Universidade Federal do Rio Grande do Sul
Escola de Engenharia
Departamento de Sistemas Elétricos de Automação e Energia
ENG10032 Microcontroladores

Roteiro de Laboratório 12
Drivers

Prof. Walter Fetter Lages

20 de novembro de 2019

1 Objetivo

O objetivo deste laboratório é compreender o mecanismo de funcionamento dos *drivers* de dispositivos e desenvolver um *driver* simples para um dispositivo de caractere.

2 Fundamentação Teórica

Um *driver* de dispositivo é um módulo do *kernel* responsável por fazer a interface entre o sistema operacional e o *hardware* de um determinado dispositivo. Os *drivers* também são responsáveis por fornecer uma interface uniforme para os níveis superiores do *software*, como outras partes do sistema operacional e programas do usuário. Para tanto, existe um conjunto de funções padronizadas que o *driver* deve (ou pode) implementar. Com isto, todos os dispositivos podem ser acessados pelos demais programas da mesma forma, independentemente do *hardware* específico.

Na maioria dos sistemas operacionais existem dois tipos de *drivers*: *drivers* para dispositivos de caractere e *drivers* para dispositivos de bloco. Dispositivos de caractere são dispositivos onde a transferência de dados é feita com base em caracteres, como teclados, portas seriais, portas de impressora, etc. Dispositivos de blocos são dispositivos onde os dados são transferidos em blocos, como discos e fitas magnéticos ou ópticos. Obviamente, como o método de acesso é diferente, as funções que devem ser implementadas por cada tipo de *driver* são diferentes.

A tarefa de desenvolver um *driver* consiste basicamente em mapear as funcionalidades suportadas pelo *hardware* nas funções padronizadas que o *driver* deve implementar.

No Un*x os dispositivos são acessados pelos programas do usuário através de pseudo-arquivos, usualmente no diretório `/dev`. Assim, as mesmas funções utilizadas para acessar arquivos podem ser utilizadas para acessar o *hardware*.

A associação entre o nome do pseudo-arquivo e o *driver* do dispositivo é feita através de dois números denominados *major* e *minor*. Cada pseudo-arquivo que representa um dispositivo possui associado um *major* e um *minor*. Similarmente, cada *driver* possui associado um *major*. O *major* identifica o tipo de dispositivo suportado pelo *driver*, enquanto o *minor* identifica, qual dos diversos dispositivos suportados pelo mesmo driver está efetivamente sendo acessado.

Por exemplo, as 3 portas de impressora de um PC são representadas pelos pseudo-arquivos `/dev/lp0`, `/dev/lp1`, `/dev/lp2`. Todos eles possuem o *major* 6, mas cada um possui um *minor* diferente, 0, 1 e 2, respectivamente. O *driver* que suporta as impressoras está associado ao *major* 6. Ao serem chamadas, as funções do *driver* recebem como parâmetro o *minor* referente a qual impressora está sendo acessada.

2.1 Criação de Nodos no `/dev`

2.1.1 Criação de Nodos Manualmente

A forma tradicional de criação dos nodos no diretório `/dev` é manualmente, através do comando:

```
mknod [OPTION]... NAME TYPE [MAJOR MINOR]
```

onde `OPTION` é `-m MODE` ou `--mode=MODE`, sendo `MODE` as permissões de acesso ao nodo, `NAME` é o nome do nodo, `TYPE` é o tipo do nodo: `b`, para dispositivo de bloco, `c` ou `u` para dispositivo de caractere e `p`, para pipe.

Por exemplo, o dispositivo para o *mixer* da placa de som seria criado com o comando

```
mknod -m 0660 /dev/mixer c 14 0
```

Embora simples, este método não funciona bem com a maioria dos sistemas atuais, já que tornou-se comum o uso de um sistema de arquivos baseado em RAM para o `/dev`. Com isso, cada vez que o computador é reinicializado, o conteúdo do `/dev` é perdido. O nodo teria que ser criado novamente cada vez que o sistema é reinicializado.

2.1.2 Criação dos Nodos pelo `udev`

O `udev` é um sistema para criar os nodos de dispositivos no `/dev` de forma automática. Também permite que quando um nodo é criado sejam criados *links*

simbólicos para ele, suas permissões sejam ajustadas e/ou que seja executado um determinado programa.

A operação do `udev` é baseada em uma série de regras, que especificam o que deve ocorrer quando um determinado *driver* é carregado ou quando um determinado nodo é criado. Os arquivos com essas regras são armazenados nos diretórios `/lib/udev/rules.d` para as regras padrão e `/etc/udev/rules.d` para as regras personalizadas. Está fora do escopo deste laboratório entender toda a sintaxe das regras do `udev`. Serão apenas utilizadas algumas construções necessárias para criar nodos para os dispositivos.

O `udev` funciona com o auxílio de um *daemon* denominado `udevd`, que monitora os eventos gerados pelo *kernel* e executa as regras apropriadas.

2.1.3 Criação do Nodo pelo Próprio *Driver*

Outra forma, mais simples e prática para criar os nodos no `/dev` é fazer com que o próprio módulo que implementa o *driver* para o dispositivo crie o nodo ao ser carregado e remova-o ao ser descarregado. Com isso, evita-se a necessidade de ter um *script* para fazer isso e eventuais inconsistências entre os valores de *major* e *minor* utilizados pelo *driver* e pelo *script*.

Versões mais recentes do *kernel* podem alocar o *major* para cada *driver* dinamicamente, de forma que não é mais necessário alocar explicitamente os *majors*. Por outro lado, complica as coisas para a criação dos nodos de dispositivos, pois o *major* não será sempre o mesmo para cada *driver*. A criação do nodo pelo próprio *driver* resolve este problema.

Para que o *kernel* aloque dinamicamente um *major* para o *driver* basta passar 0 no parâmetro correspondente ao *major* na chamada à função `register_chrdev()`, que retornará o *major* alocado.

Antes de criar um nodo para o dispositivo através do *driver* é conveniente criar uma classe para o dispositivo (ou utilizar uma classe já existente). Esta classe irá gerar uma entrada no diretório `/sys/class` onde serão armazenadas as informações dos dispositivos criados. O nome desta classe também pode ser utilizado nas regras do `udev` através da chave `SUBSYSTEM`. A classe é criada com a macro:

```
#include <linux/device.h>

class_create(owner,name)
```

onde `owner` é tipicamente a constante `THIS_MODULE`. A macro retorna um ponteiro para uma `struct class`, que é utilizada para criar os nodos para os dispositivos.

Tendo-se a classe, pode-se criar os nodos para os dispositivos com a função:

```
#include <linux/device.h>
```

```
struct device *device_create(struct class *cls,
                             struct device *parent,
                             dev_t devt,void *drvdata,
                             const char *fmt, ...);
```

onde `cls` é o ponteiro para a classe, `parent` é o ponteiro para dispositivo pai, se houver algum, `devt` é o descritor do dispositivo, criado a partir da macro `MADEV(major,minor)`, `drvdata` é um ponteiro para dados privados do dispositivo, e `fmt` é uma *string*, no formato utilizado pela função `printf()` com o nome do nodo a ser criado. Os parâmetros adicionais dependem do formato especificado em `fmt`.

Quando não foram mais necessários os nodos dos dispositivos e a classe devem ser removidos com as funções:

```
#include <linux/device.h>

void device_destroy(struct class *cls, dev_t devt);
void class_destroy(struct class *cls);
```

A listagem 1 mostra um módulo que implementa um *driver* que cria seus próprios nodos para os dispositivos suportados.

Listagem 1: Módulo com *driver* que cria os próprios nodos para os dispositivos suportados.

```
#include <linux/module.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/gpio.h>
#include <linux/uaccess.h>

#define DEVICE_NAME "led"
#define CLASS_NAME "eng10032"

static int major=0;
static int minor=0;
static struct class *ledclass=NULL;

MODULE_AUTHOR("Walter Fetter Lages <fetter@ece.ufrgs.br>");
MODULE_DESCRIPTION("LED Driver");
MODULE_SUPPORTED_DEVICE("led");
MODULE_LICENSE("GPL");

module_param(major,int,0);
MODULE_PARM_DESC(major,"major number; default is kernel allocated.");

static ssize_t led_write(struct file *file,const char *buf,
        size_t count,loff_t *ppos)
{
        unsigned char data;
        int error;

        if(count != sizeof(unsigned char)) return -EINVAL;
        if( (error=get_user(data,buf)) ) return error;
        gpio_set_value(7,data & 1);
        return sizeof(unsigned char);
```

```
}

static struct gpio routers[]={
        {46,GPIOF_OUT_INIT_LOW|GPIOF_EXPORT_DIR_FIXED,"IO13 is GPIO"},
        {30,GPIOF_OUT_INIT_LOW|GPIOF_EXPORT_DIR_FIXED,"IO13 is Out"},
        {7,GPIOF_OUT_INIT_HIGH|GPIOF_EXPORT_DIR_FIXED,"IO13"}
};

static int led_open(struct inode *inode,struct file *file)
{
        return gpio_request_array(routers,ARRAY_SIZE(routers));
}

static int led_release(struct inode *inode,struct file *file)
{
        gpio_free_array(routers,ARRAY_SIZE(routers));
        return 0;
}

static struct file_operations led_fops=
{
        .owner=THIS_MODULE,
        .write=led_write,
        .open=led_open,
        .release=led_release
};

int init_module(void)
{
        dev_t ledno;

        if((major=register_chrdev(major,DEVICE_NAME,&led_fops))==-1)
        {
                printk("major %d can't be registered.\n",major);
                return -EIO;
        }
        ledclass=class_create(THIS_MODULE,CLASS_NAME);
        ledno=MKDEV(major,minor);
        device_create(ledclass,NULL,ledno,NULL,DEVICE_NAME "%d",minor);

        return 0;
}

void cleanup_module(void)
{
        dev_t ledno;

        ledno=MKDEV(major,minor);
        device_destroy(ledclass,ledno);
        if(ledclass != NULL) class_destroy(ledclass);
        unregister_chrdev(major,DEVICE_NAME);
}
```

Note que mesmo com o *driver* criando os próprios nodos para os dispositivos, é ainda necessário utilizar o *udev* para configurar as permissões de acesso, proprietário e grupo do nodo criado, e talvez criar *links* simbólicos para ele. A listagem 2 mostra uma regra do `udev` típica para estes casos.

Listagem 2: Arquivo de regras para o `udev`, sendo os nodos criados pelo próprio *driver*.

```
SUBSYSTEM=="eng10032", KERNEL=="led[0-3]", MODE="0620", GROUP="gpio"
```

2.2 Funções Implementadas pelo *Driver*

As funções que podem ser implementadas por um *driver* de dispositivo de caractere estão definidas na estrutura `file_operations`, declarada no arquivo `linux/fs.h`. Para um *driver* simples, as principais funções são equivalentes às disponíveis na biblioteca padrão de C, ou seja:

`int open(struct inode *inode,struct file *file)` Para inicializar o dispositivo.

`int release(struct inode *inode,struct file *file)` Para fechar o dispositivo.

`ssize_t write(struct file *file,const char _user *buf, size_t count,loff_t *ppos)` Para escrever no dispositivo.

`ssize_t read(struct file *file,const char _user *buf, size_t count,loff_t *ppos)` Para ler do dispositivo.

3 Experimentos

Neste laboratório será desenvolvido um *driver* para acionar o LED conectado no pio IO13 da Galileo. Os drivers no Linux são específicos para cada versão do *kernel* e para serem compilados requerem que o respectivo *kernel* esteja configurado e que, pelo menos, os seus arquivos de cabeçalho estejam disponíveis. A princípio, os *drivers* deveriam ser compilados na própria árvore do *kernel*, mas existem *hooks* nos *makefiles* do *kernel* que permitem que *drivers* sejam compilados fora da árvore, como será feito aqui.

Será utilizado o *kernel* 3.8.7, que é o *kernel* que está na imagem do cartão microSD usado no laboratório. No entanto, para compilar módulos para este *kernel* é necessário que o seu código fonte esteja disponível. Para evitar congestionamento na rede com todos os alunos baixando a sua cópia ao mesmo tempo, ele já está instalado em `~/src/lab11/linux-3.8.7`. Obviamente, para compilar *drivers* para a Galileo, também deve-se usar o compilador cruzado.

1. Ajuste as variáveis de ambiente para a arquitetura x86 e o `PATH` do sistema para localizar o compilador cruzado:

```
export ARCH=x86
export DEVKIT=/opt/iot-devkit/devkit-x86
export POKYSDK=$DEVKIT/sysroots/x86_64-pokysdk-linux
export PATH=$PATH:$POKYSDK/usr/bin/i586-poky-linux
export CROSS_COMPILE=i586-poky-linux-
```

2. Crie um *link* para o código-fonte do *kernel* usado no laboratório 11, com os comandos

```
mkdir ~/lab12
cd ~/lab12
ln -s ../lab11/linux-3.8.7
```

3. Copiar a configuração do *kernel* que está executando na Galileo e descompactar no diretório onde está o código-fonte do Linux:

```
cd ~/src/lab12/linux-3.8.7
scp <login@galileo>:/proc/config.gz .
zcat config.gz > .config
rm config.gz
```

   onde `<login@galileo>` é o seu *login* e nome da Galileo em uso.

4. Configurar o *kernel* com o comando:

```
make olddefconfig
```

5. Preparar o *kernel* para compilação de módulos fora da árvore:

```
make modules_prepare
```

6. Na listagem 1 tem-se o código fonte de um *driver* para dispositivo de caracter que permite acionar o LED. O *driver* implementa a função `write()`, que escreve o byte passado como parâmetro na linha de comando na `gpio7`.

7. O `Makefile` para compilar o *driver* (módulo do *kernel*) é mostrado na Listagem 3. Note que este `Makefile` é bem diferente dos demais. Isto ocorre porque, na verdade, não é ele que compila o *driver*, mas sim utiliza *hooks* nos *makefiles* do *kernel*, que são executados para fazer a compilação. Isto tem que ser assim porque um *driver* é um pedaço do *kernel*. Note também que este `Makefile` assume que o código-fonte do *kernel* está no diretório `../linux-3.8.7`. Ajuste o `Makefile` de acordo, ou crie um *link* `../linux-3.8.7` apontando para a localização do código-fonte do *kernel*.

Listagem 3: `Makefile` para compilar um *driver*.

```
MODULEDIR=/lib/modules/`uname -r`/char

TARGET=led.ko
SRCS=$(TARGET:.ko=.c)

KERNEL_SOURCE=/lib/modules/`uname -r`/build
#KERNEL_SOURCE=../linux-3.8.7
EXTRA_CFLAGS+=-D__KERNEL__ -DMODULE

obj-m+=$(TARGET:.ko=.o)

all: $(TARGET) $(UDEV_RULES)

$(TARGET): $(TARGET:.ko=.c)
	$(MAKE) -C $(KERNEL_SOURCE) O=. M=`pwd` modules

clean:
	rm -rf *~ *.bak *.o *.mod.c .*.ko.cmd .*.o.cmd .tmp* \
		Module.symvers modules.order

distclean: clean
	rm -f $(TARGET)


install:
	install -m 0755 -d $(MODULEDIR)
	install -m 0644 $(TARGET) $(MODULEDIR)

uninstall:
	rm -f $(MODULEDIR)/$(TARGET)
```

8. Compile o *driver* com o comando:

```
make
```

9. Instale o arquivo de regras do `udev` em `/etc/udev/rules.d` com o nome `99-eng10032-lab12.rules`.

10. Transfira o binário do *driver* (arquivo `.ko`) para a Galileo e instale-o no diretório `/lib/modules/3.8.7-yocto-standard/char`.

11. Atualize a lista de dependências do *kernel* executando como superusuário o comando:

```
depmod -a
```

12. Faça um *script* de inicialização para carregar o módulo do *kernel* que implementa o *driver* usando o comando `modprobe` e instale-o na Galileo.

13. Reinicialize a Galileo.

14. Verifique se ele foi mesmo carregado com o comando `lsmod`.

15. Verifique que o *major* atribuído para o *driver* através do comando

```
cat /proc/devices | more
```

16. Procure no diretório `/dev` o nome dos dispositivos com o *major* descoberto no ítem anterior.

17. Faça um programa no espaço usuário que acenda e apague o LED dependendo de um parâmetro passado na linha de comando, utilizando o *driver* desenvolvido, ao invés da interface `/sys/class/gpio`.

18. Teste o programa no espaço do usuário.