Introduction to Linux kernel driver programming

The Linux kernel device model

Authors and license

- Authors
  - Michael Opdenacker,
    Thomas Petazzoni,
    Bootlin: kernel and embedded Linux engineering company
    https://bootlin.com
- License
  -
  - Document sources: https://github.com/e-ale/Slides

bootlin

OSS Essentials

Need for a device model

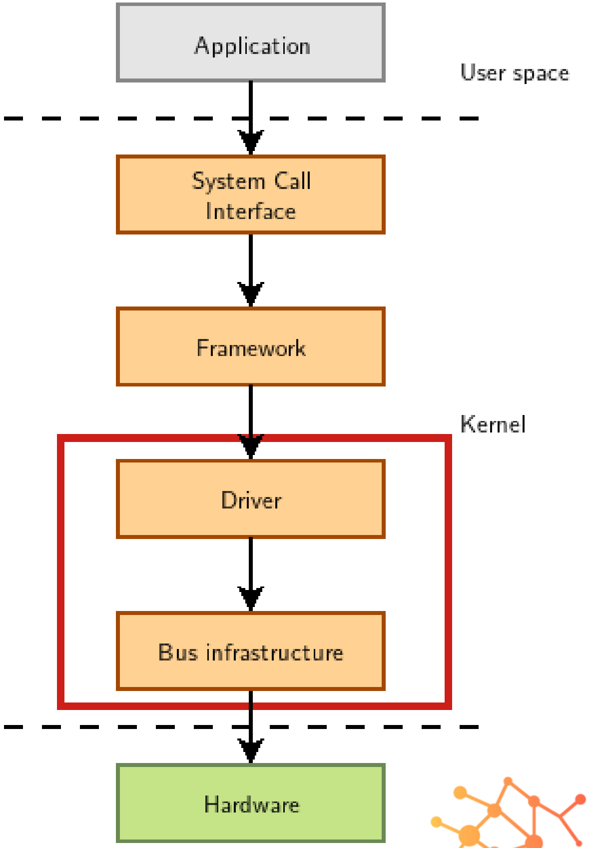
- For the same device, need to use the **same device driver** on multiple CPU architectures (x86, ARM…), even though the hardware controllers are different.
- Need for a single driver to **support multiple devices** of the same kind.
- This requires a clean organization of the code, with the *device drivers* separated from the *controller drivers*, the hardware description separated from the drivers themselves, etc.

Driver: between bus infrastructure and framework

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.

Let’s focus on the bus infrastructure for now



bootlin

OSS Essentials

Device model data structures

The device model is organized around three main data structures:

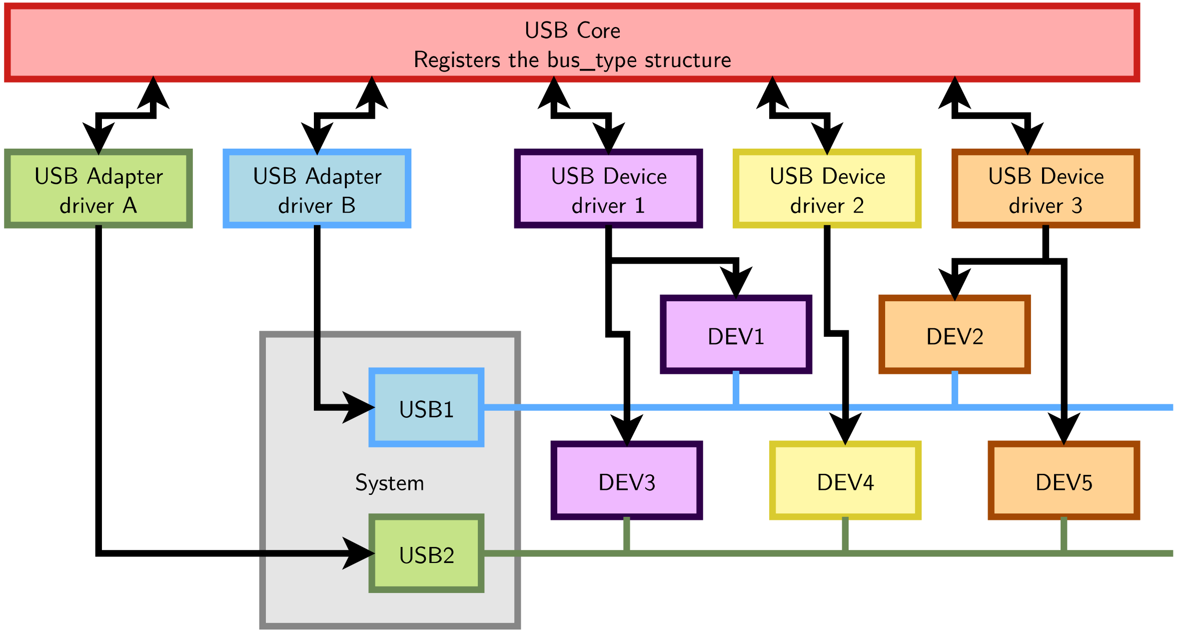
- The `struct bus_type` structure, which represent one type of bus (USB, PCI, I2C, etc.)
- The `struct device_driver` structure, which represents one driver capable of handling certain devices on a certain bus.
- The `struct device` structure, which represents one device connected to a bus

The kernel uses inheritance to create more specialized versions of `struct device_driver` and `struct device` for each bus subsystem.

The bus driver

- Example: USB. Implemented in `drivers/usb/core/`
- Creates and registers the `bus_type` structure
- Provides an API to register and implement adapter drivers (here USB controllers), able to detect the connected devices and allowing to communicate with them.
- Provides an API to register and implement device drivers (here USB device drivers)
- Matches the device drivers against the devices detected by the adapter drivers.
- Defines driver and device specific structures, here mainly `struct usb_driver` and `struct usb_interface`

USB bus example



A single driver for compatible devices, though connected to buses with different controllers.

Device drivers (1)

Need to **register supported devices** to the bus core.

Example: `drivers/net/usb/rtl8150.c`

```
static struct usb_device_id rtl8150_table[] =
{{ USB_DEVICE(VENDOR_ID_REALTEK, PRODUCT_ID_RTL8150) },
{ USB_DEVICE(VENDOR_ID_MELCO, PRODUCT_ID_LUAKTX) },
{ USB_DEVICE(VENDOR_ID_MICRONET, PRODUCT_ID_SP128AR) },
{ USB_DEVICE(VENDOR_ID_LONGSHINE, PRODUCT_ID_LCS8138TX) },[…]
{}
};
MODULE_DEVICE_TABLE(usb, rtl8150_table);
```

Device drivers (2)

Need to register **hooks to manage devices** (newly detected or removed ones), as well as to react to power management events (suspend and resume)

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

Device drivers (3)

The last step is to **register the driver structure to the bus 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.

Now the bus driver knows the association between the devices and the device driver.

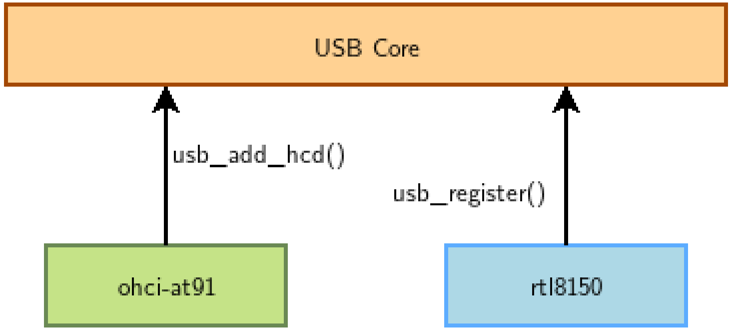
Work in the probe() function

`probe()` is called for each newly matched device

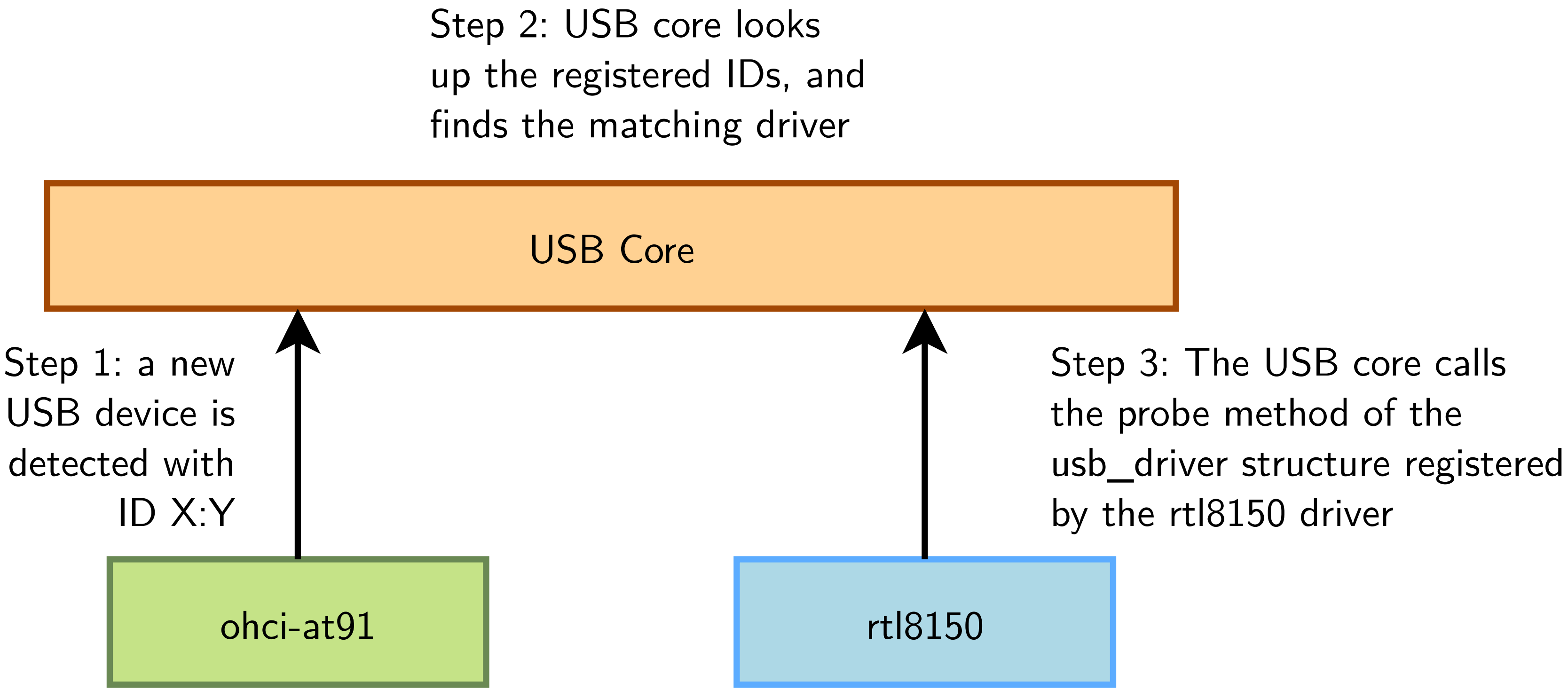
- Initialize the device
- Prepare driver work: allocate a structure for a suitable framework, allocate memory, map I/O memory, register interrupts…
- When everything is ready, register the new device to the framework.

At driver loading time

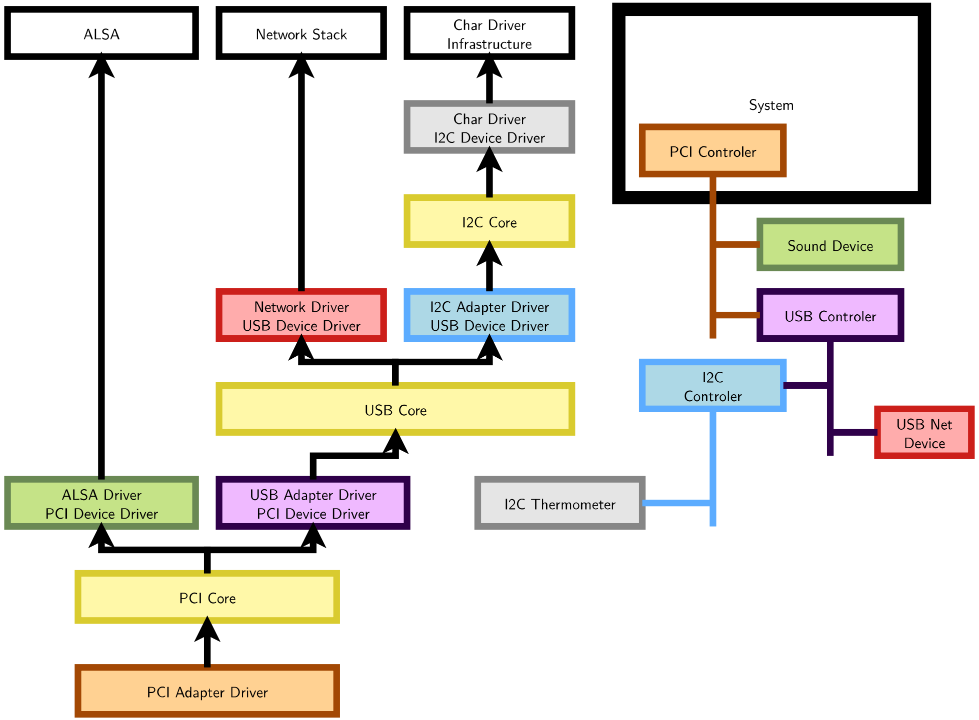
- The USB adapter driver that corresponds to the USB controller registers itself to the USB core
- The `rtl8150` USB device driver registers itself to the USB core
- The USB core now knows the association between the vendor/product IDs of `rtl8150` and the `struct usb_driver` structure of this driver



When a device is detected



The model is recursive



Adapter drivers are device drivers too!

Platform devices and drivers

- Want to use the Device Model for devices that are not on buses that can auto-detect devices (very frequent in embedded systems)
- Examples: UARTs, flash memory, LEDs, GPIOs, MMC/SD, Ethernet…
- Solution:

1) Provide a description of devices
2) Manage them through a virtual bus: the `platform` bus.
3) Drive the platform devices

Describing non-detectable devices

- Description through a Device Tree (on ARM, PowerPC, ARC…)
- In `arch/arm/boot/dts/` on ARM
- Two parts:
  - Device Tree Source (`.dts`)
    One per board to support in the Linux kernel
    Advantage: no need to write kernel code to support a new board (if all devices are supported).
  - Device Tree Source Includes (`.dtsi`)
    Typically to describe devices on a particular SoC, or devices shared between similar SoCs or boards
- Other method for describing non-detectable devices: ACPI (on x86 platforms). Not covered here.

Declaring a device: .dtsi example

Label

Node name

Node properties

```
...
    i2c0: i2c@44e0b000 {
        compatible = "ti,omap4-i2c";
        #address-cells = <1>;
        #size-cells = <0>;
        ti,hwmods = "i2c1";
        reg = <0x44e0b000 0x1000>;
        interrupts = <70>;
        status = "disabled";
    };
...
```

Compatible drivers

HW register start address and range

Present but not used by default

From arch/arm/boot/dts/am33xx.dtsi

Instantiating a device: .dts example

```
&i2c0 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pins>;

        status = "okay";
        clock-frequency = <400000>;

        tps: tps@24 {
                reg = <0x24>;
        };

        baseboard_eeprom: baseboard_eeprom@50 {
                compatible = "at,24c256";
                reg = <0x50>;

                #address-cells = <1>;
                #size-cells = <1>;
                baseboard_data: baseboard_data@0 {
                        reg = <0 0x100>;
                };
        };
};
```

Phandle (reference to label)

Pin muxing configuration (routing to external package pins)

Enabling this device, otherwise ignored

Node property: frequency

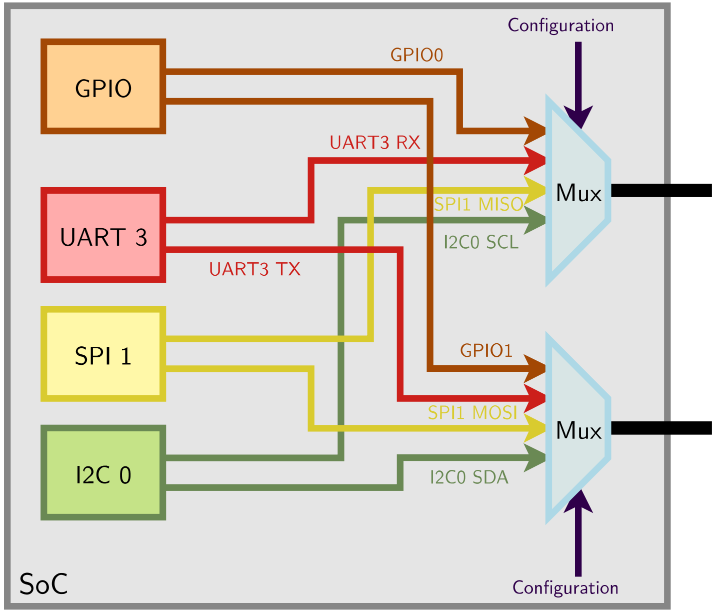
List of devices on i2c0

I2C bus identifier

From arch/arm/boot/dts/am335x-boneblue.dts

Pin multiplexing

- Modern SoCs have too many hardware blocks compared to physical pins exposed on the chip package.
- Therefore, pins have to be multiplexed
- Pin configurations are defined in the Device Tree
- Correct pin multiplexing is mandatory to make a device work from an electronic point of view.



DT pin definitions

```
&am33xx_pinmux {
      ...
      i2c0_pins: pinmux_i2c0_pins {
            pinctrl-single,pins = <
                  AM33XX_IOPAD(0x988, PIN_INPUT_PULLUP | MUX_MODE0)       /* (C17) I2C0_SDA.I2C0_SDA */
                  AM33XX_IOPAD(0x98c, PIN_INPUT_PULLUP | MUX_MODE0)       /* (C16) I2C0_SCL.I2C0_SCL */
            >;
      };
      ...
};
...

&i2c0 {
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pins>;

        status = "okay";
        clock-frequency = <400000>;
        ...
};
```

- Register offset corresponding to a given package pin
- Configures the pin: input, output, drive strengh, pull up/down...
- Allows to select a given SoC signal
- Package pin name
- SoC signal name

From arch/arm/boot/dts/am335x-boneblue.dts

DT: matching devices and drivers

Platform drivers are matched with platform devices that have the same `compatible` property.

```
static const struct of_device_id omap_i2c_of_match[] = {
        {
                .compatible = "ti,omap4-i2c",
                .data = &omap4_pdata,
        },
        {
…
};
…
static struct platform_driver omap_i2c_driver = {
        .probe          = omap_i2c_probe,
        .remove         = omap_i2c_remove,
        .driver         = {
                .name   = "omap_i2c",
                .pm     = OMAP_I2C_PM_OPS,
                .of_match_table = of_match_ptr(omap_i2c_of_match),
        },
};
```

From drivers/i2c/busses/i2c-omap.c

Usage of the platform bus

Like for physical buses, the platform bus is used by the driver to retrieve information about each device

```
static int omap_i2c_probe(struct platform_device *pdev)
{
     ...
     struct device_node      *node = pdev→dev.of_node;
     struct omap_i2c_dev     *omap;
     ...
     irq = platform_get_irq(pdev, 0);

     ...
     omap = devm_kzalloc(&pdev->dev, sizeof(struct omap_i2c_dev), GFP_KERNEL);
     ...
     mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
     omap->base = devm_ioremap_resource(&pdev->dev, mem);
     u32 freq = 100000; /* default to 100000 Hz */

     ...
     of_property_read_u32(node, "clock-frequency", &freq);

     ...
     return 0;
}
```

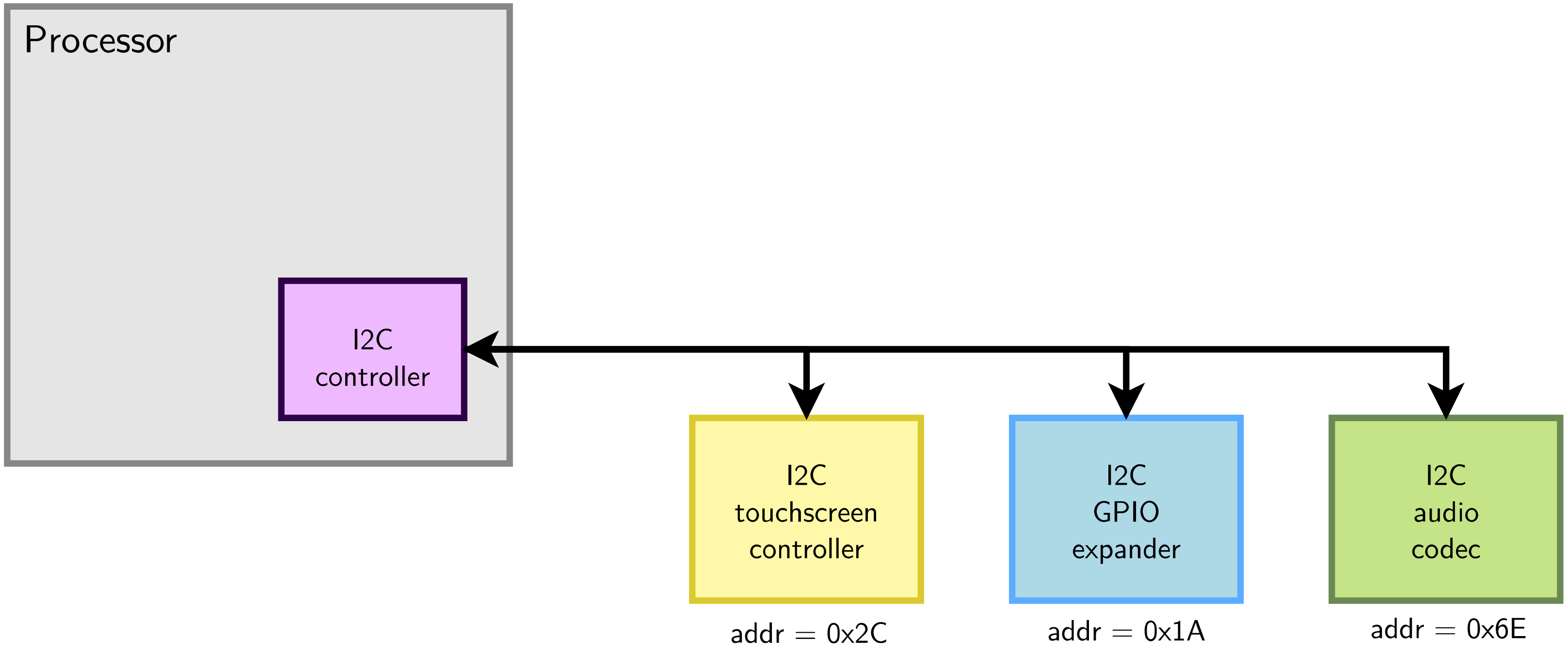
From drivers/i2c/busses/i2c-omap.c

Device tree bindings

- *Device tree bindings* provide a specification of properties that a driver expects in a DT
- Bindings are available in `Documentation/devicetree/bindings` in kernel sources.
- To know how to set device properties, look for a binding for the same compatible string:

```
$ git grep “ti,omap4-i2c” Documentation/devicetree/bindings/
```

Another bus example: I2C



I2C drivers: probe() function

```
static int mma7660_probe(struct i2c_client *client,
                         const struct i2c_device_id *id)
{
        int ret;
        struct iio_dev *indio_dev;
        struct mma7660_data *data;

        indio_dev = devm_iio_device_alloc(&client->dev, sizeof(*data));
        if (!indio_dev) {
                dev_err(&client->dev, "iio allocation failed!\n");
                return -ENOMEM;
        }

        data = iio_priv(indio_dev);
        data->client = client;
        i2c_set_clientdata(client, indio_dev);
        mutex_init(&data->lock);
        data->mode = MMA7660_MODE_STANDBY;

        indio_dev->dev.parent = &client->dev;
        indio_dev->info = &mma7660_info;
        indio_dev->name = MMA7660_DRIVER_NAME;
        indio_dev->modes = INDIO_DIRECT_MODE;
        indio_dev->channels = mma7660_channels;
        indio_dev->num_channels = ARRAY_SIZE(mma7660_channels);

        ret = mma7660_set_mode(data, MMA7660_MODE_ACTIVE);
        if (ret < 0)
                return ret;

        ret = iio_device_register(indio_dev);
        if (ret < 0) {
                dev_err(&client->dev, "device_register failed\n");
                mma7660_set_mode(data, MMA7660_MODE_STANDBY);
        }

        return ret;
}
```

`device` structure for the i2c bus needed to communicate with the device

Framework (here iio) structure for each device

Per device structure. Used by the driver to store references to bus and framework structures, plus its own data (locks, wait queues, etc.)

Allocation of the framework structure. This structure also contains the per device structure (`data`)

Reference to the bus structure stored in the framework structure.

Reference to the framework structure stored in the bus structure.

Enabling the device (i2c reading and writing)

Register a new framework device when everything is ready (device now accessible in user-space)

Disabling the device in case of failure

From drivers/iio/accel/mma7660.c

I2C drivers: remove() function

```
sstatic int mma7660_remove(struct i2c_client *client)
{
        struct iio_dev *indio_dev = i2c_get_clientdata(client);

        iio_device_unregister(indio_dev);

        return mma7660_set_mode(iio_priv(indio_dev),
                MMA7660_MODE_STANDBY);
}
```

Same i2c device structure as in `probe()`

Get back the framework structure. Needed to unregister the framework device from the system

Unregister the framework device from the system

Now that user-space can’t access the device any more, disable the device.

From drivers/iio/accel/mma7660.c

I2C driver registration

```
static const struct i2c_device_id mma7660_i2c_id[] = {
        {"mma7660", 0},
        {}
};
MODULE_DEVICE_TABLE(i2c, mma7660_i2c_id);

static const struct of_device_id mma7660_of_match[] = {
        { .compatible = "fsl,mma7660" },
        { }
};
MODULE_DEVICE_TABLE(of, mma7660_of_match);

static const struct acpi_device_id mma7660_acpi_id[] = {
        {"MMA7660", 0},
        {}
};

MODULE_DEVICE_TABLE(acpi, mma7660_acpi_id);

static struct i2c_driver mma7660_driver = {
        .driver = {
                .name = "mma7660",
                .pm = MMA7660_PM_OPS,
                .of_match_table = mma7660_of_match,
                .acpi_match_table = ACPI_PTR(mma7660_acpi_id),
        },
        .probe          = mma7660_probe,
        .remove         = mma7660_remove,
        .id_table       = mma7660_i2c_id,
};

module_i2c_driver(mma7660_driver);
```

Matching by name (mandatory for I2C)

Matching by compatible property (for DT)

Matching by ACPI ID (for ACPI systems - x86)

From drivers/iio/accel/mma7660.c

Driver development advise

- Look for code for devices similar to yours
- Read the recent code.
  You can use Elixir (https://elixir.bootlin.com/)
- Always read code from the bottom up. You see the big picture first, and then progressively how the details are implemented.

Further reading

- Bootlin's kernel and driver development training materials for full details https://bootlin.com/training/kernel/
- Device Tree for Dummies presentation Thomas Petazzoni (Apr. 2014) http://j.mp/1jQU6NR