Rights to copy

- © Copyright 2004-2019, Bootlin
- **License: Creative Commons Attribution - Share Alike 3.0**
- https://creativecommons.org/licenses/by-sa/3.0/legalcode
- You are free:
  - to copy, distribute, display, and perform the work
  - to make derivative works
  - to make commercial use of the work
- Under the following conditions:
  - Attribution. You must give the original author credit.
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only
- under a license identical to this one.
  - For any reuse or distribution, you must make clear to others the license terms of this work.
  - Any of these conditions can be waived if you get permission from the copyright holder.
- Your fair use and other rights are in no way affected by the above.
- **Document sources:** https://git.bootlin.com/training-materials/

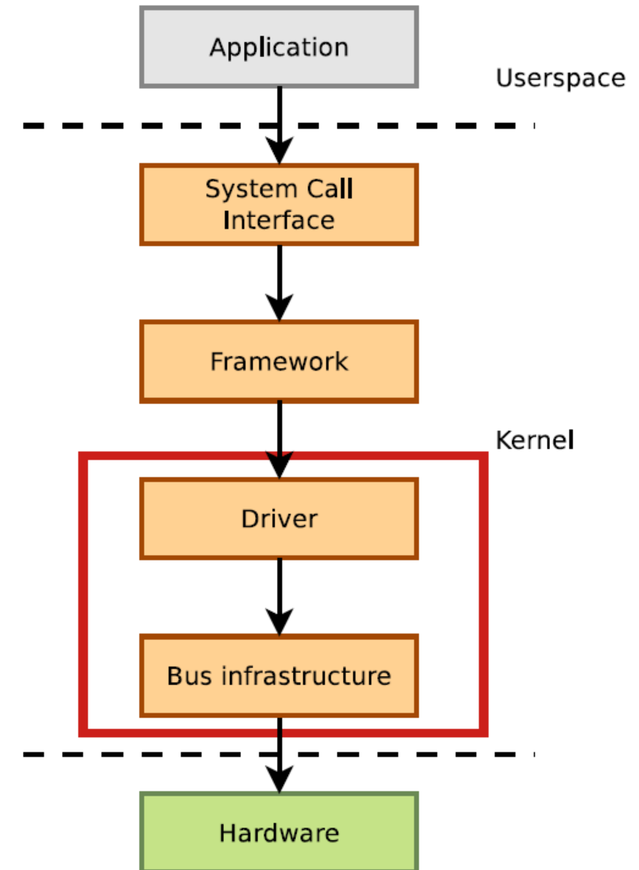
4. Linux Device and Driver Model

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 these 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.

Linux Device and Driver Model

- 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.

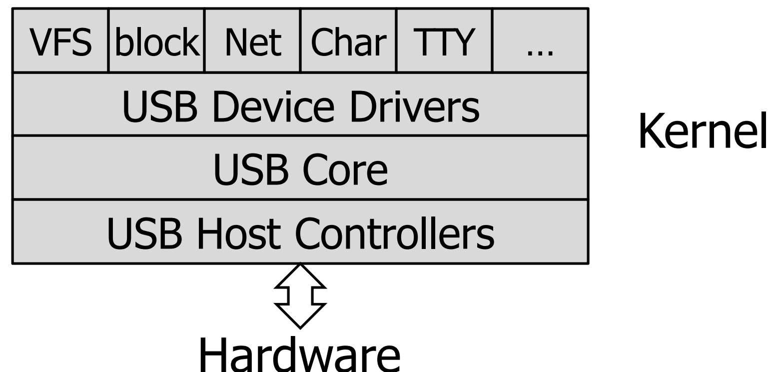


Device Model data structures

- Device model
  - struct bus_type structure: one type of bus (USB, PCI, I2C, etc.)
  - struct device_driver structure: one driver capable of handling certain devices on a certain bus.
  - struct device structure: 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.

Bus Drivers

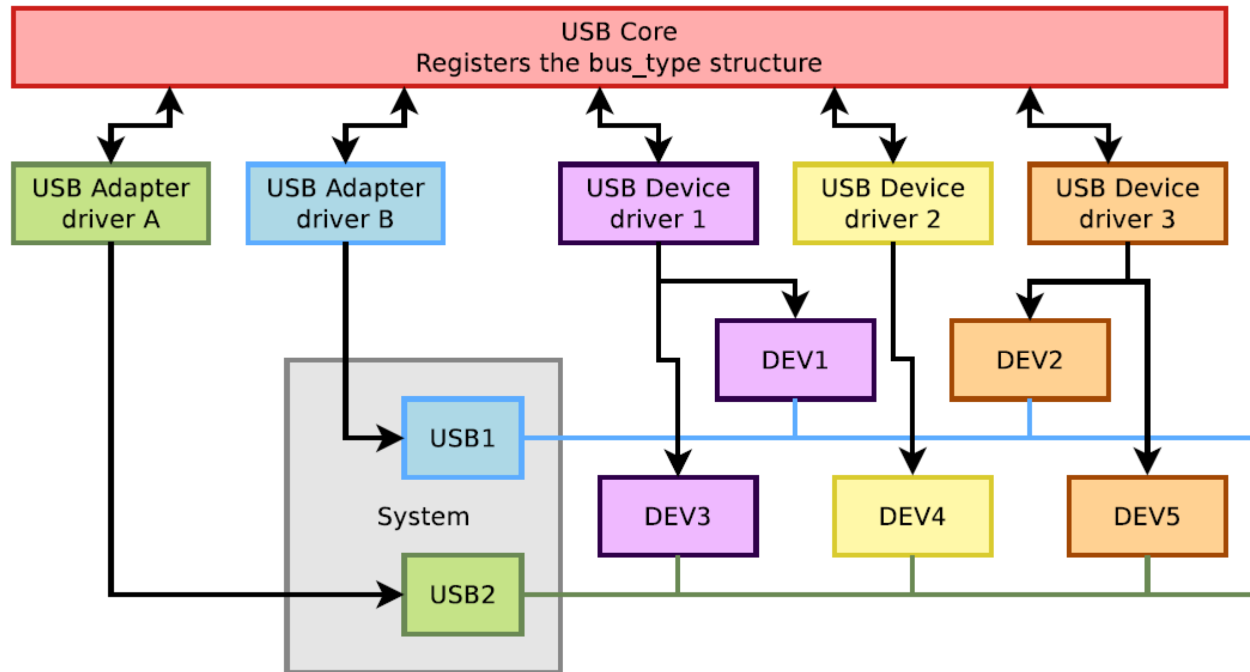
- One bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.
- It is responsible for
  - Registering the bus type (struct bus_type)
  - Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able to detect the connected devices, and providing a communication mechanism 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, mainly struct usb_driver and struct usb_interface



Example: USB Bus

- Core infrastructure (bus driver), drivers/usb/core
  - struct 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 (Microchip, IXP, Xilinx, OMAP, Samsung, PXA, etc.)
- Device drivers
  - Everywhere in the kernel tree, classified by their type
  - Example: drivers/net/usb/

OHCI: USB 1.1
UHCI: USB 1.0 (intel)
EHCI: USB 2.0
XHCI: USB 3.0



Example of Device Driver

- 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)
- drivers/net/usb/rtl8150.c (device driver side, and not the adapter driver side)

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
  - 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.
  - /lib/modules/$(uname -r)/modules.{alias,usbmap}

```
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);
```

Instantiation of usb_driver

- struct 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 struct 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

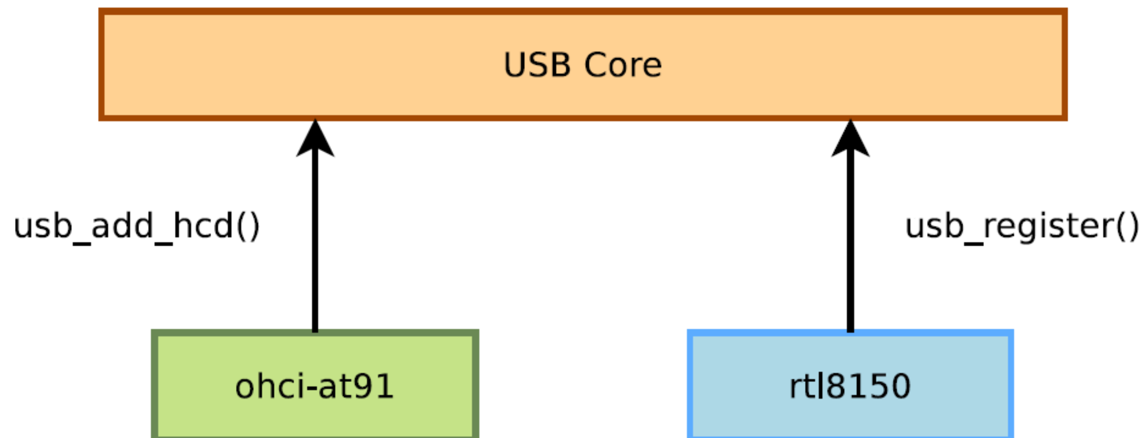
- When the driver is loaded or unloaded, it must register or unregister itself from the USB core
- Done using usb_register() and usb_deregister(), provided by the USB core.

```
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.

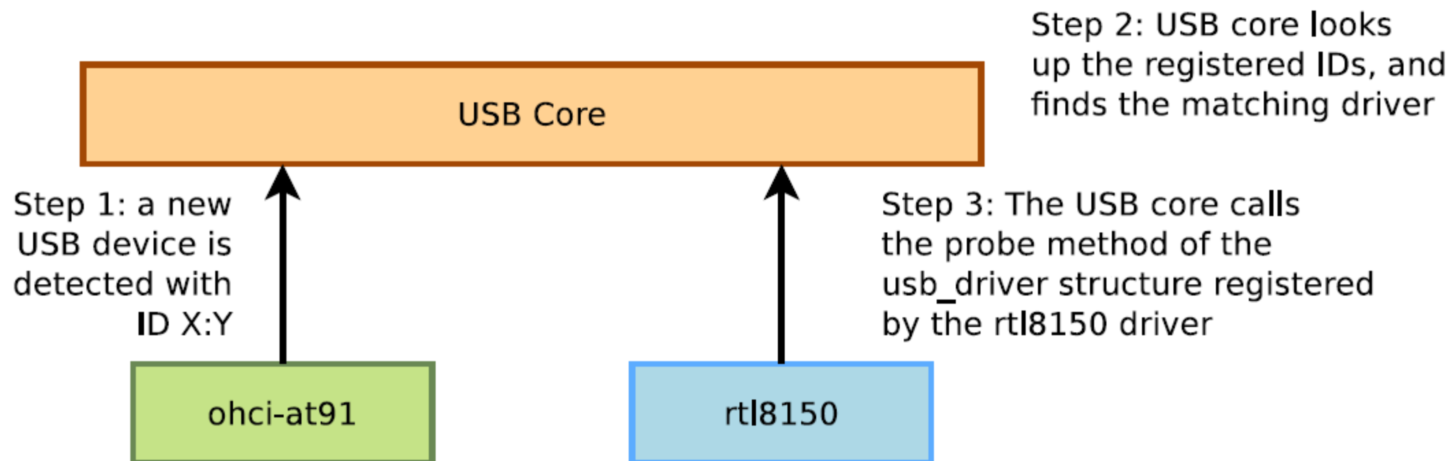
At Initialization

- 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 struct usb_driver structure of this driver



When a Device is Detected

- The probe() method receives as argument a structure describing the device, usually specialized by the bus infrastructure (struct pci_dev, struct usb_interface, etc.)
- probe() 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

```c
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;
}
```

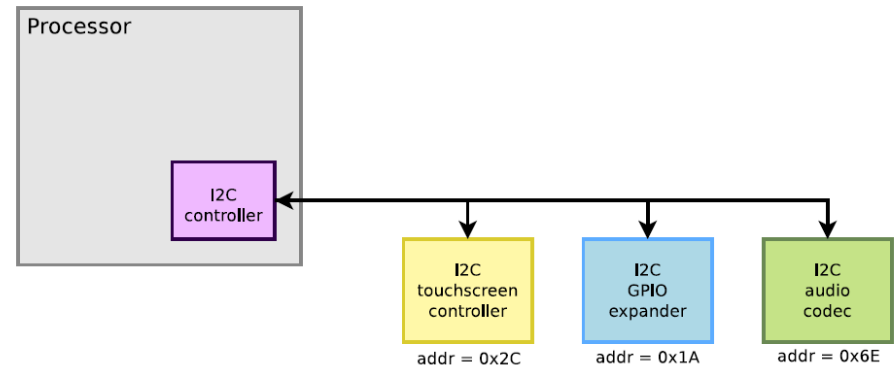
Non-discoverable busses

- 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.
- Instead of being dynamically detected, must be statically described in either:
  - kernel source code
  - 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.

```
struct platform_device {
        const char              *name;
        u32                     id;
        struct device           dev;
        u32                     num_resources;
        struct resource         *resource;
};
```



Implementation of a Platform Driver

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

```
static struct platform_driver serial_imx_driver = {
        .probe          = serial_imx_probe,
        .remove         = serial_imx_remove,
        .id_table       = imx_uart_devtype,
        .driver         = {
                .name   = "imx-uart",
                .of_match_table = imx_uart_dt_ids,
                .pm     = &imx_serial_port_pm_ops,
        },
};
```

```
struct platform_driver {
   int (*probe)(struct platform_device *);
   int (*remove)(struct platform_device *);
   void (*shutdown)(struct platform_device *);
   int (*suspend)(struct platform_device *, pm_message_t state);
   int (*resume)(struct platform_device *);
   struct device_driver driver;
   const struct platform_device_id *id_table;
   bool prevent_deferred_probe;
};
```

- 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: old style (1/2)

- As platform devices cannot be detected dynamically, they are defined statically
  - By direct instantiation of struct 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 struct platform_device structures are created
- Example on ARM, where the instantiation is done in

  arch/arm/mach-imx/mx1ads.c   Target machine-dependent initialization

```
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: old style (2/2)

- 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 struct resource, and an array of struct resource is associated to a struct platform_device
- Allows a driver to be instantiated for multiple devices functioning similarly, but with different addresses, IRQs, etc.

```
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,
    },
};
```

Declaring resources

Using Resources

- When a struct 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 (e.g., 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 struct platform_data field of struct device (from which struct 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 struct platform_data

platform_data example (1/2)

- The i.MX serial port driver defines the following structure to be passed through struct 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 instantiated such a structure

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

platform_data example (2/2)

- The uart1_pdata structure was associated to the struct platform_device structure in the MX1ADS board file (the real code was 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.
  - 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.
- On ARM, they are located in arch/arm/boot/dts.
- See https://elixir.bootlin.com/linux/v4.19.24/source/arch/arm/boot/dts/bcm2837-rpi-3-b-plus.dts

Device Tree example

```
auart0: serial@8006a000 {
        Defines the "programming model" for the device. Allows the
        operating system to identify the corresponding device driver.
        compatible = "fsl,imx28-auart", "fsl,imx23-auart";
        Address and length of the register area.
        reg = <0x8006a000 0x2000>;
        Interrupt number.
        interrupts = <112>;
        DMA engine and channels, with names.
        dmas = <&dma_apbx 8>, <&dma_apbx 9>;
        dma-names = "rx", "tx";
        Reference to the clock.
        clocks = <&clks 45>;
        The device is not enabled.
        status = "disabled";
};
```

Taken from arch/arm/boot/dts/imx28.dtsi

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.
  - Either by using the /include/ statement provided by the Device Tree language.
  - Either by using the #include statement, which requires calling the C preprocessor before parsing the Device Tree.
  - The trees described by the including file overlays the tree described by the included file.
- Linux currently uses either one technique or the other, (different from one ARM subarchitecture to another, for example).

Device Tree inheritance (2/2)

Definition of the AM33xx SoC

```
/ {
    compatible = "ti,am33xx";
    [...]
    ocp {
    [...]
        uart0: serial@44e09000 {
            compatible = "ti,am3352-uart",
                         "ti,omap3-uart";
            reg = <0x44e09000 0x2000>;
            interrupts = <72>;
            status = "disabled";
            [...]
        };
    };
};                                  am33xx.dtsi
```

+

Common definitions for BeagleBone boards

```
[...]
&uart0 {
        pinctrl-names = "default";
        pinctrl-0 = <&uart0_pins>;
        status = "okay";
};

                    am335x-bone-common.dtsi
```

+

Definition for BeagleBone Black

```
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
/ {
        model = "TI AM335x BeagleBone Black";
        compatible = "ti,am335x-bone-black",
                     "ti,am335x-bone",
                     "ti,am33xx";
};

[...]

                    am335x-boneblack.dts
```

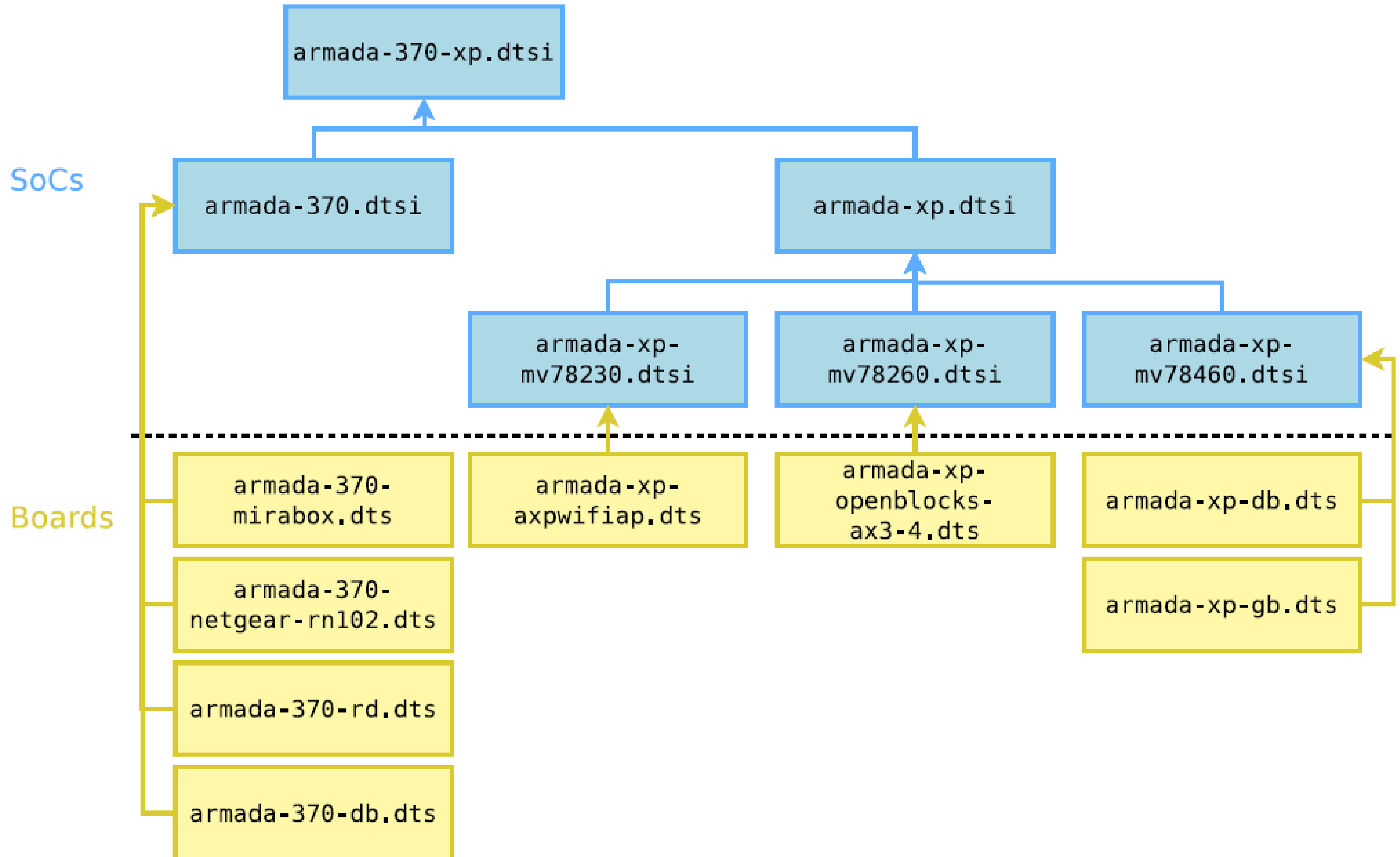
=

Compiled DTB

```
/ {
    compatible = "ti,am335x-bone-black", "ti,am335x-bone",
                 "ti,am33xx";
    model = "TI AMM335x BeagleBone Black";
    [...]
    ocp {
        uart0: serial@44e09000 {
            compatible = "ti,am3352-uart", "ti,omap3-uart";
            reg = <0x44e09000 0x2000>;
            interrupts = <72>;
            pinctrl-names = "default";
            pinctrl-0 = <&uart0_pins>;
            status = "okay";
        };
    };
};                                  am335x-boneblack.dtb
```

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

Device Tree inclusion example



Device Tree: compatible string

- With the *device tree*, a *device* is bound to the corresponding *driver* using the **compatible** string.
- The of_match_table field of struct 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*
  - When creating a new device tree representation for a device, a binding should be created that fully describes the required properties and value of the device.
  - https://elixir.bootlin.com/linux/v4.19.24/source/Documentation/devicetree/bindings

Device Tree binding documentation example

```
* Freescale MXS Application UART (AUART)

Required properties:
- compatible : Should be "fsl,<soc>-auart". The supported SoCs include
  imx23 and imx28.
- reg : Address and length of the register set for the device
- interrupts : Should contain the auart interrupt numbers
- dmas: DMA specifier, consisting of a phandle to DMA controller node
  and AUART DMA channel ID.
  Refer to dma.txt and fsl-mxs-dma.txt for details.
- dma-names: "rx" for RX channel, "tx" for TX channel.

Example:
auart0: serial@8006a000 {
        compatible = "fsl,imx28-auart", "fsl,imx23-auart";
        reg = <0x8006a000 0x2000>;
        interrupts = <112>;
        dmas = <&dma_apbx 8>, <&dma_apbx 9>;
        dma-names = "rx", "tx";
};

Note: Each auart port should have an alias correctly numbered in "aliases"
node.

Example:
[...]
```

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.