Embedded Labworks

LINUX DEVICE DRIVERS

Laboratório e exercícios

Embedded Labworks

Índice

Sobre este documento

Este documento contém uma série de atividades de laboratório e exercícios para complementar o conteúdo apresentado no treinamento “Linux device drivers”.

Cada item de laboratório possui uma série de atividades a serem realizadas sob supervisão do instrutor.

Algumas atividades de laboratório podem ter exercícios extras, que podem ser realizados durante o treinamento, conforme a disponibilidade de tempo.

Orientações gerais

Não hesite em perguntar e tirar todas as suas dúvidas, bem como em trocar experiências e sugerir soluções diferentes das apresentadas nas aulas de laboratório. Sua participação pode deixar o conteúdo apresentado mais rico e o treinamento mais interativo.

Se você terminar suas tarefas de laboratório antes de seu colega, ajude-o a investigar os problemas que ele possa estar enfrentando. Quanto mais rápido progredirmos como um grupo, mas chances teremos de ver tópicos adicionais e aprender mais.

Não hesite em reportar problemas ou erros ao instrutor.

Algumas imagens exibidas neste documento tem um significado especial:

Esta imagem representa uma explicação ou orientação que será passada pelo instrutor.

Esta imagem representa uma dica ou sugestão importante para a execução de um exercício proposto.

Convenções

Durante as atividades de laboratório, chamaremos de *host* a máquina de desenvolvimento (PC, desktop, notebook) e *target* o kit de desenvolvimento (hardware).

Comandos que devem ser executados no terminal do Linux no host começam com “$”. Exemplo:

```
$ ls
```

Comandos que devem ser executados no terminal do Linux no target começam com “#”. Exemplo:

```
# ls
```

Comandos que devem ser executados no prompt do bootloader no target começam com “>”. Exemplo:

```
> help
```

Comandos de substituição

Expressões entre os sinais de menor e maior (Ex: <expressão>) devem ser substituídas de acordo com as instruções documentadas nos exercícios, incluindo-se os próprios sinais.

Para facilitar para o aluno, as expressões que devem ser substituídas estarão ressaltadas na cor vermelha, conforme abaixo:

```
$ sudo dd if=arquivo.img of=/dev/<device> bs=1M
```

Operações em bits

Em alguns exercícios durante o treinamento podemos referenciar determinado bit de um registrador da CPU. Por convenção, a contagem dos bits começa em 0. Por exemplo, um registrador de 32 bits vai do bit 0 (menos significativo) ao bit 31 (mais significativo).

Editor de texto

Durante o treinamento, precisaremos de um editor de texto para desenvolver os exercícios de laboratório.

Existem diversos editores de texto para Linux, desde os mais simples até os mais complexos e integrados à uma IDE.

O seu ambiente de desenvolvimento já esta preparado com alguns editores de texto comuns no Linux, incluindo `vim`, `emacs`, `gedit`, `kate` e `kdevelop`.

Use o editor que preferir ou estiver mais acostumado. Se você não tiver experiência com editores de texto em Linux, é sugerido o uso do `kate` ou do `kdevelop`.

Código-fonte

Em alguns exercícios, iremos alterar o código-fonte de um driver para modificar seu comportamento ou implementar uma nova funcionalidade.

Para facilitar para o aluno, em alguns casos as linhas que devem ser alteradas ou incluídas serão exibidas na cor azul, conforme abaixo:

```
static struct file_operations drvled_fops = {
    .owner = THIS_MODULE,
    .write = drvled_write,
    .read = drvled_read,        // linha alterada ou adicionada
};
```

Leitor de PDF

Ao copiar um comando ou trecho de código-fonte deste documento, alguns leitores de PDF podem não copiar o texto corretamente. Ao colar, sempre verifique se o texto copiado está correto.

Tome cuidado também ao copiar os comandos do leitor de PDF, já que alguns leitores inserem automaticamente um caractere de nova linha, fazendo com que o comando seja executado automaticamente quando você colar no terminal.

Respostas dos exercícios

As respostas dos exercícios deste documento estão disponíveis em `/opt/labs/ex/respostas`.

Na dúvida sobre o desenvolvimento de algum exercício, sinta-se livre para consultar as respostas. Mas lembre-se, quanto mais você se esforçar para desenvolver um exercício, melhor será o aprendizado!

Laboratório 1 - Estudando o código-fonte do Linux

Nesta atividade iremos estudar o código-fonte do kernel Linux.

O código-fonte do kernel Linux para o kit de desenvolvimento pode ser obtido no repositório GIT do fabricante em http://git.toradex.com/cgit/linux-toradex.git/.

Para facilitar nosso trabalho, o código-fonte já foi baixado e disponibilizado no ambiente de laboratório do treinamento em `/opt/labs/dl/kernel`.

1.1 O kernel Linux

Descompacte o código-fonte do kernel no diretório de exercícios do treinamento:

```
$ cd /opt/labs/ex/01
$ tar xfv /opt/labs/dl/kernel/linux.tar.bz2
$ cd linux/
```

Com o auxílio do instrutor, estude a organização do código-fonte do kernel Linux.

1.2 A ferramenta LXR

Obs.: Para executar este exercício é necessário ter conexão com a Internet.

Acesse o link abaixo e navegue pelo código-fonte da última versão do kernel.

http://lxr.free-electrons.com/

Procure pela implementação da função `start_kernel()`, responsável pela inicialização do kernel.

Identifique quais são os parâmetros e o retorno da função `alloc_chrdev_region()`.

A constante `THREAD_SIZE` é responsável por armazenar o tamanho do stack de um processo rodando em kernel space. Qual o valor desta constante para a arquitetura ARM?

Laboratório 2 - Conectando e testando o hardware

Nesta atividade iremos conectar o target ao host pela porta serial e configurar uma aplicação de terminal para acessar a porta serial do target.

2.1 Conectando o target

Com o auxílio do instrutor, conecte o host ao target pela porta serial.

2.2 Configurando a conexão serial

Após conectar a porta serial do target ao host, precisamos de uma aplicação de terminal para acessar a porta serial.

Existem algumas aplicações de terminal disponíveis para distribuições Linux, dentre elas temos `minicom`, `picocom`, `putty` e `screen`.

Se você tiver experiência com alguma destas ferramentas, sinta-se livre para configurá-la e executá-la com baud de 115200, 8 bits, sem paridade, 1 stop bit e sem handshake de hardware/software.

Caso não tenha experiência com nenhuma destas ferramentas, utilize a aplicação `picocom` que já está configurada no ambiente de laboratório do treinamento.

Para utilizar o `picocom`, basta executá-lo conforme abaixo:

```
$ picocom
```

Ao final deverá ser exibida a mensagem "*Terminal ready*", indicando que a comunicação com a porta serial está funcionando normalmente.

É aconselhável manter uma janela com o `picocom` em execução durante todo o treinamento. De qualquer forma, caso queira sair do `picocom`, basta pressionar `CTRL-A` e depois `CTRL-X`.

2.3 Testando a conexão

Ligue o target e verifique se você consegue visualizar as mensagens de boot e ter acesso ao terminal de comandos do bootloader.

Laboratório 3 - Compilando o kernel e testando o sistema

Nesta atividade aprenderemos a configurar e compilar o kernel, e testaremos o boot do sistema.

3.1 Preparando o código-fonte:

Vamos preparar o kernel para compilar para ARM.

Entre no diretório do kernel:

```
$ cd /opt/labs/ex/01/linux
```

Abra o arquivo `Makefile` e altere as variáveis `ARCH` e `CROSS_COMPILE`, conforme abaixo:

```
$ gedit Makefile
```

```
# Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
ARCH            ?= arm
CROSS_COMPILE   ?= arm-linux-gnueabi-

# Architecture as present in compile.h
UTS_MACHINE     := $(ARCH)
```

3.2 Configurando o kernel

Configure o kernel para o kit de desenvolvimento:

```
$ make colibri_imx6_defconfig
```

3.3 Compilando o kernel

Compile o kernel:

```
$ make zImage -j4
```

Copie a imagem do kernel para o servidor de TFTP:

```
$ cp arch/arm/boot/zImage /tftpboot/
```

Consulte esta seção sempre que for necessário recompilar o kernel.

3.4 Compilando o device tree

Compile o device tree do kit de desenvolvimento:

```
$ make imx6dl-colibri-ipe.dtb
```

Copie a imagem do device tree para o servidor de TFTP:

```
$ cp arch/arm/boot/dts/imx6dl-colibri-ipe.dtb /tftpboot/
```

Consulte esta seção sempre que for necessário recompilar o device tree.

3.5 Compilando os módulos do kernel

Compile os módulos do kernel:

```
$ make modules -j4
```

Instale os módulos do kernel no rootfs do kit de desenvolvimento:

```
$ sudo make INSTALL_MOD_PATH=/opt/labs/ex/03/rootfs modules_install
```

Consulte esta seção sempre que for necessário recompilar os módulos do kernel.

3.6 Testando o sistema

Durante o processo de boot, o kernel e o device tree serão baixados da rede por TFTP.

O rootfs que utilizaremos no treinamento está disponível em `/opt/labs/ex/03/rootfs/`. No final do boot, o kernel irá montar este sistema de arquivos pela rede via NFS.

Para que este processo funcione, iremos configurar uma rede ponto-a-ponto entre o host e o target. O target deve estar com o IP `192.168.0.2` e o host com o IP `192.168.0.1`.

O bootloader do target já está configurado com o endereço IP `192.168.0.2`. Você precisa apenas configurar o host com o endereço IP `192.168.0.1`. No caso de dúvidas, peça ajuda ao instrutor.

Com a rede configurada, reinicie o target e teste. No final do boot, você deverá ter acesso ao terminal de comandos do Linux no target:

```
Bem vindo ao treinamento de drivers da Embedded Labworks!
labworks login:
```

Para fazer o login, o usuário padrão é `root` (sem senha).

3.7 Habilitando o suporte ao NTFS

Abra o menu de configuração do kernel e habilite o suporte ao sistema de arquivos NTFS de forma estática (integrado à imagem do kernel).

Compile e teste.

Para verificar se o suporte ao sistema de arquivos NTFS foi habilitado corretamente, liste o conteúdo do arquivo `/proc/filesystems`.

Laboratório 4 - O primeiro módulo do kernel

Nesta atividade iremos aprender a desenvolver e trabalhar com módulos do kernel.

4.1 Desenvolvendo o primeiro módulo

Entre no diretório de exercícios do treinamento:

```
$ cd /opt/labs/ex/04
```

Implemente um módulo chamado `labmodule.c`:

```
$ gedit labmodule.c
```

```
#include <linux/module.h>
#include <linux/kernel.h>

#define MODULE_NAME  "labmodule"

static int __init labmodule_init(void)
{
     printk("%s: initialized.\n", MODULE_NAME);
     return 0;
}

static void __exit labmodule_exit(void)
{
     printk("%s: exiting.\n", MODULE_NAME);
}

module_init(labmodule_init);
module_exit(labmodule_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your name here <name@e-mail.com>");
MODULE_DESCRIPTION("Laboratory exercise 4.");
MODULE_VERSION("1.0");
```

Sinta-se livre para substituir as informações da macro `MODULE_AUTHOR()` pelo seu nome e e-mail.

Agora crie um `Makefile` para compilar o módulo:

```
$ gedit Makefile
```

```
KDIR   := /opt/labs/ex/01/linux
ROOTFS := /opt/labs/ex/03/rootfs

obj-m += labmodule.o

module:
	@$(MAKE) -C $(KDIR) M=$(PWD) modules

install:
	@sudo $(MAKE) -C $(KDIR) M=$(PWD) INSTALL_MOD_PATH=$(ROOTFS) modules_install

clean:
	@$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Obs.: As linhas após os targets (`module`, `install`, `clean`) não podem ter espaços! Você precisa usar necessariamente a tecla TAB. Esta é uma característica da sintaxe da ferramenta `make`.

Por fim, compile o módulo e instale no rootfs:

```
$ make
$ make install
```

4.2 Testando o módulo

Acesse o terminal serial do target para testar o módulo criado.

Exiba as informações do módulo:

```
# modinfo labmodule
```

Depois insira o módulo:

```
# modprobe labmodule
```

Verifique se o módulo foi carregado:

```
# lsmod
```

E teste a remoção do módulo:

```
# modprobe -r labmodule
```

Perceba que você consegue visualizar as mensagens do módulo porque está conectado à console do target. Se não estiver conectado à console, você pode exibir as mensagens do kernel com o comando `dmesg`.

```
# dmesg
```

4.3 Passando parâmetros

Altere o módulo para receber um parâmetro chamado `name` do tipo ponteiro para char (`charp`).

Altere a função de inicialização do módulo para exibir o valor do parâmetro passado.

Compile o módulo, carregue no target e teste.

4.4 Adicionando o módulo aos fontes do kernel

Agora vamos adicionar o módulo à árvore do código-fonte do kernel para compilá-lo de forma estática (built-in). Copie o arquivo-fonte do módulo para os fontes do kernel:

```
$ cp /opt/labs/ex/04/labmodule.c /opt/labs/ex/01/linux/drivers/char/
```

Edite o arquivo `Kconfig` e adicione a opção de configuração do módulo no começo do arquivo, logo após a configuração do menu:

```
$ cd /opt/labs/ex/01/linux/drivers/char/
$ gedit Kconfig
```

```
menu "Character devices"

config LABMODULE
        tristate "Embedded Labworks module (ex4)"
        default n
        help
          This option enables Embedded Labworks module from exercise 4.

source "drivers/tty/Kconfig"
```

Edite também o `Makefile`, adicionando no começo do arquivo a linha de compilação do módulo:

```
$ gedit Makefile
```

```
obj-$(CONFIG_LABMODULE)         += labmodule.o
obj-y                           += mem.o random.o
```

Agora entre na configuração do kernel, acesse o menu "*Device Drivers*" → "*Character devices*" e habilite o módulo de forma estática.

Recompile o kernel e teste. Verifique se o módulo foi carregado corretamente, procurando a mensagem de inicialização do módulo no boot do kernel.

```
# dmesg | grep labmodule
```

Laboratório 5 - O primeiro driver

Nesta atividade começaremos o desenvolvimento de um driver para acionar um led conectado a um GPIO do kit de desenvolvimento.

5.1 Alocando o major/minor

Entre no diretório de exercícios do treinamento e abra o esqueleto do driver disponibilizado para esta atividade:

```
$ cd /opt/labs/ex/05
$ gedit drvled.c
```

Inclua o arquivo de cabeçalho `<linux/fs.h>`:

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

Declare uma variável global do tipo `dev_t` para armazenar o device number (major/minor).

```
static dev_t drvled_dev;
```

Por padrão, declare sempre funções e variáveis globais como `static`, a não ser que você pretenda exportar algum símbolo para o kernel.

Aloque o major/minor dinamicamente através da função `alloc_chrdev_region()` na inicialização do driver. Não esqueça de fazer o tratamento do erro, exibindo uma mensagem com a função `pr_err()` e retornando o erro:

```
int result;

if ((result = alloc_chrdev_region(&drvled_dev, 0, 1, DRIVER_NAME))) {
     pr_err("%s: Failed to allocate device number!\n", DRIVER_NAME);
     return result;
}
```

A partir de agora, usaremos as funções da família `pr` (`pr_info()`, `pr_err()`, etc) para imprimir mensagens no kernel. Na seção de depuração do kernel estudaremos os motivos.

Desaloque o major/minor na rotina de limpeza do driver com a função `unregister_chrdev_region()`.

```
unregister_chrdev_region(drvled_dev, 1);
```

Compile o driver e instale no target.

Carregue o driver:

```
# modprobe drvled
```

Liste o arquivo `/proc/devices` e verifique o valor do major alocado dinamicamente para ele:

```
# cat /proc/devices | grep drvled
```

Descarregue o driver, liste novamente o arquivo `/proc/devices` e verifique se o major number foi desalocado corretamente.

5.2 Implementando a operação de escrita do driver

O driver será responsável pelo acionamento de um led conectado a um GPIO da placa. Este led poderá estar em dois estados: ON e OFF. No momento, vamos nos preocupar apenas com o controle do status do led. Veremos o acionamento físico do led via GPIO mais adiante no treinamento.

Inclua o arquivo de cabeçalho `<asm/uaccess.h>`:

```
#include <asm/uaccess.h>
```

Crie uma variável global para armazenar o status do led (0 para desligado e 1 para ligado):

```
#define LED_OFF         0
#define LED_ON          1

static unsigned int led_status = LED_OFF;
```

Crie uma função para alterar o status do led:

```
static void drvled_setled(unsigned int status)
{
     led_status = status;
}
```

Implemente a função de escrita do driver `drvled_write()`, que deverá receber do usuário uma string (“`0`” ou “`1`”) e mudar o status do led.

```
static ssize_t drvled_write(struct file *file, const char __user *buf,
                            size_t count, loff_t *ppos)
{
     char kbuf = 0;
```

```
     if (copy_from_user(&kbuf, buf, 1)) {
          return -EFAULT;
     }

     if (kbuf == '1') {
          drvled_setled(LED_ON);
          pr_info("LED ON!\n");
     } else if (kbuf == '0') {
          drvled_setled(LED_OFF);
          pr_info("LED OFF!\n");
     }

     return count;
}
```

Implemente a estrutura `file_operations`, adicionando a rotina de escrita do driver:

```
static struct file_operations drvled_fops = {
     .owner = THIS_MODULE,
     .write = drvled_write,
};
```

Agora precisamos registrar o dispositivo de caractere.

5.3 Registrando o dispositivo de caractere

Inclua o arquivo de cabeçalho `<linux/cdev.h>`:

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

Crie uma variável global do tipo `struct cdev` para armazenar a estrutura do dispositivo de caractere.

```
static struct cdev drvled_cdev;
```

Na rotina de inicialização do driver, após a chamada à função `alloc_chrdev_region()`, inicialize e registre o dispositivo de caractere com as funções `cdev_init()` e `cdev_add()`.

```
cdev_init(&drvled_cdev, &drvled_fops);

if ((result = cdev_add(&drvled_cdev, drvled_dev, 1))) {
     pr_err("%s: Char driver registration failed!\n", DRIVER_NAME);
     unregister_chrdev_region(drvled_dev, 1);
     return result;
}
```

Na rotina de limpeza do driver, remova o dispositivo de caractere com a função `cdev_del()`.

```
cdev_del(&drvled_cdev);
unregister_chrdev_region(drvled_dev, 1);
pr_info("%s: exiting.\n", DRIVER_NAME);
```

Compile e instale o módulo.

Acesse o terminal serial do target e carregue o módulo.

Crie o arquivo de dispositivo do driver, substituindo o campo `<major>` pelo major alocado para o seu driver (consulte `/proc/devices` se necessário):

```
# mknod /dev/led c <major> 0
```

Teste a rotina de escrita do driver:

```
# echo 1 > /dev/led
LED ON!

# echo 0 > /dev/led
LED OFF!
```

Como os arquivos no diretório `/dev` são mantidos em memória pelo kernel, sempre que o target for reiniciado, é necessário recriar o arquivo de dispositivo com o comando `mknod`.

5.4 Implementando a operação de leitura do driver

Implemente a operação de leitura do driver `drvled_read()`, que deverá retornar a string "`OFF`" se o status do led for OFF ou a string "`ON`" se o status do led for ON.

Para testar, você pode ler o arquivo de dispositivo com o comando `cat`, conforme exemplos abaixo:

```
# echo 1 > /dev/led
# cat /dev/led
ON

# echo 0 > /dev/led
# cat /dev/led
OFF
```

Embedded Labworks

Laboratório 6 - Acessando o hardware

Nesta atividade iremos implementar o acesso físico ao hardware no driver de led que desenvolvemos no exercício anterior.

6.1 Requisitando e mapeando a região de I/O

Entre no diretório de exercícios do treinamento e abra o esqueleto do driver disponibilizado para esta atividade:

```
$ cd /opt/labs/ex/06
$ gedit drvled.c
```

Com o auxílio do instrutor, estude o uso da palavra-chave `goto` no código-fonte do Linux.

Inclua os arquivos de cabeçalho das funções que utilizaremos neste exercício:

```
#include <linux/ioport.h>
#include <asm/io.h>
```

O driver que iremos desenvolver irá controlar o led conectado ao GPIO 2.21 do SoC.

Para controlar este GPIO, vamos precisar dos endereços dos registradores que configuram a direção (input/output) e o status (ligado/desligado) da porta de GPIO 2 do SoC.

Estes endereços podem ser identificados na página 1538 do datasheet do SoC, disponível em `/opt/labs/docs/hardware/SOC_imx6sdl_datasheet.pdf`.

Para facilitar nosso trabalho, os endereços já foram identificados. Vamos então definir estes endereços no começo do código-fonte do driver, conforme abaixo:

```
#define GPIO2_DATA      0x020A0000
#define GPIO2_DIR       0x020A0004
```

Defina também a variável global que irá armazenar o endereço base de memória virtual para acessar os registradores do GPIO 2.

```
static void __iomem *baseaddr = NULL;
```

No começo da rotina de inicialização do driver, requisite o uso da região de I/O mapeada em memória com a função `request_mem_region()` e depois faça o mapeamento do endereço físico do GPIO para um endereço virtual com a função `ioremap()`.

```
int result = 0;

if (request_mem_region(GPIO2_DATA, 8, DRIVER_NAME) == NULL) {
     pr_err("%s: Error requesting memory-mapped I/O!\n", DRIVER_NAME);
     result = -EBUSY;
     goto ret_err_request_mem_region;
}

if ((baseaddr = ioremap(GPIO2_DATA, 8)) == NULL) {
     pr_err("%s: Error mapping I/O!\n", DRIVER_NAME);
     result = -ENOMEM;
     goto err_ioremap;
}
```

Ajuste os labels de erro no final da função de inicialização do driver:

```
ret_err_cdev_add:
     unregister_chrdev_region(drvled_dev, 1);
ret_err_alloc_chrdev_region:
     iounmap(baseaddr);
err_ioremap:
     release_mem_region(GPIO2_DATA, 8);
ret_err_request_mem_region:
ret_ok:
     return result;
```

Por fim, na rotina de limpeza do driver, libere o uso da região de I/O mapeada em memória e desfaça o mapeamento de memória virtual:

```
cdev_del(&drvled_cdev);
unregister_chrdev_region(drvled_dev, 1);
iounmap(baseaddr);
release_mem_region(GPIO2_DATA, 8);
pr_info("%s: exiting.\n", DRIVER_NAME);
```

Compile, instale e carregue o driver no target.

Confirme que a região de I/O foi alocada corretamente para o seu driver em `/proc/iomem`.

```
# cat /proc/iomem | grep drvled
```

6.2 Configurando a direção do GPIO

Crie uma função para configurar a direção do GPIO 2.21 como saída, setando o bit 21 do registrador de direção (`GPIO2_DIR`):

```
static void drvled_setdirection(void)
{
     unsigned int val;

     val = readl(baseaddr + 4);
     val |= 0x200000;
     writel(val, baseaddr + 4);
}
```

Chame esta função no final da inicialização do driver:

```
drvled_setdirection();

pr_info("%s: initialized.\n", DRIVER_NAME);
goto ret_ok;
```

Verifique se o driver compila sem erros ou warnings.

Nunca ignore um warning de compilação, principalmente quando você está compilando um código do kernel!

6.3 Acionando o GPIO

Agora altere a função `drvled_setled()`, setando ou limpando o bit 21 do registrador de dado (`GPIO2_DATA`), de acordo com o parâmetro `status` passado para a função.

Compile, instale e carregue o driver no target.

Teste o acionamento do led:

```
# echo 1 > /dev/led
LED ON!

# echo 0 > /dev/led
LED OFF!
```

Com o auxílio do instrutor, estude as deficiências deste driver e os camadas de abstração providas pelo kernel para resolver estes problemas.

Laboratório 7 - Integrando com o framework de led

Nesta atividade iremos integrar o driver desenvolvido na atividade anterior com o framework de leds do kernel.

7.1 Habilitando o framework de leds

Antes de iniciar o desenvolvimento do driver, vamos habilitar no kernel o framework de leds.

Abra a configuração do kernel, acesse o menu "`Device Drivers`" e habilite as opções abaixo:

```
[*] LED Support  --->
      [*]   LED Class Support
      [*]   LED Trigger support  --->
            <*>   LED Timer Trigger
            <*>   LED One-shot Trigger
            <*>   LED Heartbeat Trigger
            <*>   LED backlight Trigger
            [*]   LED CPU Trigger
            <*>   LED GPIO Trigger
            <*>   LED Default ON Trigger
```

Salve a configuração, recompile o kernel e teste.

Verifique se o framework de leds foi habilitado corretamente confirmando a existência do diretório abaixo:

```
# ls /sys/class/leds/
```

7.2 Integrando o driver no framework de leds

Entre no diretório de exercícios do treinamento e abra o esqueleto do driver disponibilizado para esta atividade:

```
$ cd /opt/labs/ex/07
$ gedit drvled.c
```

Inclua os arquivos de cabeçalho necessários para usar o framework de leds e as rotinas de alocação de memória do kernel:

```
#include <linux/leds.h>
#include <linux/slab.h>
```

Declare uma estrutura global para armazenar os dados do driver:

```
struct drvled_info {
     struct led_classdev cdev;
     unsigned int gpio;
};
```

E defina um ponteiro global para esta estrutura:

```
static struct drvled_info *led_info;
```

No início da rotina de inicialização do driver, aloque memória para esta estrutura:

```
int result = 0;

if (!(led_info = kzalloc(sizeof(struct drvled_info), GFP_KERNEL))) {
     pr_err("%s: kmalloc error\n", DRIVER_NAME);
     result = -ENOMEM;
     goto ret_err_kzalloc;
}
```

Logo após a alocação de memória, inicialize os campos desta estrutura com as informações do led que pretendemos registrar:

```
led_info->cdev.name = "ipe:green:usr1";
led_info->gpio = 53;
led_info->cdev.brightness_set = drvled_change_state;
```

A variável `name` indica o nome que será atribuído ao led. A variável `gpio` indica o número do GPIO associado a este led (esta informação será utilizada apenas no próximo exercício). A variável `brightness_set` armazena a rotina de callback que será chamada para alterar o status do led.

Antes do final da rotina de inicialização do driver, registre o led no framework de leds do kernel:

```
if ((result = led_classdev_register(NULL, &led_info->cdev))) {
     pr_err("%s: led_classdev_register error\n", DRIVER_NAME);
     goto ret_err_led_classdev_register;
}

drvled_setdirection();

pr_info("%s: initialized.\n", DRIVER_NAME);
goto ret_ok;
```

Agora ajuste os labels de tratamento de erro no final da rotina de inicialização do driver:

```
ret_err_led_classdev_register:
     iounmap(baseaddr);
err_ioremap:
     release_mem_region(GPIO2_DATA, 8);
ret_err_request_mem_region:
     kfree(led_info);
ret_err_kzalloc:
ret_ok:
     return result;
```

Não esqueça de desalocar os recursos na rotina de limpeza do driver:

```
led_classdev_unregister(&led_info->cdev);
iounmap(baseaddr);
release_mem_region(GPIO2_DATA, 8);
kfree(led_info);
pr_info("%s: exiting.\n", DRIVER_NAME);
```

7.3 Implementando a rotina de callback do led

Agora implemente a rotina de callback do led `drvled_change_state()`, que será chamada pelo framework de leds para alterar o status do led.

Compile e instale o driver no target.

7.4 Testando o driver

Carregue o driver e verifique se o led foi registrado com sucesso:

```
# ls /sys/class/leds/ipe:green:usr1/
```

Teste o acionamento do led:

```
# echo 1 > /sys/class/leds/ipe:green:usr1/brightness
# echo 0 > /sys/class/leds/ipe:green:usr1/brightness
```

O framework de leds possui a funcionalidade de trigger, que é basicamente uma forma do kernel controlar o status do led de acordo com algum evento interno ou externo.

Liste os triggers disponíveis com o comando abaixo (o trigger habilitado estará entre colchetes):

```
# cat /sys/class/leds/ipe:green:usr1/trigger
[none] mmc0 mmc2 timer oneshot heartbeat backlight gpio cpu0 cpu1 default-on
```

Teste o trigger de heartbeat (o led ficará piscando em uma frequência pré-definida pelo kernel):

```
# echo heartbeat > /sys/class/leds/ipe:green:usr1/trigger
```

Teste também o trigger de acesso à interface de cartão SD:

```
# echo mmc0 > /sys/class/leds/ipe:green:usr1/trigger
```

Force uma operação de escrita no cartão para ver o trigger funcionando:

```
# dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=1 seek=10 && sync
```

Laboratório 8 - Usando a camada de GPIO do kernel

Nesta atividade iremos alterar o driver desenvolvido na atividade anterior para utilizar a camada de GPIO do kernel.

A documentação da camada de GPIO do Linux está disponível no código-fonte do kernel em `Documentation/gpio.txt`.

8.1 Habilitando a controladora de GPIO

Para que pudéssemos acessar diretamente os registradores nos exercícios anteriores, a controladora da porta de GPIO 2 foi desabilitada no device tree do kit de desenvolvimento.

Agora, como a idéia é acessar o GPIO através da biblioteca de GPIOs do kernel, precisamos da controladora de GPIO 2 habilitada no kernel.

Para isso, abra o device tree do kit de desenvolvimento, procure pela entrada `gpio2` no final do arquivo e altere conforme abaixo:

```
$ cd /opt/labs/ex/01/linux
$ gedit arch/arm/boot/dts/imx6dl-colibri-ipe.dts
```

```
&gpio2 {
     status = "okay";
};
```

Não se preocupe neste momento com os detalhes desta alteração. Estudaremos sobre o device tree em detalhes mais adiante no treinamento.

Agora recompile o device tree, atualize o diretório de TFTP e reinicie o target.

Para testar, execute o comando abaixo para exportar o acesso ao GPIO 2.21:

```
# echo 53 > /sys/class/gpio/export
```

Os comandos abaixo deverão acionar diretamente o GPIO 2.21, e consequentemente ligar e desligar o led do kit de desenvolvimento:

```
# echo out > /sys/class/gpio/gpio53/direction
# echo 1 > /sys/class/gpio/gpio53/value
# echo 0 > /sys/class/gpio/gpio53/value
```

Por fim, desfaça o export para que o driver que desenvolveremos neste exercício possa alocar o GPIO 2.21:

```
# echo 53 > /sys/class/gpio/unexport
```

8.2 Usando a biblioteca de GPIO do kernel

Entre no diretório de exercícios do treinamento e abra o esqueleto do driver disponibilizado para esta atividade:

```
$ cd /opt/labs/ex/08
$ gedit drvled.c
```

Inclua o arquivo de cabeçalho para acessar a biblioteca de GPIO do kernel:

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

Na rotina de inicialização do driver, antes da chamada à função `led_classdev_register()`, requisite o uso do GPIO com a função `gpio_request_one()`:

```
led_info->cdev.brightness_set = drvled_change_state;

if ((result = gpio_request_one(led_info->gpio, GPIOF_DIR_OUT,
                               led_info->cdev.name))) {
     pr_err("%s: gpio_request error\n", DRIVER_NAME);
     goto ret_err_gpio_request;
}
```

No final da rotina de inicialização do driver, atualize os labels de tratamento de erro:

```
ret_err_led_classdev_register:
        gpio_free(led_info->gpio);
ret_err_gpio_request:
        kfree(led_info);
ret_err_kzalloc:
ret_ok:
        return result;
```

Na rotina de limpeza do driver, não esqueça de liberar o GPIO alocado:

```
led_classdev_unregister(&led_info->cdev);
gpio_free(led_info->gpio);
kfree(led_info);
```

Verifique se o driver compila sem problemas ou warnings.

8.3 Implementando a rotina de callback

Agora implemente a rotina de callback do led `drvled_change_state()`, acionando o GPIO com a função `gpio_set_value()` da biblioteca de GPIO do kernel.

Compile, instale e carregue o driver no target.

8.4 Testando o driver

Monte o sistema de arquivos `debugfs`:

```
# mount -t debugfs none /sys/kernel/debug/
```

Verifique se o GPIO 53 foi alocado para o driver do led:

```
# cat /sys/kernel/debug/gpio | grep gpio-53
  gpio-53  (ipe:green:usr1      ) out lo
```

Teste novamente o funcionamento do driver:

```
# echo 1 > /sys/class/leds/ipe:green:usr1/brightness
# echo 0 > /sys/class/leds/ipe:green:usr1/brightness
# echo heartbeat > /sys/class/leds/ipe:green:usr1/trigger
```

Laboratório 9 - Integração com o driver model do kernel

Nesta atividade iremos integrar o driver desenvolvido no exercício anterior ao driver model do kernel.

9.1 Definindo a estrutura platform_data

Entre no código-fonte do kernel e defina a estrutura `platform_data` do driver de led, que será a interface entre o driver e a declaração do dispositivo de hardware.

```
$ cd /opt/labs/ex/01/linux/
$ gedit include/linux/drvled.h
```

```
struct drvled_platform_data {
     const char *name;
     unsigned int gpio;
};
```

9.2 Transformando o driver em um platform driver

Entre no diretório de exercícios do treinamento e abra o esqueleto do driver disponibilizado para esta atividade:

```
$ cd /opt/labs/ex/09
$ gedit drvled.c
```

Inclua os arquivos de cabeçalho necessários para transformar o driver em um platform driver:

```
#include <linux/platform_device.h>
#include <linux/drvled.h>
```

No final do arquivo, antes da macro `module_init()`, defina a estrutura do platform driver:

```
static struct platform_driver drvled_driver = {
     .probe  = drvled_probe,
     .remove = drvled_remove,
     .driver = {
          .name   = "drvled",
          .owner  = THIS_MODULE,
     },
};
```

Logo abaixo da definição desta estrutura, implemente as novas rotinas de inicialização e limpeza do driver:

```
static int __init drvled_init(void)
{
     return platform_driver_register(&drvled_driver);
}

static void __exit drvled_exit(void)
{
     platform_driver_unregister(&drvled_driver);
}
```

A antiga função de inicialização será agora a função de `probe()` do driver. Altere seu protótipo conforme abaixo:

```
static int drvled_probe(struct platform_device *pdev)
{
```

Ainda nesta função, declare um ponteiro para armazenar o `platform_data` do driver:

```
struct drvled_platform_data *pdata = pdev->dev.platform_data;
int result = 0;
```

E inicialize as informações do led, lendo do `platform_data`:

```
led_info->cdev.name = pdata->name;
led_info->gpio = pdata->gpio;
led_info->cdev.brightness_set = drvled_change_state;
```

Desta forma, as informações necessárias para o funcionamento do driver, incluindo o número do GPIO, não estão mais codificadas dentro do driver. Estas informações serão passadas durante o registro do dispositivo, que implementaremos na próxima seção deste exercício.

Por fim, a antiga função de limpeza será agora a função de `remove()` do driver. Altere seu protótipo conforme abaixo:

```
static int drvled_remove(struct platform_device *pdev)
{
```

Como esta função tem um retorno, retorne 0 no fim da função:

```
kfree(led_info);
pr_info("%s: exiting.\n", DRIVER_NAME);
return 0;
```

Compile o driver e instale no target.

9.3 Registrando o dispositivo

Acesse o código-fonte do kernel e abra o arquivo que contém o código de inicialização do kit de desenvolvimento:

```
$ cd /opt/labs/ex/01/linux
$ gedit arch/arm/mach-imx/mach-imx6q.c
```

Inclua o arquivo de cabeçalho com a definição da estrutura `platform_data` do driver de led:

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

Implemente as estruturas `platform_device` e `platform_data` do driver:

```
struct drvled_platform_data drvled_pdata = {
	.name = "ipe:green:usr1",
	.gpio = 53,
};

static struct platform_device drvled_device = {
	.name = "drvled",
	.dev = {
		.platform_data = &drvled_pdata,
	},
};
```

No final da função `imx6q_init_machine()`, registre o dispositivo:

```
	if (of_machine_is_compatible("toradex,apalis_imx6q"))
		apalis_reset_moci();

	platform_device_register(&drvled_device);
}
```

Recompile o kernel, atualize o diretório de TFTP e reinicie o target.

9.4 Testando o driver

Carregue o driver.

Verifique se o dispositivo foi registrado:

```
# ls /sys/bus/platform/devices/drvled.0/
driver     modalias   power      subsystem  uevent
```

Verifique se o driver foi registrado:

```
# ls /sys/bus/platform/drivers/drvled/
bind      drvled.0  module    uevent    unbind
```

Teste novamente o funcionamento do driver:

```
# echo 1 > /sys/class/leds/ipe:green:usr1/brightness
# echo 0 > /sys/class/leds/ipe:green:usr1/brightness
# echo heartbeat > /sys/class/leds/ipe:green:usr1/trigger
```

Embedded Labworks

Laboratório 10 - Trabalhando com device tree

Neste exercício iremos adaptar o driver desenvolvido na atividade anterior para trabalhar com o mecanismo de device tree.

10.1 Driver com suporte ao device tree

Entre no diretório de exercícios do treinamento e abra o esqueleto do driver disponibilizado para esta atividade:

```
$ cd /opt/labs/ex/10
$ gedit drvled.c
```

Inclua os arquivos de cabeçalho necessários para se trabalhar neste exercício com o device tree:

```
#include <linux/of.h>
#include <linux/of_platform.h>
#include <linux/of_gpio.h>
```

Antes da definição da estrutura `platform_driver`, implemente a estrutura `of_device_id`:

```
static const struct of_device_id of_drvled_match[] = {
     { .compatible = "labworks,drvled", },
     {},
};
```

E altere a estrutura `platform_driver`, inicializando o campo `of_match_table` com o ponteiro para a estrutura `of_device_id` criada acima:

```
static struct platform_driver drvled_driver = {
     .probe  = drvled_probe,
     .remove = drvled_remove,
     .driver = {
          .name   = "led",
          .owner  = THIS_MODULE,
          .of_match_table = of_drvled_match,
     },
};
```

Agora precisamos alterar a função `probe()` para ler do device tree as informações do dispositivo.

No início da função `probe()`, remova o ponteiro `pdata` e declare a estrutura que contém o nó do dispositivo no device tree.

```
static int drvled_probe(struct platform_device *pdev)
{
     struct device_node *np = pdev->dev.of_node, *child;
     int result = 0;

     child = of_get_next_child(np, NULL);
```

E então inicialize os campos `name` e `gpio` do driver, lendo as informações do nó do device tree:

```
led_info->cdev.name = of_get_property(child, "label", NULL);
led_info->gpio = of_get_gpio(child, 0);
led_info->cdev.brightness_set = drvled_set_led;
```

Compile e instale o driver no target.

10.2 Alterando o device tree

Agora só falta registrar o dispositivo no device tree:

```
$ cd /opt/labs/ex/01/linux/
$ gedit arch/arm/boot/dts/imx6dl-colibri-ipe.dts
```

```
     regulators {
          reg_usb_host_vbus: usb_host_vbus {
               status = "okay";
          };
     };

     leds {
          compatible = "labworks,drvled";

          led1 {
               label = "ipe:green:usr1";
               gpios = <&gpio2 21 GPIO_ACTIVE_HIGH>;
          };
     };
};
```

Recompile o device tree, atualize o diretório de TFTP e reinicie o target.

10.3 Testando o driver com suporte ao device tree

Verifique se a descrição do hardware no device tree foi lida corretamente.

```
# ls /proc/device-tree/leds
```

```
# echo `cat /proc/device-tree/leds/led1/label`
```

Carregue o driver e teste novamente seu funcionamento:

```
# echo 1 > /sys/class/leds/ipe:green:usr1/brightness
# echo 0 > /sys/class/leds/ipe:green:usr1/brightness
# echo heartbeat > /sys/class/leds/ipe:green:usr1/trigger
```

10.4 Adicionando suporte ao mecanismo de trigger

Passe a opção "`default-trigger`" no nó do led, conforme abaixo:

```
led1 {
    label = "ipe:green:usr1";
    gpios = <&gpio2 21 GPIO_ACTIVE_HIGH>;
    default-trigger = "heartbeat";
};
```

Altere o driver para ler e inicializar o led com o trigger configurado no device tree.

Compile e teste.

10.5 Suportando múltiplos leds

Quais seriam as alterações necessárias para o driver suportar múltiplos leds?

Laboratório 11 - Kernel threads e wait queues

Nesta atividade estudaremos kernel threads e wait queues através do desenvolvimento de um driver de dispositivo de caractere que irá exportar para o usuário os eventos de um botão conectado no kit de desenvolvimento.

O driver terá uma thread do kernel monitorando periodicamente um botão conectado a um GPIO do kit de desenvolvimento. Quando o botão for pressionado, a thread utilizará uma wait queue para notificar a rotina de leitura do botão.

11.1 Desenvolvendo o driver

Entre no diretório de exercícios do treinamento e abra o esqueleto do driver disponibilizado para esta atividade:

```
$ cd /opt/labs/ex/11
$ gedit drvbtn.c
```

Antes de começar, estude o esqueleto do driver disponibilizado para esta atividade.

Inclua os arquivos de cabeçalho necessários para as funções que utilizaremos neste exercício:

```
#include <linux/kthread.h>
#include <linux/delay.h>
#include <linux/wait.h>
```

Inclua na estrutura `drvbtn_data` as variáveis para armazenar os ponteiros da wait queue e da thread:

```
struct drvbtn_data {
     dev_t devt;
     struct cdev cdev;
     int gpio;
     unsigned int event;
     unsigned long counter;
     struct task_struct *task;
     wait_queue_head_t wait;
};
```

Na rotina de `probe()` do driver, logo após a chamada à função `cdev_add()`, crie e inicialize a thread:

```
data->task = kthread_run(drvbtn_thread, data, DRIVER_NAME);
if (IS_ERR(data->task)) {
     pr_err("%s: Error creating thread!\n", DRIVER_NAME);
     ret = PTR_ERR(data->task);
     goto ret_err_kthread_run;
}
```

E logo abaixo inicialize a wait queue:

```
init_waitqueue_head(&data->wait);
```

Não esqueça de criar o label de erro no fim da função de `probe()` do driver:

```
ret_err_kthread_run:
     cdev_del(&data->cdev);
ret_err_cdev_add:
     unregister_chrdev_region(data->devt, 1);
```

E na rotina de `remove()` do driver, é necessário parar a execução da thread:

```
static int drvbtn_remove(struct platform_device *pdev)
{
     struct drvbtn_data *data = platform_get_drvdata(pdev);

     kthread_stop(data->task);
     cdev_del(&data->cdev);
     unregister_chrdev_region(data->devt, 1);
```

Agora vamos implementar as rotinas de callback do driver.

A rotina de callback de abertura do driver servirá apenas para armazenar o ponteiro para a estrutura que contém os dados do driver, para que esta estrutura possa ser referenciada mais adiante pela rotina de leitura do driver.

```
static int drvbtn_open(struct inode *i, struct file *file)
{
     struct drvbtn_data *data;

     data = container_of(i->i_cdev, struct drvbtn_data, cdev);
     file->private_data = data;

     data->counter = 0;

     return 0;
}
```

A rotina de callback de leitura do driver irá esperar na wait queue por um evento do botão, e quando o

evento acontecer, enviará uma mensagem formatada para o usuário:

```
static ssize_t drvbtn_read(struct file *file, char __user * buf,
                           size_t count, loff_t * ppos)
{
     struct drvbtn_data *data = file->private_data;
     char msg[256];
     int ret = 0;

     if (!wait_event_interruptible(data->wait, data->event == 1)) {

          data->event = 0;

          sprintf(msg, "Button pressed %ld times!\n", ++data->counter);

          if ((ret = copy_to_user(buf, msg, strlen(msg))))
               ret = -EFAULT;
          else
               ret = strlen(msg);
     }

     return ret;
}
```

11.2 Implementando a thread

Agora vamos implementar a thread. Utilize o código-fonte abaixo como referência para a implementação da thread:

```
static int drvbtn_thread(void *param)
{
     struct drvbtn_data *data = (struct drvbtn_data *)param;

     while (!kthread_should_stop()) {
          // implementar a lógica aqui!
     }

     return 0;
}
```

Implemente a lógica da thread de forma que:

- ✔ Seja uma thread periódica de 100ms. Para isso, use a função `msleep()` do kernel.
- ✔ A cada iteração, a thread deverá ler o status do botão com a função `gpio_get_value()`.
- ✔ Se o botão for pressionado, a thread deverá setar a flag `data->event` e notificar a rotina de leitura do botão com a função `wake_up_interruptible()`.

Para saber quais rotinas de delay são providas pelo kernel, consulte a documentação em `Documentation/timers/timers-howto.txt` e o arquivo de cabeçalho em `<linux/delay.h>`.

Compile e instale o driver no target.

11.3 Adicionando o dispositivo no device tree

Antes de testar o driver, é necessário adicionar o dispositivo no device tree:

```
$ cd /opt/labs/ex/01/linux/
$ gedit arch/arm/boot/dts/imx6dl-colibri-ipe.dts
```

```
     leds {
          compatible = "labworks,drvled";

          led1 {
               label = "ipe:green:usr1";
               gpios = <&gpio2 21 GPIO_ACTIVE_HIGH>;
               default-trigger = "heartbeat";
          };
     };

     buttons {
          compatible = "labworks,drvbtn";

          btn1 {
               gpios = <&gpio6 6 GPIO_ACTIVE_LOW>;
          };
     };
};
```

Recompile o device tree, atualize o diretório do TFTP e reinicie o target.

Confirme se o nó do botão foi lido corretamente do device tree:

```
# echo `cat /proc/device-tree/buttons/btn1/name`
```

11.4 Testando o driver

Carregue o driver.

Verifique se a thread de leitura do botão está em execução:

```
# ps | grep drvbtn
```

Para testar o driver, é necessário criar um arquivo de dispositivo. Para isso, use o comando abaixo, substituindo o campo `<major>` pelo major number criado para o driver (consulte `/proc/devices` se necessário):

```
# mknod /dev/btn c <major> 0
```

Use o comando `cat` para ler o arquivo de dispositivo e verifique se ele exibe os eventos do botão S1:

```
# cat /dev/btn
```

Verifique se você consegue cancelar a leitura com `CTRL+C`.

Remova o driver e verifique se a thread é encerrada.

O driver que desenvolvemos neste exercício tem algumas deficiências, dentre elas ele trabalha por polling e não está integrado ao framework de input do kernel.

No próximo exercício iremos resolver estes problemas, implementando a rotina de tratamento de interrupção do botão e integrando o driver ao framework de input do kernel.

Laboratório 12 - Trabalhando com interrupções

Neste exercício iremos alterar o driver de botão desenvolvido na atividade anterior para trabalhar com interrupções e workqueues.

12.1 Desenvolvendo o driver

Entre no diretório de exercícios do treinamento e abra o esqueleto do driver disponibilizado para esta atividade:

```
$ cd /opt/labs/ex/12
$ gedit drvbtn.c
```

Antes de começar, analise o código-fonte do driver e estude sua integração com o framework de input do kernel.

Inclua os arquivos de cabeçalho necessários para as funções que utilizaremos neste exercício:

```
#include <linux/interrupt.h>
#include <linux/workqueue.h>
```

Inclua na estrutura `drvbtn_data` as variáveis para armazenar o número da IRQ e a workqueue:

```
struct drvbtn_data {
     struct input_dev *input;
     unsigned int gpio;
     unsigned int key;
     unsigned int irq;
     struct work_struct work;
};
```

Na rotina de `probe()` do driver, logo após a chamada à função `gpio_request_one()`, identifique o número da IRQ associada ao GPIO:

```
if ((data->irq = gpio_to_irq(data->gpio)) < 0) {
     pr_err("%s: Unable to get irq number!\n", DRIVER_NAME);
     ret = data->irq;
     goto ret_err_gpio_to_irq;
}
```

Logo em seguida, requisite o uso da IRQ:

```
ret = request_irq(data->irq, drvbtn_isr,
                  IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                  DRIVER_NAME, data);
if (ret < 0) {
     pr_err("%s: Error requesting irq!\n", DRIVER_NAME);
     goto ret_err_request_irq;
}
```

E no final da função `probe()`, inicialize a workqueue:

```
INIT_WORK(&data->work, drvbtn_work_func);

pr_info("%s: probe successful.\n", DRIVER_NAME);
return 0;
```

Ainda na função `probe()`, ajuste os labels de erro:

```
err_input_register_device:
     free_irq(data->irq, data);
ret_err_request_irq:
ret_err_gpio_to_irq:
     gpio_free(data->gpio);
ret_err_gpio_request:
ret_err_of_get_gpio:
     input_free_device(data->input);
```

E na rotina `remove()` do driver, libere a IRQ:

```
static int drvbtn_remove(struct platform_device *pdev)
{
     struct drvbtn_data *data = platform_get_drvdata(pdev);

     input_unregister_device(data->input);
     free_irq(data->irq, data);
     gpio_free(data->gpio);
```

Agora é só implementar a ISR, utilizando a workqueue para delegar o tratamento do evento do botão:

```
static irqreturn_t drvbtn_isr(int irq, void *dev_id)
{
     struct drvbtn_data *data = dev_id;

     schedule_work(&data->work);

     return IRQ_HANDLED;
}
```

12.2 Implementando a workqueue

Vamos agora implementar a workqueue.

Utilize o código-fonte abaixo como referência para a implementação da workqueue:

```
static void drvbtn_work_func(struct work_struct *work)
{
     struct drvbtn_data *data;

     data = container_of(work, struct drvbtn_data, work);

     // reportar evento aqui!
}
```

Implemente a lógica da workqueue conforme abaixo:

- ✔ Leia o status do GPIO com a função `gpio_get_value()`.
- ✔ Reporte o evento para o framework de input com as funções `input_report_key()` e `input_sync()`.

Compile e instale o driver no target.

12.3 Testando o driver

Carregue o driver.

Verifique se a interrupção foi registrada corretamente:

```
# cat /proc/interrupts
```

Verifique também se o driver se registrou corretamente no framework de input do kernel:

```
# ls -l /dev/input/event0
```

Teste o funcionamento do driver com a ferramenta `evtest`, capaz de monitorar e exibir os eventos da camada de input do kernel (um evento deverá ser gerado cada vez que o botão S1 for pressionado ou solto).

```
# evtest /dev/input/event0
```

Após os testes, você pode verificar se o contador de interrupções do GPIO foi incrementado:

```
# cat /proc/interrupts
```

Laboratório 13 - Usando mutex

Nesta atividade estudaremos o uso de mutex como mecanismo de locking do kernel.

13.1 Protegendo seções críticas com mutex

O kit de desenvolvimento possui uma expansora de GPIO conectada no barramento SPI do SoC.

Está disponível um driver para esta expansora de GPIO no diretório de exercícios do treinamento, porém este driver precisa de ajustes, já que algumas seções críticas estão desprotegidas. Nosso objetivo é utilizar um mutex para proteger as seções críticas do driver.

Abra o código-fonte do driver:

```
$ cd /opt/labs/ex/13
$ gedit gpio-mcp23s08.c
```

Implemente as seguintes alterações no driver:

- ✔ Declare um mutex dentro da estrutura `struct mcp23s08`.
- ✔ Inicialize o mutex no começo da função `mcp23s08_probe_one()`.
- ✔ Proteja as seções críticas com o mutex. As seções críticas estão indicadas no código-fonte com os comentários "`// LOCK HERE`" e "`// UNLOCK HERE`".

Compile o driver e instale no target.

13.2 Testando o driver

Carregue o driver:

```
# modprobe gpio-mcp23s08
```

A expansora de GPIO irá exportar para o sistema 8 GPIOs (248 até 255), que estão conectados ao display de 7 segmentos do kit de desenvolvimento.

Execute os comandos abaixo para testar o acesso ao GPIO 248:

```
# echo 248 > /sys/class/gpio/export
# echo out > /sys/class/gpio/gpio248/direction
# echo 1 > /sys/class/gpio/gpio248/value
# echo 0 > /sys/class/gpio/gpio248/value
```

Sinta-se livre para testar os outros GPIOs da expansora.

Laboratório 14 - Usando ferramentas de debugging

Nesta atividade estudaremos alguns mecanismos de depuração do kernel Linux.

14.1 Analisando crash dumps

Force um crash do kernel no target usando o mecanismo de SysRq:

```
# echo c > /proc/sysrq-trigger
```

Analise o dump gerado pelo kernel no terminal.

Perceba no dump que o problema aconteceu na função `sysrq_handle_crash()`:

```
SysRq : Trigger a crash
Unable to handle kernel NULL pointer dereference at virtual address 00000000
pgd = 8a920000
[00000000] *pgd=1a6c9831, *pte=00000000, *ppte=00000000
Internal error: Oops: 817 [#1] SMP ARM
Modules linked in:
CPU: 0 PID: 173 Comm: sh Not tainted 3.14.52 #14
task: 8a1b8d80 ti: 8a992000 task.ti: 8a992000
PC is at sysrq_handle_crash+0x38/0x48
LR is at l2x0_cache_sync+0x48/0x4c
pc : [<8030fcf0>]    lr : [<8001d608>]    psr: 600d0093
sp : 8a993ed0  ip : 8a993eb8  fp : 8a993edc
r10: 00000000  r9 : 8a992000  r8 : 00000007
r7 : 600d0013  r6 : 00000063  r5 : 808692a0  r4 : 8087976c
r3 : 00000000  r2 : 00000001  r1 : a00d0093  r0 : 80898d34
[...]
```

Acesse o código-fonte do kernel e faça uma busca pela implementação da função `sysrq_handle_crash()`.

Procure entender o motivo do problema.

14.2 Depurando o driver de PWM

Compile e instale o driver de PWM disponibilizado no diretório de exercícios do treinamento:

```
$ cd /opt/labs/ex/14
$ make
$ make install
```

Carregue o driver no target (um bug no driver causará um kernel oops):

```
# modprobe pwm-imx
Unable to handle kernel NULL pointer dereference at virtual address 00000018
pgd = 8a9d8000
[00000018] *pgd=1a94b831, *pte=00000000, *ppte=00000000
Internal error: Oops: 817 [#1] SMP ARM
Modules linked in: pwm_imx(O+)
CPU: 0 PID: 177 Comm: modprobe Tainted: G           O 3.14.52 #5
task: 8a60a880 ti: 8a300000 task.ti: 8a300000
PC is at imx_pwm_probe+0x60/0x310 [pwm_imx]
LR is at of_match_node+0x40/0x4c
pc : [<7f000a00>]    lr : [<804cd530>]    psr: 200f0013
sp : 8a301c98  ip : 8a301c68  fp : 8a301cf4
r10: 80087adc  r9 : 8a300000  r8 : 00000002
r7 : 7f001090  r6 : fffffdfb  r5 : 8a143210  r4 : fffffffe
r3 : 7f00104c  r2 : 00000000  r1 : a00f0013  r0 : 7f000e2c
Flags: nzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
Control: 10c5387d  Table: 1a9d804a  DAC: 00000015
Process modprobe (pid: 177, stack limit = 0x8a300238)
```

Perceba na mensagem de kernel oops a localização do PC (program counter) no momento do crash.

Podemos utilizar esta informação no GDB para identificar a linha de código com problema:

```
$ gdb-multiarch pwm-imx.ko
(gdb) set arch arm
(gdb) list *(imx_pwm_probe+0x60)
0xa20 is in imx_pwm_probe (/opt/labs/ex/14/pwm-imx.c:284).
```

Ou seja, o problema está na linha 284 do arquivo `pwm-imx.c`.

Use o comando `quit` para sair do GDB.

Agora que você identificou a linha de código com problema, corrija o driver e teste.

Para testar, você pode utilizar o buzzer conectado em uma saída de PWM do SoC.

```
# echo 0 > /sys/class/pwm/pwmchip1/export
# echo 1000000 > /sys/class/pwm/pwmchip1/pwm0/period
# echo 500000 > /sys/class/pwm/pwmchip1/pwm0/duty_cycle
# echo 1 > /sys/class/pwm/pwmchip1/pwm0/enable
```

Mais informações sobre o framework de PWM estão disponíveis no código-fonte do kernel em `Documentation/pwm.txt`.

Laboratório 15 - Analisando a implementação de drivers

Nesta atividade iremos analisar a implementação de alguns drivers disponíveis no código-fonte do kernel:

- `gpio/gpio-74x164.c`: Shift register de 8 bits com interface SPI.
- `input/keyboard/ep93xx_keypad.c`: Teclado matricial.
- `video/atmel_lcdfb.c`: Controlador LCD para chips Atmel AT91/AT32.
- `media/radio/radio-mr800.c`: Radio FM com interface USB.
- `misc/ti_dac7512.c`: Conversor analógico/digital com interface SPI.
- `rtc/rtc-bfin.c`: RTC da Blackfin.
- `usb/serial/pl2303.c`: Conversor USB/Serial.