# Linux Kernel

## Peripheral Devices for Embedded Systems

Rafal Kapela

June 26, 2016

## Outline

## Outline

## The need for a device model?

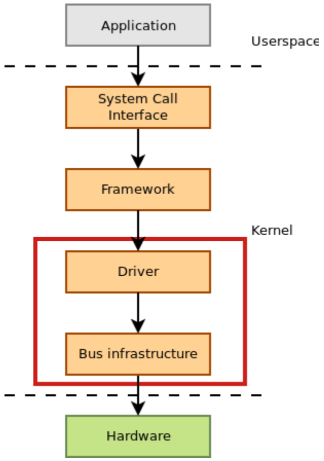
- The Linux kernel runs on a wide range of architectures and hardware platforms, and therefore needs to **maximize the reusability** of code between platforms.
- For example, we want the same *USB device driver* to be usable on a x86 PC, or an ARM platform, even though the USB controllers used on those platforms are different.
- This requires a clean organization of the code, with the *device drivers* separated from the *controller drivers*, the hardware description separated from the drivers themselves, etc.
- This is what the Linux kernel **Device Model** allows, in addition to other advantages covered in this section.

## Kernel and Device Drivers

In Linux, a driver is always interfacing with:

- a **framework** that allows the driver to expose the hardware features in a generic way.
- a **bus infrastructure**, part of the device model, to detect/communicate with the hardware.

This section focuses on the *device model*, while *kernel frameworks* are covered later in this training.



## Device Model data structures

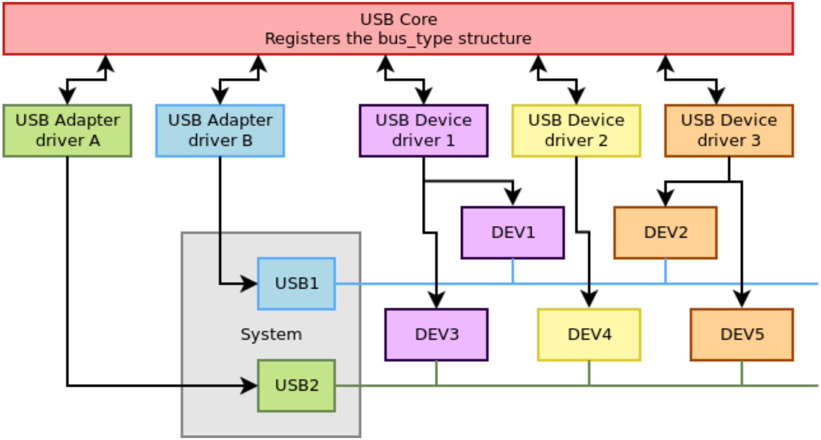
- The *device model* is organized around three main data structures:
  - The bus_type structure, which represent one type of bus (USB, PCI, I2C, etc.)
  - The device_driver structure, which represents one driver capable of handling certain devices on a certain bus.
  - The device structure, which represents one device connected to a bus
- The kernel uses inheritance to create more specialized versions of device_driver and device for each bus subsystem.
- We will then explore the device model.

## Bus Drivers

- The first component of the device model is the bus driver
  - One bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.
- It is responsible for
  - Registering the bus type (bus_type)
  - Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able of detecting the connected devices, and providing a communication mechanism with the devices
  - Allowing the registration of device drivers (USB devices, I2C devices, PCI devices, etc.), managing the devices
  - Matching the device drivers against the devices detected by the adapter drivers.
  - Provides an API to both adapter drivers and device drivers
  - Defining driver and device specific structures

## Outline

## Example: USB Bus 1/2



## Example: USB Bus 2/2

- Core infrastructure (bus driver)
    - drivers/usb/core
    - bus_type is defined in drivers/usb/core/driver.c and registered in drivers/usb/core/usb.c
- Adapter drivers
    - drivers/usb/host
    - For EHCI, UHCI, OHCI, XHCI, and their implementations on various systems (Atmel, IXP, Xilinx, OMAP, Samsung, PXA, etc.)
- Device drivers
    - Everywhere in the kernel tree, classified by their type

## Example of Device Driver

- To illustrate how drivers are implemented to work with the device model, we will study the source code of a driver for a USB network card
  - It is USB device, so it has to be a USB device driver
  - It is a network device, so it has to be a network device
  - Most drivers rely on a bus infrastructure (here, USB) and register themselves in a framework (here, network)
- We will only look at the device driver side, and not the adapter driver side
- The driver we will look at is drivers/net/usb/rtl8150.c

## Device Identifiers

- Defines the set of devices that this driver can manage, so that the USB core knows for which devices this driver should be used
- The MODULE_DEVICE_TABLE macro allows depmod to extract at compile time the relation between device identifiers and drivers, so that drivers can be loaded automatically by udev. See /lib/modules/$(uname -r)/modules.alias,usbmap

```
static struct usb_device_id rtl8150_table[] = {
  {USB_DEVICE(VENDOR_ID_REALTEK,
      PRODUCT_ID_RTL8150)}, ...
};
MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

## Instantiation of usb_driver

- usb_driver is a structure defined by the USB core. Each USB device driver must instantiate it, and register itself to the USB core using this structure
- This structure inherits from driver, which is defined by the device model.

```
static struct usb_driver rtl8150_driver = {
    .name = "rtl8150",
    .probe = rtl8150_probe,
    .disconnect = rtl8150_disconnect,
    .id_table = rtl8150_table,
    .suspend = rtl8150_suspend,
    .resume = rtl8150_resume
};
```

## Driver (Un)Registration

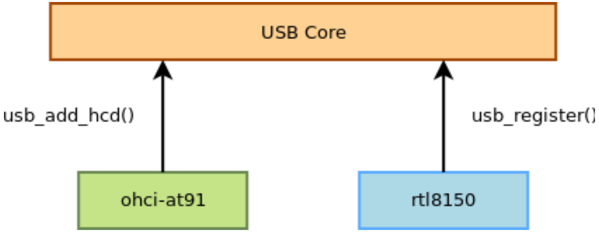
- When the driver is loaded or unloaded, it must register or unregister itself from the USB core
- Done using usb_register and usb_deregister, provided by the USB core.

```
static int __init usb_rtl8150_init(void) {
    return usb_register(&rtl8150_driver);
}
static void __exit usb_rtl8150_exit(void) {
    usb_deregister(&rtl8150_driver);
}
module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

- Note: this code has now been replaced by a shorter module_usb_driver macro call.

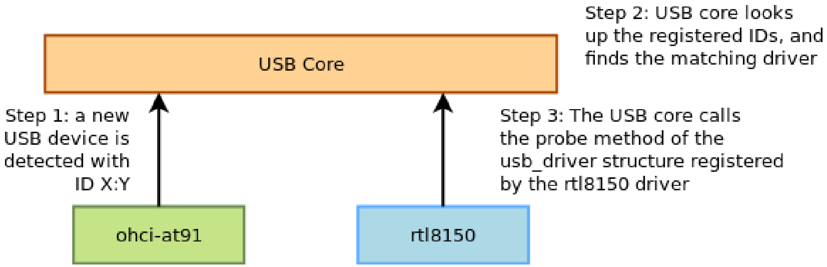
## At Initialization

- The USB adapter driver that corresponds to the USB controller of the system registers itself to the USB core
- The rtl8150 USB device driver registers itself to the USB core



- The USB core now knows the association between the vendor/product IDs of rtl8150 and the usb_driver structure of this driver

## When a Device is Detected


Step 2: USB core looks up the registered IDs, and finds the matching driver
USB Core
Step 1: a new USB device is detected with ID X:Y
Step 3: The USB core calls the probe method of the usb_driver structure registered by the rtl8150 driver
ohci-at91
rtl8150

## Probe Method

- The probe() method receives as argument a structure describing the device, usually specialized by the bus infrastructure (pci_dev, usb_interface, etc.)
- This function is responsible for
  - Initializing the device, mapping I/O memory, registering the interrupt handlers. The bus infrastructure provides methods to get the addresses, interrupt numbers and other device-specific information.
  - Registering the device to the proper kernel framework, for example the network infrastructure.

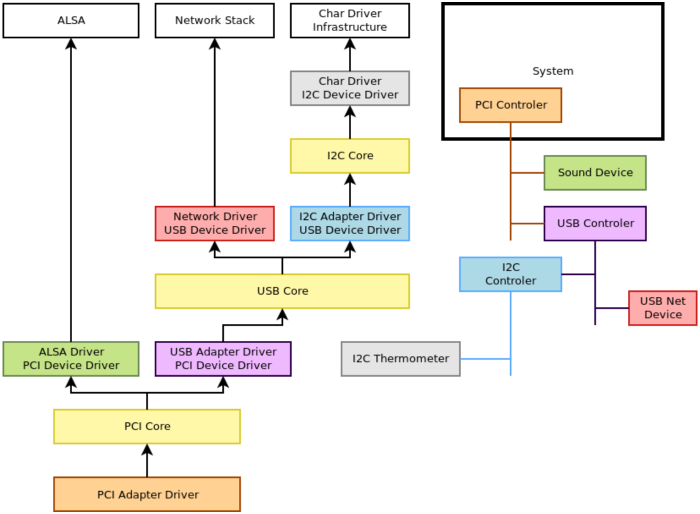
## Probe Method Example

```
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

    netdev = alloc_etherdev(sizeof(rtl8150_t));
    [...]
    dev = netdev_priv(netdev);
    tasklet_init(&dev->tl, rx_fixup, (unsigned long)dev);
    spin_lock_init(&dev->rx_pool_lock);
    [...]
    netdev->netdev_ops = &rtl8150_netdev_ops;
    alloc_all_urbs(dev);
    [...]
    usb_set_intfdata(intf, dev);
    SET_NETDEV_DEV(netdev, &intf->dev);
    register_netdev(netdev);

    return 0;
}
```

## The Model is Recursive


ALSA
Network Stack
Char Driver Infrastructure
Char Driver I2C Device Driver
I2C Core
Network Driver USB Device Driver
I2C Adapter Driver USB Device Driver
USB Core
ALSA Driver PCI Device Driver
USB Adapter Driver PCI Device Driver
PCI Core
PCI Adapter Driver
System
PCI Controler
Sound Device
USB Controler
I2C Controler
USB Net Device
I2C Thermometer

## Outline

## Non-discoverable busses

- On embedded systems, devices are often not connected through a bus allowing enumeration, hotplugging, and providing unique identifiers for devices.
- For example, the devices on I2C busses or SPI busses, or the devices directly part of the system-on-chip.
- However, we still want all of those devices to be part of the device model.
- Such devices, instead of being dynamically detected, must be statically described in either:
    - The kernel source code
    - The *Device Tree*, a hardware description file used on some architectures.

## Platform devices

- Amongst the non-discoverable devices, a huge family are the devices that are directly part of a system-on-chip: UART controllers, Ethernet controllers, SPI or I2C controllers, graphic or audio devices, etc.
- In the Linux kernel, a special bus, called the **platform bus** has been created to handle such devices.
- It supports **platform drivers** that handle **platform devices**.
- It works like any other bus (USB, PCI), except that devices are enumerated statically instead of being discovered dynamically.

## Implementation

- The driver implements a platform_driver structure (example taken from drivers/serial/imx.c)

```
static struct platform_driver serial_imx_driver = {
    .probe = serial_imx_probe,
    .remove = serial_imx_remove,
    .driver = { .name = "imx-uart",
                .owner = THIS_MODULE,
    },
};
```

- And registers its driver to the platform driver infrastructure

```
static int __init imx_serial_init(void) {
    ret = platform_driver_register(&serial_imx_driver);
}
static void __exit imx_serial_cleanup(void) {
    platform_driver_unregister(&serial_imx_driver);
}
```

## Platform Device Instantiation

- As platform devices cannot be detected dynamically, they are defined statically
  - By direct instantiation of platform_device structures, as done on some ARM platforms. Definition done in the board-specific or SoC specific code.
  - By using a *device tree*, as done on Power PC (and on some ARM platforms) from which platform_device structures are created
- Example on ARM, where the instantiation is done in arch/arm/mach-imx/mx1ads.c

```
static struct platform_device imx_uart1_device = {
    .name = "imx-uart",
    .id = 0, .num_resources = ARRAY_SIZE(imx_uart1_resources),
    .resource = imx_uart1_resources,
    .dev = { .platform_data = &uart_pdata, } };
```

## Platform device instantiation

- The device is part of a list

```
static struct platform_device *devices[] __initdata = {
    &cs89x0_device,
    &imx_uart1_device,
    &imx_uart2_device, };
```

- And the list of devices is added to the system during board initialization

```
static void __init mx1ads_init(void) {
    [...]
    platform_add_devices(devices, ARRAY_SIZE(devices));
}
MACHINE_START(MX1ADS, "Freescale MX1ADS")
    [...]
    .init_machine = mx1ads_init,
MACHINE_END
```

## The Resource Mechanism

- Each device managed by a particular driver typically uses different hardware resources: addresses for the I/O registers, DMA channels, IRQ lines, etc.
- Such information can be represented using resource, and an array of resource is associated to a platform_device
- Allows a driver to be instantiated for multiple devices functioning similarly, but with different addresses, IRQs, etc.

## Declaring resources

```
static struct resource imx_uart1_resources[] = {
    [0] = {
        .start = 0x00206000,
        .end = 0x002060FF,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = (UART1_MINT_RX),
        .end = (UART1_MINT_RX),
        .flags = IORESOURCE_IRQ,
    },
};
```

## Using Resources

- When a platform_device is added to the system using platform_add_device, the probe() method of the platform driver gets called
- This method is responsible for initializing the hardware, registering the device to the proper framework (in our case, the serial driver framework)
- The platform driver has access to the I/O resources:

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
sport->rxirq = platform_get_irq(pdev, 0);
```

## platform_data Mechanism

- In addition to the well-defined resources, many drivers require driver-specific information for each platform device
- Such information can be passed using the platform_data field of device (from which platform_device inherits)
- As it is a void * pointer, it can be used to pass any type of information.
  - Typically, each driver defines a structure to pass information through platform_data

## platform_data example 1/2

- The i.MX serial port driver defines the following structure to be passed through platform_data

```
struct imxuart_platform_data {
    int (*init)(struct platform_device *pdev);
    void (*exit)(struct platform_device *pdev);
    unsigned int flags;
    void (*irda_enable)(int enable);
    unsigned int irda_inv_rx:1;
    unsigned int irda_inv_tx:1;
    unsigned short transceiver_delay;
};
```

- The MX1ADS board code instantiates such a structure

```
static struct imxuart_platform_data uart1_pdata = {
    .flags = IMXUART_HAVE_RTSCTS,
};
```

## platform_data Example 2/2

- The uart_pdata structure is associated to the platform_device structure in the MX1ADS board file (the real code is slightly more complicated)

```
struct platform_device mx1ads_uart1 = {
    .name = "imx-uart",
    .dev {
        .platform_data = &uart1_pdata,
    },
    .resource = imx_uart1_resources,
    [...]
};
```

- The driver can access the platform data:

```
static int serial_imx_probe(struct platform_device *pdev) {
    struct imxuart_platform_data *pdata;
    pdata = pdev->dev.platform_data;
    if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
        sport->have_rtscts = 1;
```

## Device Tree

- On many embedded architectures, manual instantiation of platform devices was considered to be too verbose and not easily maintainable.
- Such architectures are moving, or have moved, to use the *Device Tree*.
- It is a **tree of nodes** that models the hierarchy of devices in the system, from the devices inside the processor to the devices on the board.
- Each node can have a number of **properties** describing various properties of the devices: addresses, interrupts, clocks, etc.
- At boot time, the kernel is given a compiled version, the **Device Tree Blob**, which is parsed to instantiate all the devices described in the DT.

## Device Tree example

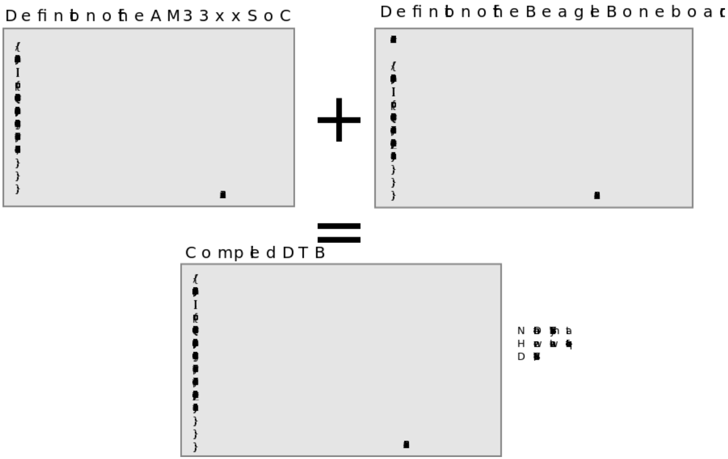
```
uart0: serial@44e09000 {
        compatible = "ti,omap3-uart";
        ti,hwmods = "uart1";
        clock-frequency = <48000000>;
        reg = <0x44e09000 0x2000>;
        interrupts = <72>;
        status = "disabled";
};
```

- serial@44e09000 is the **node name**
- uart0 is an **alias**, that can be referred to in other parts of the DT as &uart0
- other lines are **properties**. Their values are usually strings, list of integers, or references to other nodes.

## Device Tree inheritance (1/2)

- Each particular hardware platform has its own *device tree*.
- However, several hardware platforms use the same processor, and often various processors in the same family share a number of similarities.
- To allow this, a *device tree* file can include another one. The trees described by the including file overlays the tree described by the included file.

## Device Tree inheritance (2/2)


De fi n b n o fi e A M 3 3 x x S o C
De fi n b n o fi e B e a g b B o n e b o a
+
=
C o mp b d D T B
N
H
D

## DT: compatible string

- With the *device tree*, a *device* is bound with the corresponding *driver* using the **compatible** string.
- The of_match_table field of device_driver lists the compatible strings supported by the driver.

```
#if defined(CONFIG_OF)
static const struct of_device_id omap_serial_of_match[] = {
        { .compatible = "ti,omap2-uart" },
        { .compatible = "ti,omap3-uart" },
        { .compatible = "ti,omap4-uart" },
        {},
};
MODULE_DEVICE_TABLE(of, omap_serial_of_match);
#endif
static struct platform_driver serial_omap_driver = {
        .probe          = serial_omap_probe,
        .remove         = serial_omap_remove,
        .driver         = {
                .name   = DRIVER_NAME,
                .pm     = &serial_omap_dev_pm_ops,
                .of_match_table = of_match_ptr(omap_serial_of_match),
        },
};
```

## Device Tree Resources

- The drivers will use the same mechanism that we saw previously to retrieve basic information: interrupts numbers, physical addresses, etc.
- The available resources list will be built up by the kernel at boot time from the device tree, so that you don't need to make any unnecessary lookups to the DT when loading your driver.
- Any additional information will be specific to a driver or the class it belongs to, defining the *bindings*

## Device Tree bindings

- The compatible string and the associated properties define what is called a *device tree binding*.
- *Device tree bindings* are all documented in devicetree/bindings.
- Since the Device Tree is normally part of the kernel ABI, the *bindings* must remain compatible over-time.
  - A new kernel must be capable of using an old Device Tree.
- A Device Tree binding should contain only a *description of the hardware* and not *configuration*.
  - An interrupt number can be part of the Device Tree as it describes the hardware.
  - But not whether DMA should be used for a device or not.

## sysfs

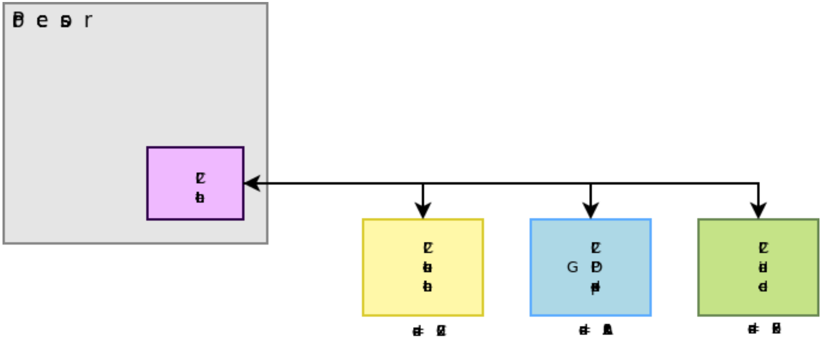
- The bus, device, drivers, etc. structures are internal to the kernel
- The sysfs virtual filesystem offers a mechanism to export such information to userspace
- Used for example by udev to provide automatic module loading, firmware loading, device file creation, etc.
- sysfs is usually mounted in /sys
  - /sys/bus/ contains the list of buses
  - /sys/devices/ contains the list of devices
  - /sys/class enumerates devices by class (net, input, block...), whatever the bus they are connected to. Very useful!
- Take your time to explore /sys on your workstation.

## Outline

## What is I2C?

- A very commonly used low-speed bus to connect on-board devices to the processor.
- Uses only two wires: SDA for the data, SCL for the clock.
- It is a master/slave bus: only the master can initiate transactions, and slaves can only reply to transactions initiated by masters.
- In a Linux system, the I2C controller embedded in the processor is typically the master, controlling the bus.
- Each slave device is identified by a unique I2C address. Each transaction initiated by the master contains this address, which allows the relevant slave to recognize that it should reply to this particular transaction.

## An I2C bus example



## The I2C subsystem

- Like all bus subsystems, the I2C subsystem is responsible for:
  - Providing an API to implement I2C controller drivers
  - Providing an API to implement I2C device drivers, in kernel space
  - Providing an API to implement I2C device drivers, in user space
- The core of the I2C subsystem is located in drivers/i2c.
- The I2C controller drivers are located in drivers/i2c/busses.
- The I2C device drivers are located throughout drivers/, depending on the type of device (ex: drivers/input for input devices).

## Registering an I2C device driver

- Like all bus subsystems, the I2C subsystem defines a i2c_driver that inherits from device_driver, and which must be instantiated and registered by each I2C device driver.
  - As usual, this structure points to the -¿probe() and remove() functions.
  - It also contains an id_table field that must point to a list of *device IDs* (which is a list of tuples containing a string and some private driver data). It is used for non-DT based probing of I2C devices.
- The i2c_add_driver and i2c_del_driver functions are used to register/unregister the driver.
- If the driver doesn't do anything else in its init()/exit() functions, it is advised to use the module_i2c_driver macro instead.

## Registering an I2C device driver

```
static const struct i2c_device_id <driver>_id[] = {
        { "<device-name>", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, <driver>_id);

#ifdef CONFIG_OF
static const struct of_device_id <driver>_dt_ids[] = {
        { .compatible = "<vendor>,<device-name>", },
        { }
};
MODULE_DEVICE_TABLE(of, <driver>_dt_ids);
#endif

static struct i2c_driver <driver>_driver = {
        .probe          = <driver>_probe,
        .remove         = <driver>_remove,
        .id_table       = <driver>_id,
        .driver = {
                .name   = "<driver-name>",
                .owner  = THIS_MODULE,
                .of_match_table = of_match_ptr(<driver>_dt_ids),
        },
};

module_i2c_driver(<driver>_driver);
```

## Registering an I2C dev.: non-DT

- On non-DT platforms, the i2c_board_info structure allows to describe how an I2C device is connected to a board.
- Such structures are normally defined with the I2C_BOARD_INFO helper macro.
  - Takes as argument the device name and the slave address of the device on the bus.
- An array of such structures is registed on a per-bus basis using i2c_register_board_info, when the platform is initialized.

## Registering an I2C dev.: non-DT

```
static struct i2c_board_info <board>_i2c_devices[] __initdata = {
        {
                I2C_BOARD_INFO("cs42l51", 0x4a),
        },
};

void board_init(void)
{
        /*
         * Here should be the registration of all devices, including
         * the I2C controller device.
         */

        i2c_register_board_info(0, <board>_i2c_devices,
                                ARRAY_SIZE(<board>_i2c_devices));

        /* More devices registered here */
}
```

## Registering an I2C dev.: the DT

- In the Device Tree, the I2C controller device is typically defined in the .dtsi file that describes the processor.
  - Normally defined with status = "disabled".
- At the board/platform level:
  - the I2C controller device is enabled (status = "okay")
  - the I2C bus frequency is defined, using the clock-frequency property.
  - the I2C devices on the bus are described as children of the I2C controller node, where the reg property gives the I2C slave address on the bus.

## Registering an I2C dev.: DT

### Definition of the I2C controller, .dtsi file

```
i2c@7000c000 {
        compatible = "nvidia,tegra20-i2c";
        reg = <0x7000c000 0x100>;
        interrupts = <GIC_SPI 38 IRQ_TYPE_LEVEL_HIGH>;
        #address-cells = <1>;
        #size-cells = <0>;
        clocks = <&tegra_car TEGRA20_CLK_I2C1>,
                 <&tegra_car TEGRA20_CLK_PLL_P_OUT3>;
        clock-names = "div-clk", "fast-clk";
        status = "disabled";
};
```

## Registering an I2C dev.: DT

### Definition of the I2C device, .dts file

```
i2c@7000c000 {
        status = "okay";
        clock-frequency = <400000>;

        alc5632: alc5632@1e {
                compatible = "realtek,alc5632";
                reg = <0x1e>;
                gpio-controller;
                #gpio-cells = <2>;
        };
};
```

## probe() and remove()

- The probe() function is responsible for initializing the device and registering it in the appropriate kernel framework. It receives as argument:
  - A i2c_client pointer, which represents the I2C device itself. This structure inherits from device.
  - A i2c_device_id pointer, which points to the I2C device ID entry that matched the device that is being probed.
- The remove() function is responsible for unregistering the device from the kernel framework and shut it down. It receives as argument:
  - The same i2c_client pointer that was passed as argument to probe()

## Probe/remove example

```
static int <driver>_probe(struct i2c_client *client,
                          const struct i2c_device_id *id)
{
        /* initialize device */
        /* register to a kernel framework */

        i2c_set_clientdata(client, <private data>);
        return 0;
}

static int <driver>_remove(struct i2c_client *client)
{
        <private data> = i2c_get_clientdata(client);
        /* unregister device from kernel framework */
        /* shut down the device */
        return 0;
}
```

## Communicating with the I2C dev

The most **basic API** to communicate with the I2C device provides functions to either send or receive data:

- int i2c_master_send(struct i2c_client *client, const char *buf, int count);
  Sends the contents of buf to the client.
- int i2c_master_recv(struct i2c_client *client, char *buf, int count);
  Receives count bytes from the client, and store them into buf.

## Communicating with the I2C dev

The message transfer API allows to describe **transfers** that consists of several **messages**, with each message being a transaction in one direction:

- int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg, int num);
- The i2c_adapter pointer can be found by using client-adapter
- The i2c_msg structure defines the length, location, and direction of the message.

## I2C: message transfer example

```
struct i2c_msg msg[2];
int error;
u8 start_reg;
u8 buf[10];

msg[0].addr = client->addr;
msg[0].flags = 0;
msg[0].len = 1;
msg[0].buf = &start_reg;
start_reg = 0x10;

msg[1].addr = client->addr;
msg[1].flags = I2C_M_RD;
msg[1].len = sizeof(buf);
msg[1].buf = buf;

error = i2c_transfer(client->adapter, msg, 2);
```

## SMBus calls

- SMBus is a subset of the I2C protocol.
- It defines a standard set of transactions, for example to read or write a register into a device.
- Linux provides SMBus functions that *should be used* when possible instead of the raw API, if the I2C device uses this standard type of transactions.
- Example: the i2c_smbus_read_byte_data function allows to read one byte of data from a device register.
  - It does the following operations: S Addr Wr [A] Comm [A] S Addr Rd [A] [Data] NA P
  - Which means it first writes a one byte data command (*Comm*), and then reads back one byte of data (*[Data]*).
- See i2c/smbus-protocol for details.

## List of SMBus functions

- Read/write one byte
  - s32 i2c_smbus_read_byte(const struct i2c_client *client);
  - s32 i2c_smbus_write_byte(const struct i2c_client *client, u8 value);
- Write a command byte, and read or write one byte
  - s32 i2c_smbus_read_byte_data(const struct i2c_client *client, u8 command);
  - s32 i2c_smbus_write_byte_data(const struct i2c_client *client, u8 command, u8 value);
- Write a command byte, and read or write one word
  - s32 i2c_smbus_read_word_data(const struct i2c_client *client, u8 command);
  - s32 i2c_smbus_write_word_data(const struct i2c_client *client, u8 command, u16 value);
- Write a command byte, and read or write a block of data (max 32 bytes)
  - s32 i2c_smbus_read_block_data(const struct i2c_client *client, u8 command, u8 *values);
  - s32 i2c_smbus_write_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values);
- Write a command byte, and read or write a block of data (no limit)
  - s32 i2c_smbus_read_i2c_block_data(const struct i2c_client *client, u8 command, u8 length, u8 *values);
  - s32 i2c_smbus_write_i2c_block_data(const struct i2c_client *client, u8 command, u8 length, const u8 *values);

## I2C functionality

- Not all I2C controllers support all functionalities.
- The I2C controller drivers therefore tell the I2C core which functionalities they support.
- An I2C device driver must check that the functionalities they need are provided by the I2C controller in use on the system.
- The i2c_check_functionality function allows to make such a check.
- Examples of functionalities: I2C_FUNC_I2C to be able to use the raw I2C functions, I2C_FUNC_SMBUS_BYTE_DATA to be able to use SMBus commands to write a command and read/write one byte of data.
- See include/uapi/linux/i2c.h for the full list of existing functionalities.

## Resources

If you want to gain some knowledge by your own...

- Wikipedia – Embedded system
  `http://en.wikipedia.org/wiki/Embedded_system`
- Greg Kroah-Hartman, O'Reilly, Linux Kernel in a Nutshell, Dec 2006.
  `http://www.kroah.com/lkn/`
- Free Electrons - embedded Linux experts
  `http://free-electrons.com/`

Questions ?

Rafal Kapela
rafal.kapela@put.poznan.pl