev/${device} c $major $minor
# Seta as permissoes de uso
chmod 666 /dev/${device}
```

Por fim, uma vez compilado o driver, dois comandos são necessários para carregar o driver apropriadamente no kernel e tornar-lo utilizável:

```
insmod ./lcdisplay.ko
sudo ./load.sh
```

Todos estes métodos foram agregados diretamente ao arquivo Makefile.

H. Utilizando o Driver

Após a chamada do arquivo Makefile acima, o driver pode ser finalmente utilizado. Para isto, fez-se um programa em C a nível de usuário, que abre o arquivo do driver de dispositivo e faz as chamadas de sistema do driver. O uso do driver em um programa em C exige a importação das bibliotecas *fcntel.h*, *unistd.h* e *sys/iocontrol.h* para as chamadas *open()*, *close()* e *ioctl()*, respectivamente. O uso destas chamadas são feitas da seguinte forma:

```
int fd = open("/dev/lcdisplay", O_RDWR);

write(fd, "Test session", 0); sleep(2);

while(g_stop != 'q')
{
  ioctl(fd, LCD_CLEAR, 0);

  write(fd, "Backlight off", 0);        sleep(3);
  ioctl(fd, LCD_BACKLIGHT, 0);        sleep(3);

  write(fd, "? Not anymore", 0);        sleep(3);

  ioctl(fd, LCD_CLEAR, 0);
  write(fd, "Yes, I cleared the display", 0); sleep(3);

  ioctl(fd, LCD_CLEAR, 0);
  write(fd, "ABCDEFGHIJKLMNOPQRSTUVXZ", 0); sleep(3);
  write(fd, "0123456789", 0);         sleep(3);
}

close(fd);
```

III. Resultados

Após implementado o código do driver, utilizou-se os procedimentos apresentados na seção II-G e foi executado um programa de teste em nível de usuário, como descrito em II-H. O driver funcionou corretamente, da forma esperada e sem apresentar

erros. O programa *dmesg* possibilitou visualizar o fluxo de execução das chamadas de funções do driver – geradas pela função *printk()*. Abaixo está mostrada uma foto do Raspberry com o display em funcionamento:

Figura 2. Raspberry Pi com display I^2C em funcionamento.

IV. Conclusão

O trabalho permitiu o desenvolvimento e a familiarização com a programação em nível de kernel, o quê traz ao programador a capacidade de desenvolver módulos para o sistema operacional. Isto ficou evidente ao partir de um módulo simples, como o *Hello Kernel*, para um de maior complexidade, no caso, o driver do display LCD. A passagem da programação em nível de usuário para a programação em nível de kernel mostrou ser um desafio para estudantes da disciplina de sistemas operacionais, tendo em vista que os recursos disponíveis são mais restritos – não possui todas as bibliotecas padrões da linguagem C utilizadas até então. Entretanto, evidenciou-se o contato com os conceitos abordados em sala de aula, como os diferentes níveis de entrada e saída do sistema operacional e a organização dos dispositivos em forma de arquivo na estrutura do sistema UNIX. Uma forma de dar continuidade ao trabalho feito, seria criar uma biblioteca em alto nível para fornecer uma interface mais agradável ao usuário, como uma função *printd()*, evitando que o usuário final realizasse as chamadas de sistema diretamente, no caso, o método de escrita *write()* e o método *ioctl()*. Ainda, seria possível fornecer opções de personalização da saída do display ao usuário, como alterar o tamanho da fonte, setar o cursor etc.
Finalmente, tanto na prática quanto na teoria, a conclusão do trabalho foi enriquecedora e incorporou aos autores técnicas e competências para a elaboração de programas cada vez mais complexos.

Referências

[1] “Simple i/o device driver for raspberrypi - codeproject,” https://www.codeproject.com/Articles/1032794/Simple-I-O-device-driver-for-RaspberryPi, (Accessed on 11/22/2018).
[2] A. Rubini and J. Corbet, *Linux device drivers*. "O’Reilly Media, Inc.", 2001.
[3] “Writing i2c clients in linux,” https://opensourceforu.com/2015/01/writing-i2c-clients-in-linux/, (Accessed on 11/22/2018).
[4] “Github - lucidm/lcdi2c: Linux kernel module for hd44780 with i2c expander,” https://github.com/lucidm/lcdi2c, (Accessed on 11/23/2018).
[5] “Hd44780u (lcd-ii), (dot matrix liquid crystal display controller/driver),” https://www.sparkfun.com/datasheets/LCD/HD44780.pdf, (Accessed on 11/23/2018).
[6] L. Loflin, “Interface i2c lcd to raspberry pi in c,” http://www.bristolwatch.com/rpi/i2clcd.htm, (Accessed on 11/23/2018).