Linux Developer Conference Brazil 2018

O modelo de desenvolvimento de drivers do kernel Linux

Embedded **Labworks**

Por Sergio Prado. São Paulo, Agosto de 2018
® Copyright Embedded Labworks 2004-2018. All rights reserved.

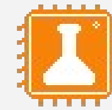
SOBRE ESTE DOCUMENTO

- Este documento é disponibilizado sob a Licença Creative Commons BY-SA 3.0.
  http://creativecommons.org/licenses/by-sa/3.0/legalcode

- Os fontes deste documento estão disponíveis em:
  https://sergioprado.org/palestras/linuxdevbr2018

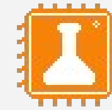
SOBRE O PALESTRANTE

- Mais de 20 anos de experiência em desenvolvimento de software para sistemas embarcados.
- Sócio da Embedded Labworks, onde atua com consultoria, treinamento e desenvolvimento de software para sistemas embarcados.
  https://e-labworks.com
- Ativo na comunidade de sistemas embarcados no Brasil, sendo um dos criadores do site Embarcados, administrador do grupo sis_embarcados no Google Groups, além de manter um blog pessoal sobre assuntos da área.
  http://sergioprado.org
- Colaborador de alguns projetos de software livre, incluindo o Buildroot e o kernel Linux.



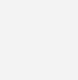
SOBRE A PALESTRA

- O objetivo da palestra é apresentar a arquitetura moderna de um driver de dispositivo do kernel Linux, baseado no driver model.
- Não falaremos sobre a implementação do driver model e suas estruturas internas (kobjects, ktypes, ksets, etc).
- Também não entraremos em detalhes sobre a estrutura interna do sysfs, um sistema de arquivos virtual que exporta uma visão dos kobjects e seus atributos para o espaço do usuário.
- O foco da palestra é no desenvolvimento de drivers de dispositivo para o kernel Linux.



AGENDA

1. Driver de dispositivo de caractere.
2. Introdução ao driver model.
3. Classes ou frameworks.
4. Infraestrutura de barramento.
5. Device tree.



KERNEL LINUX

Aplicação
Aplicação
Biblioteca
Biblioteca
Aplicação
Biblioteca C

User space

Chamadas de sistema

Notificação de eventos
Exportação de informações

Kernel Linux

Gerenciamento do hardware

Notificação de eventos

Hardware

DRIVERS DE DISPOSITIVO

- As bibliotecas e aplicações rodam em espaço de usuário, com acesso restrito aos recursos do sistema, incluindo o hardware.
- Portanto, um papel importante do kernel é prover um mecanismo de acesso ao hardware para as bibliotecas e aplicações.
- Seguindo os conceitos de sistemas Unix, onde os "objetos" do sistema operacional são exportados através de arquivos, uma das formas mais comuns utilizadas para implementar drivers é através do conceito de arquivos de dispositivo.

ARQUIVOS DE DISPOSITIVO

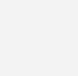
- Os arquivos de dispositivo são arquivos de acesso ao hardware, exportados para o usuário no diretório `/dev`.

- Cada arquivo de dispositivo possui 3 informações básicas, que identificam internamente o dispositivo ao qual o arquivo pertence:
  - Tipo (caractere ou bloco).
  - Major number (categoria do dispositivo).
  - Minor number (identificador do dispositivo).

ARQUIVOS DE DISPOSITIVO (cont.)

- Exemplos de arquivos de dispositivo:

```
brw-r-----  1 root   root  31,   0 Feb  7  2012 /dev/mtdblock0
brw-r-----  1 root   root   8,   1 Feb  7  2012 /dev/sda1
crw-rw-rw-  1 root   root   4,  64 Feb  7  2012 /dev/ttyS0
crw-rw-rw-  1 root   root   4,  65 Feb  7  2012 /dev/ttyS1
crw-rw-rw-  1 root   root  29,   0 Feb  7  2012 /dev/fb0
crw-rw-rw-  1 root   root   1,   1 Feb  7  2012 /dev/mem
crw-rw-rw-  1 root   root   1,   3 Feb  7  2012 /dev/null
```

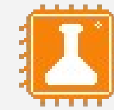


ARQUIVOS DE DISPOSITIVO (cont.)

- Através dos arquivos de dispositivo, o acesso ao hardware é abstraído para as aplicações com uma API comum de acesso a arquivos (open, read, write, close, ioctl, etc).

- Por exemplo, uma aplicação poderia escrever na porta serial com o trecho de código abaixo:

```c
int fd;

fd = open("/dev/ttyS0", O_RDWR);
write(fd, "Hello world!", 12);
close(fd);
```



ARQUIVOS DE DISPOSITIVO (cont.)

Aplicação

Leitura

Escrita

/dev/ttyS0

major/minor

read()

write()

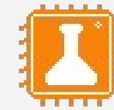
Device driver

User space

Kernel space

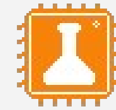
IMPLEMENTANDO UM CHAR DRIVER

- Para implementar um driver de dispositivo de caractere, são três passos principais:
  - Reservar o major/minor number, que pode ser feito com as funções `register_chrdev_region()` ou `alloc_chrdev_region()`.
  - Implementar as operações em arquivo (open, read, write, close, ioctl, etc) e defini-las na estrutura `file_operations`.
  - Registrar o char driver, inicializando uma estrutura do tipo `cdev` com a função `cdev_init()` e registrando-a no kernel com a função `cdev_add()`.



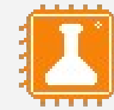
CHAR DRIVER

cdev	dev_t
cdev1	10:0
cdev2	10:1
cdev3	50:32
cdev4	4:64



Hands-on

Driver de dispositivo de caractere



Embedded Labworks

DRIVER DE DISPOSITIVO DE CARACTERE

/dev/led

User space

Problema 1

LED Driver

Kernel space

GPIOLIB

Problema 2

GPIO Driver

LED

GPIO 2.21

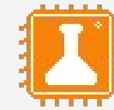
SoC

Hardware

LINUX DRIVER MODEL

- A solução para estes problemas é o modelo de desenvolvimento de drivers do kernel Linux (driver model).

- O driver model provê diversas camadas de abstração para o desenvolvimento de drivers de dispositivo no kernel Linux, e inclui os seguintes componentes:
  - **Classes** ou **frameworks**: a interface exportada pelo driver é padronizada através um framework ou classe no driver model.
  - **Barramentos**: a separação entre o driver e o dispositivo de hardware é realizada através de uma infraestrutura de barramento.


Embedded Labworks

LINUX DRIVER MODEL (cont.)

/sys/class/leds/led0/...

User space

LED Framework

Device Description

Led name = "led0"
GPIO = 2.21

LED Driver

Kernel space

GPIOLIB

Bus Infrastructure

GPIO Driver

LED

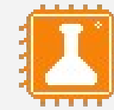
GPIO 2.21

SoC

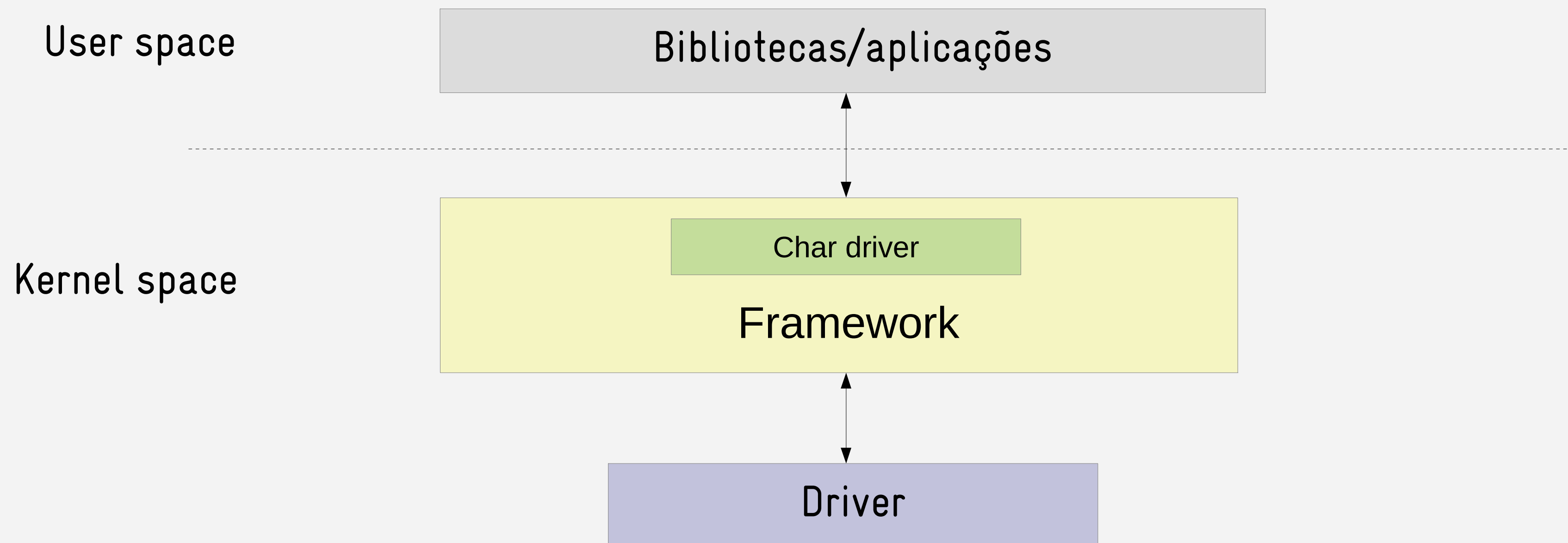
Hardware

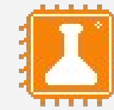
FRAMEWORKS

- Desenvolver drivers usando diretamente a API de dispositivos de caractere pode trazer alguns problemas, dentre eles:
  - Falta de padrão para dispositivos do mesmo tipo ou categoria.
  - Duplicação de código.
- Por estes motivos, os drivers *não* são normalmente implementados utilizando diretamente a API de um driver de dispositivo de caractere.
- Os drivers são normalmente implementados utilizando um framework do kernel (também chamado de classe no driver model).

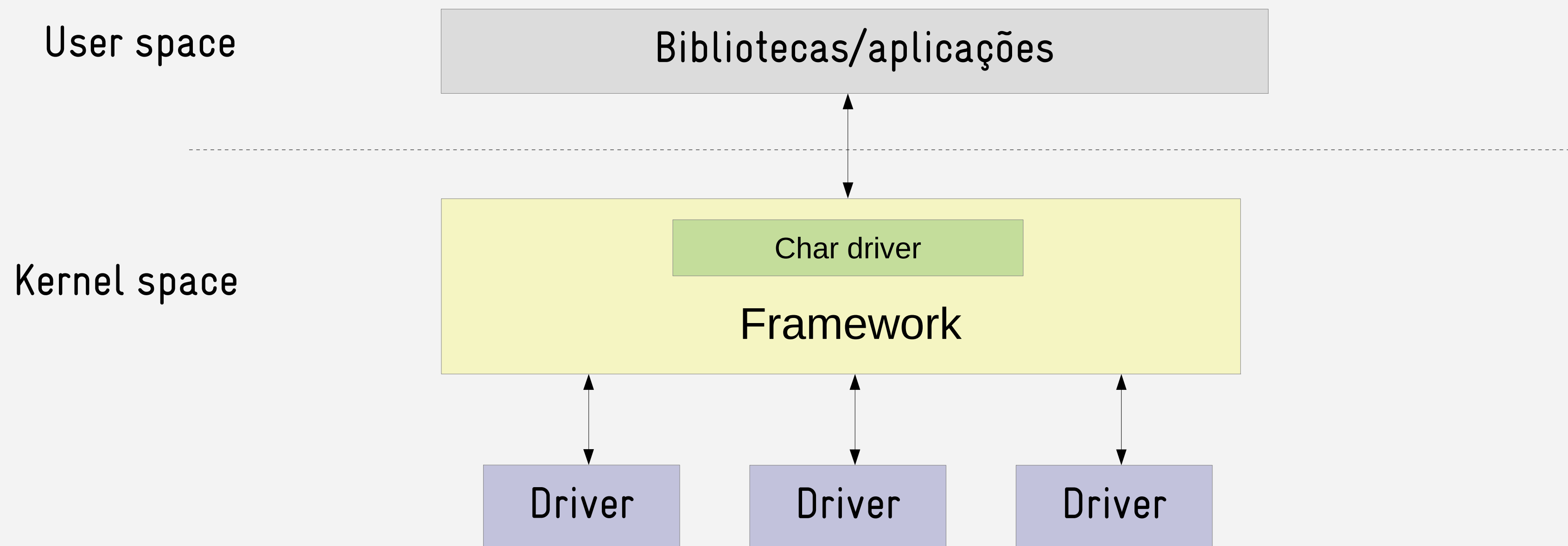


FRAMEWORKS (cont.)





FRAMEWORKS (cont.)



FRAMEWORKS (cont.)

Aplicação

Aplicação

Aplicação

Interface de chamada de sistema

Char driver

Framebuffer

Input

TTY

Block

Framebuffer driver

Input driver

TTY driver

Serial

IDE

SCSI

Serial driver

IDE driver

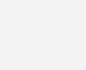
SCSI driver

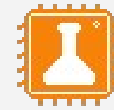
ALGUNS FRAMEWORKS

- **TTY**: dispositivos seriais (porta serial RS232, porta serial RS485, conversor USB/Serial, etc).
- **INPUT**: dispositivos de entrada do usuário (mouse, teclado, touchscreen, joystick, botão, etc).
- **IIO**: dispositivos conversores analógico/digital (DAC, ADC, acelerômetro, sensor de luz, sensor de proximidade, compasso, giroscópio, magnetômetro, etc).
- **ALSA**: dispositivos de som (controladores de áudio, placas de som, etc).

FRAMEWORK DE LEDS

- Existe um framework específico para dispositivos do tipo led.
- Este framework é habilitado na opção do kernel `CONFIG_NEW_LEDS`.
- O código-fonte do framework está disponível em `drivers/leds/` e a definição da API em `include/linux/leds.h`.
- A documentação deste framework está disponível em `Documentation/leds/`.

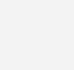


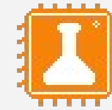


FRAMEWORK DE LEDS (cont.)

- Este framework exporta um diretório em /sys/class/leds/ para cada led registrado, disponibilizando alguns arquivos para manipular estes leds.

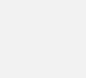
```
$ ls /sys/class/leds/led0/
brightness
device
max_brightness
power
subsystem
trigger
Uevent
```





IMPLEMENTANDO O DRIVER

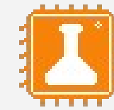
- Para um driver de leds se registrar no framework de leds é necessário:
  - Definir e inicializar uma estrutura do tipo `led_classdev`.
  - Prover uma função de callback para mudar o status do led.
  - Se registrar no framework com a função `led_classdev_register()`.



- Vários exemplos de drivers de led estão disponíveis em `drivers/leds/`.

Hands-on

Frameworks



INFRAESTRUTURA DE BARRAMENTO

/dev/led

User space

LED Framework

LED Driver

Kernel space

GPIOLIB

Problema

GPIO Driver

LED

GPIO 2.21

SoC

Hardware

INFRAESTRUTURA DE BARRAMENTO (cont.)

/sys/class/leds/led0/...

User space

LED Framework

Device Description

Led name = "led0"
GPIO = 2.21

LED Driver

Kernel space

GPIOLIB

Bus Infrastructure

GPIO Driver

LED

GPIO 2.21

SoC

Hardware

COMPONENTES

- A infraestrutura de barramento é composta por quatro componentes principais:
  - **Bus core**: implementa a API para determinado barramento (USB core, SPI core, I2C core, PCI core, etc). Representado no kernel pela estrutura `bus_type`.
  - **Bus adapters**: drivers da controladoras de barramento. Representado no kernel pela estrutura `device_driver`.
  - **Bus drivers**: drivers responsáveis por gerenciar determinado dispositivo no barramento. Representado no kernel pela estrutura `device_driver`.
  - **Bus devices**: dispositivos conectados ao barramento. Representado no kernel pela estrutura `device`.

INFRAESTRUTURA DE BARRAMENTO (cont.)

Framework

Bus driver

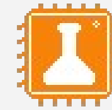
Bus infrastructure

Bus device

Bus core

Bus adapter

Hardware



INFRAESTRUTURA DE BARRAMENTO (cont.)

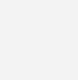
- A infraestrutura de barramento traz algumas funcionalidades, incluindo:
  - Centralização e controle do acesso a determinado barramento do sistema.
  - Separação do código do driver da descrição do dispositivo de hardware.
  - Identificação da hierarquia de dispositivos conectados aos barramentos do sistema, facilitando o gerenciamento de energia.

GERENCIAMENTO DE ENERGIA

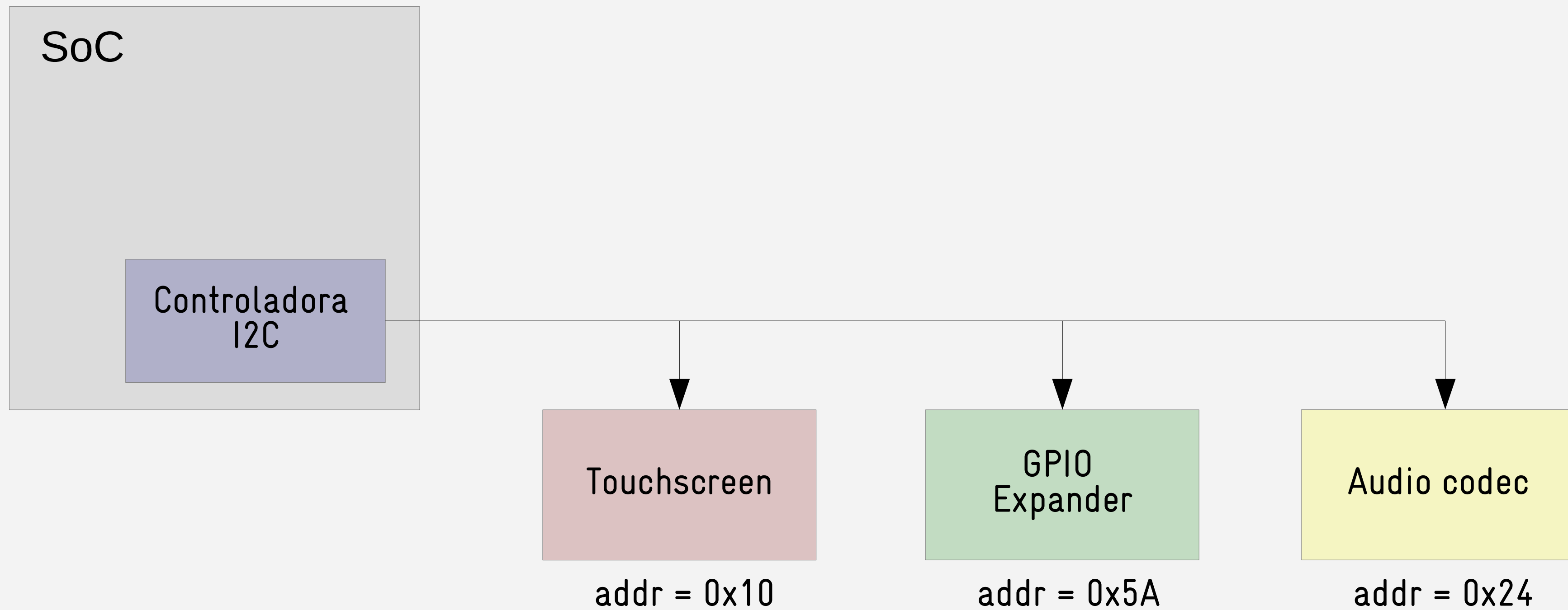
- CPU
  - PCI
    - SOUND
    - USB
      - DISK
      - MODEM
  - USB
    - ETH
  - I2C
    - ACCEL
    - E2PROM
  - PLATFORM
    - RTC

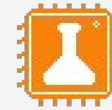
BARRAMENTO I2C

- Barramento de baixa velocidade para conectar dispositivos próximos ao processador (na mesma placa).
- Comunicação através de dois sinais: SCL (clock) e SDA (dado).
- Barramento do tipo master/slave, onde o processador rodando o kernel Linux é normalmente o master.
- Cada slave possui um endereço associado a ele.



BARRAMENTO I2C (cont.)





BARRAMENTO I2C (cont.)

- O código-fonte responsável pelo barramento I2C está disponível em `drivers/i2c/`.
- Os drivers de controladoras de barramento I2C estão disponíveis em `drivers/i2c/busses/`.
- Os drivers de dispositivos I2C estão espalhados no código-fonte do kernel em `drivers/`, de acordo com o tipo do driver.

BARRAMENTO I2C (cont.)

Driver I2C

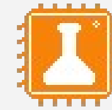
I2C core

I2C0 Adapter Driver

I2C1 Adapter Driver

I2C2 Adapter Driver

Hardware

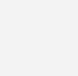


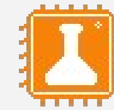
BARRAMENTO DE PLATAFORMA

- Alguns dispositivos podem não estar conectados fisicamente em um barramento, como por exemplo uma UART ou um dispositivo conectado a um GPIO.

- Como prover a mesma solução de infraestrutura de barramento sem um barramento físico?

- Através de um (pseudo) barramento chamado **platform bus**.

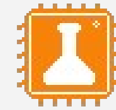




BARRAMENTO DE PLATAFORMA (cont.)

PLATFORM DRIVERS

- Para implementar um platform driver, é necessário:
  - Implementar uma estrutura do tipo `platform_device_id` ou `of_device_id` para registrar todos os dispositivos que o driver é capaz de gerenciar.
  - Definir uma estrutura do tipo `platform_driver` e inicializá-la com as callbacks do driver.
  - Registrar o driver com a função `platform_driver_register()` na inicialização do driver.



Embedded Labworks

Hands-on

Platform driver

BARRAMENTO DE PLATAFORMA

/sys/class/leds/led0/...

User space

LED Framework

Device Description

Led name = “led0”
GPIO = 2.21

LED Driver

Kernel space

GPIOLIB

Bus Infrastructure

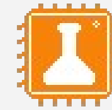
GPIO Driver

LED

GPIO 2.21

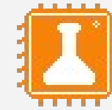
SoC

Hardware



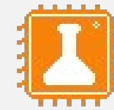
REGISTRANDO O DISPOSITIVO

- Os drivers registram no barramento informações sobre qual dispositivo são capazes de gerenciar, mas eles não contém nenhuma informação sobre o hardware em si.
- Portanto, é necessário um mecanismo para informar ao barramento correspondente quais dispositivos estão conectados no barramento.
- Desta forma, quando um dispositivo for registrado no barramento, será feito o match ou binding com o driver correspondente, e a função `probe()` do driver será chamada.



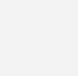
REGISTRANDO O DISPOSITIVO (cont.)

- Como registrar um dispositivo?
  - Um dispositivo pode ser registrado dinamicamente pelo barramento se este possuir suporte à enumeração de dispositivos (Ex: USB e PCI).
  - Um dispositivo pode ser registrado estaticamente através de uma API disponibilizada pelo núcleo do barramento, como por exemplo `i2c_register_board_info()` ou `platform_device_register()`.
  - Através do device tree, suportado em algumas arquiteturas como PPC e ARM (mecanismo padrão nas versões mais atuais do kernel).



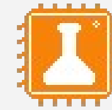
REGISTRANDO NO CÓDIGO

```
/* arch/arm/mach-mx5/board-mx53_loco.c */

static struct i2c_board_info mxc_i2c1_board_info[] __initdata = {
    {
        .type = "mma8450",
        .addr = 0x10,
    },
};

static void __init mx53_loco_board_init(void)
{
    [...]
    i2c_register_board_info(1, mxc_i2c1_board_info,
                            ARRAY_SIZE(mxc_i2c1_board_info));
    [...]
}
```



REGISTRANDO NO CÓDIGO (cont.)

- Descrever os dispositivos de hardware no código traz muitas desvantagens, dentre elas:
  - Muito código duplicado (diferentes implementações para diferentes plataformas/placas).
  - Qualquer alteração de hardware (ex: adicionar um novo dispositivo SPI ou alterar o endereço de um dispositivo I2C) requer alteração do código e recompilação do kernel.
  - Fica difícil liberar uma mesma imagem do kernel que funcione em diferentes plataformas de hardware.

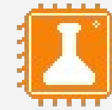


DEVICE TREE

- É necessário então um mecanismo para identificar a topologia e a configuração do hardware (CPU, memória e dispositivos de I/O) em tempo de execução.

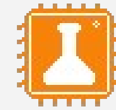
- Este problema foi solucionado com o device tree, uma estrutura de dados capaz de descrever de forma hierárquica o hardware presente no sistema.

- O device tree é passado para o kernel no boot, para que ele possa identificar a topologia e a configuração do hardware, carregar os drivers correspondentes e inicializar o sistema.



DEVICE TREE NA PRÁTICA

- Cada placa possui um arquivo de especificação do device tree (arquivos com extensão DTS).
- Uma ferramenta chamada Device Tree Compiler (`dtc`) é responsável por compilar o device tree (DTS) e gerar uma versão binária (DTB).
- O bootloader é normalmente o responsável por passar o DTB para o kernel no boot.



Embedded Labworks

Hands-on

Device tree

A FLEXIBILIDADE DO DRIVER MODEL

/sys/class/leds/...

User space

LED Framework

Device Tree

LED Driver

Kernel space

```
LED {
    <&gpio2 21>
}
```

GPIOLIB

Platform Core

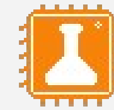
GPIO SoC Driver

LED

GPIO 2.21

SoC

Hardware



A FLEXIBILIDADE DO DRIVER MODEL (cont.)

/sys/class/leds/...

LED Framework

Device Tree

LED Driver

GPIO Expander Driver

```
gpioexp {
    I2C0, 0x10
}
LED {
    <&gpioexp 3>
}
```

Platform Core

GPIOLIB

I2C Core

GPIO SoC Driver

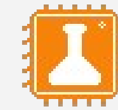
I2C SoC Adapter

LED

GPIO Expander

I2C

SoC



A FLEXIBILIDADE DO DRIVER MODEL (cont.)

/sys/class/leds/...

LED Framework

Device Tree

LED Driver

GPIO Expander Driver

gpioexp {
I2C1, 0x10
}
LED {
<&gpioexp 3>
}

GPIOLIB

I2C Core

Platform Core

GPIO SoC Driver

I2C SoC Adapter

I2C GPIO Adapter

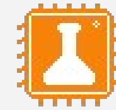
LED

GPIO Expander

I2C Bit Banging

SoC

PERGUNTAS?



OBRIGADO!

E-mail sergio.prado@e-labworks.com
Website http://e-labworks.com

Embedded **Labworks**

Por Sergio Prado. São Paulo, Agosto de 2018
® Copyright Embedded Labworks 2004-2018. All rights reserved.