Introduction to Embedded Linux

Joel Fernandes
www.LinuxInternals.org
joel@linuxinternals.org

About me

- Working at Amazon's R&D group (Lab126)
- Embedded Linux for work and hobby
- OS Internals: Gives a complete picture
- Linux vs proprietary OS:
  - Picked-up Linux many years ago thanks to easy availability of source code and documentation

Quick stats about meetup

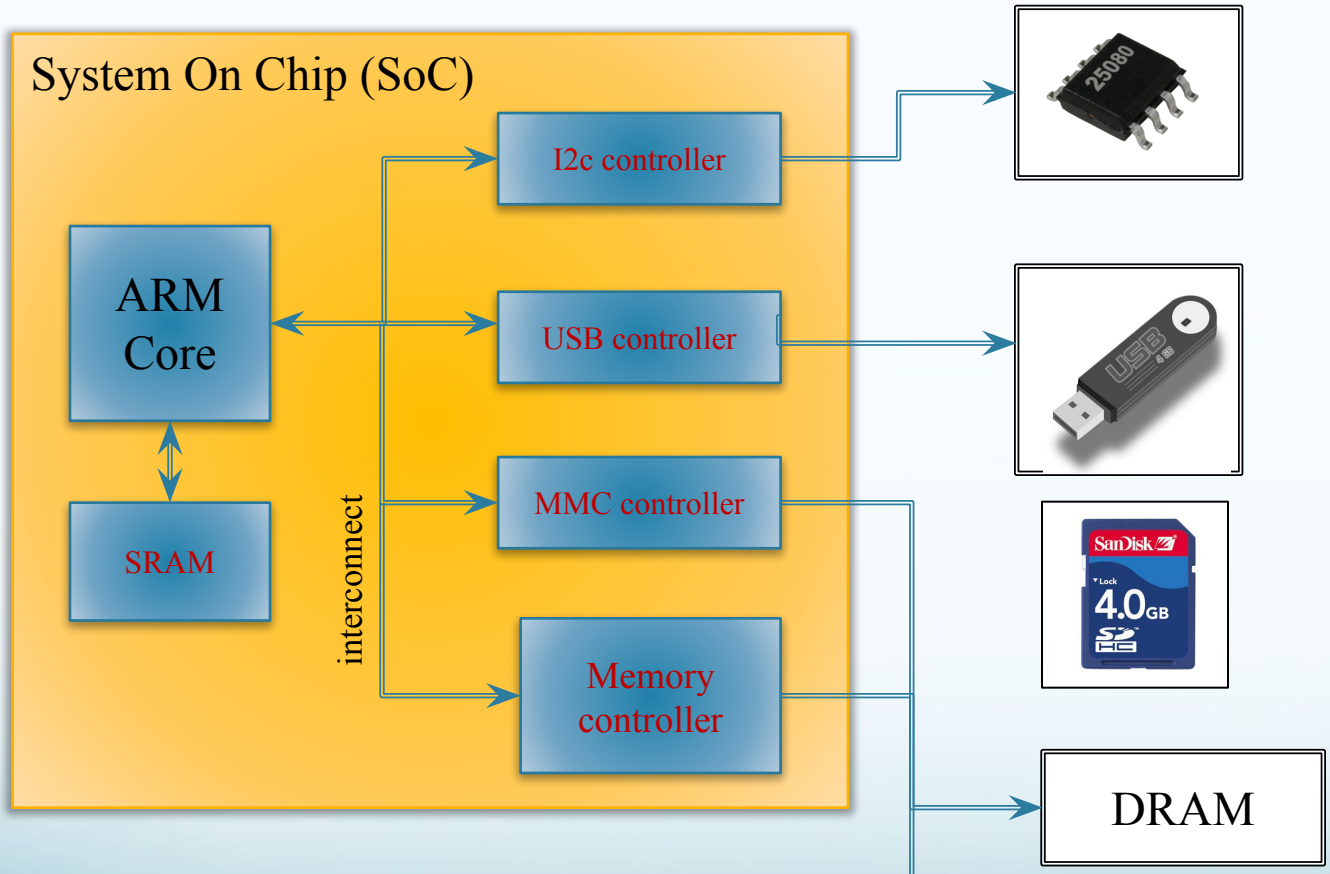
- First meetup early december, this ones a repeat
- 30% of RSVPs came in on the last 2 days
- 50+ new members in the last 2 weeks
- Still thinking about venue, but this one's better.

What's Embedded Linux? Some wikipedia definitions..

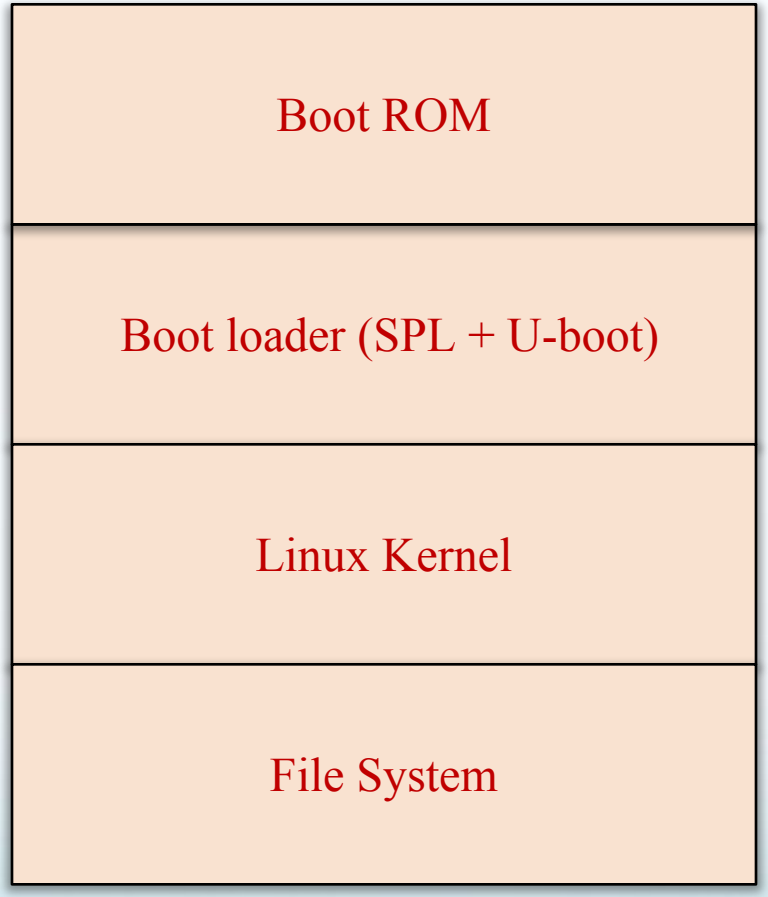
Embedded System:
An embedded system is a **computer system** with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts. Embedded systems **control** many devices in common use today.

Linux on Embedded:
Operating systems based on **the Linux kernel** are used in embedded systems such as consumer electronics (i.e. set-top boxes, smart TVs, in-vehicle infotainment (IVI), networking equipment (such as wireless routers), machine control, industrial automation, navigation equipment, spacecraft flight software, and medical instruments in general).

System On Chip (SoC) Architecture



Important Software Components



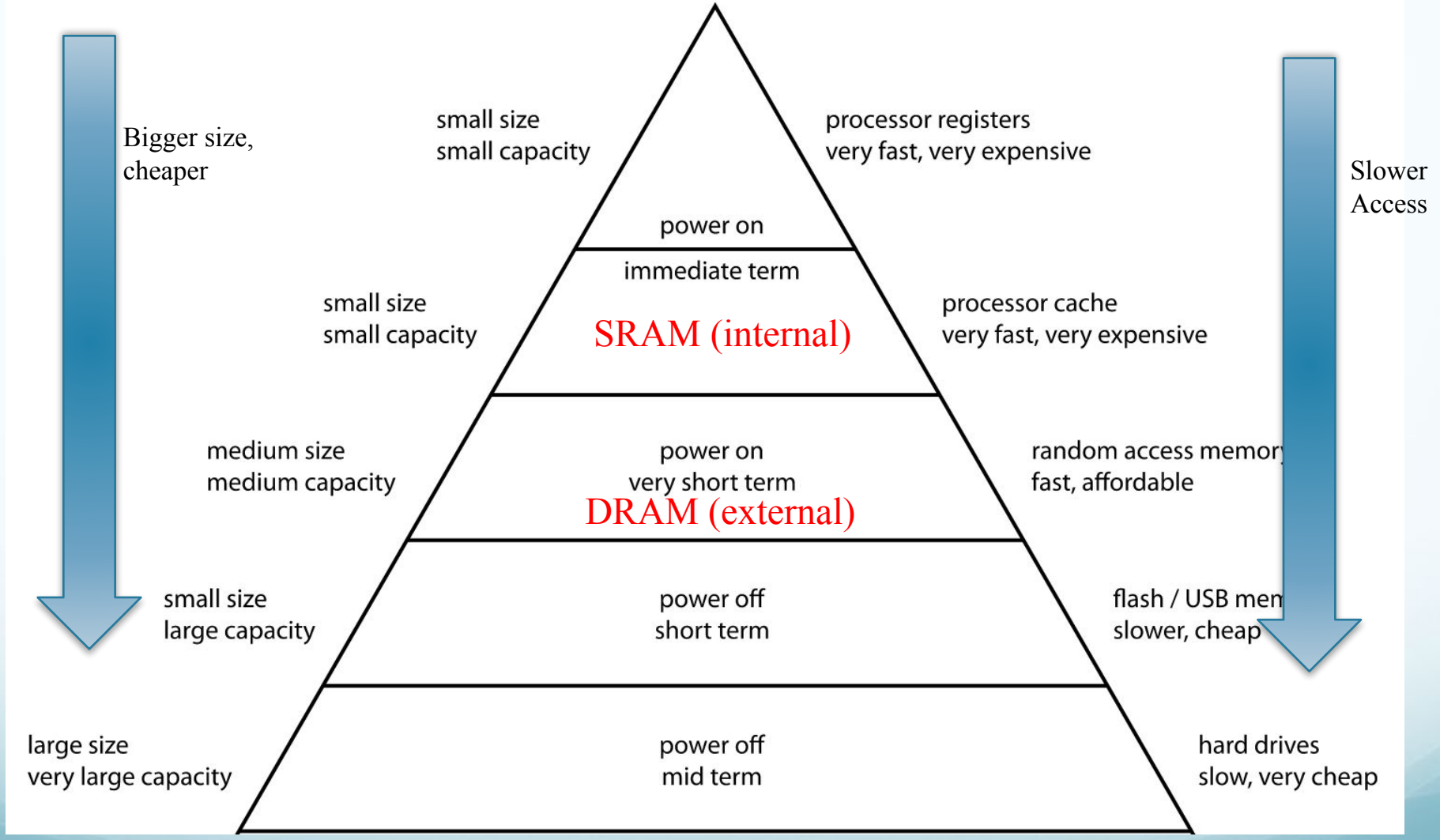
Bootloaders

- Role: Initializes the system enough to load the OS and load it.
- Main Responsibilities:
  - Where to load into? -> DRAM (can be DDR2/3/4 etc)
    - Initialize hardware: Memory controller, clocks for peripherals.
  - Where to load from?
    - Code to load from various media (Flash, USB, network, UART etc.)

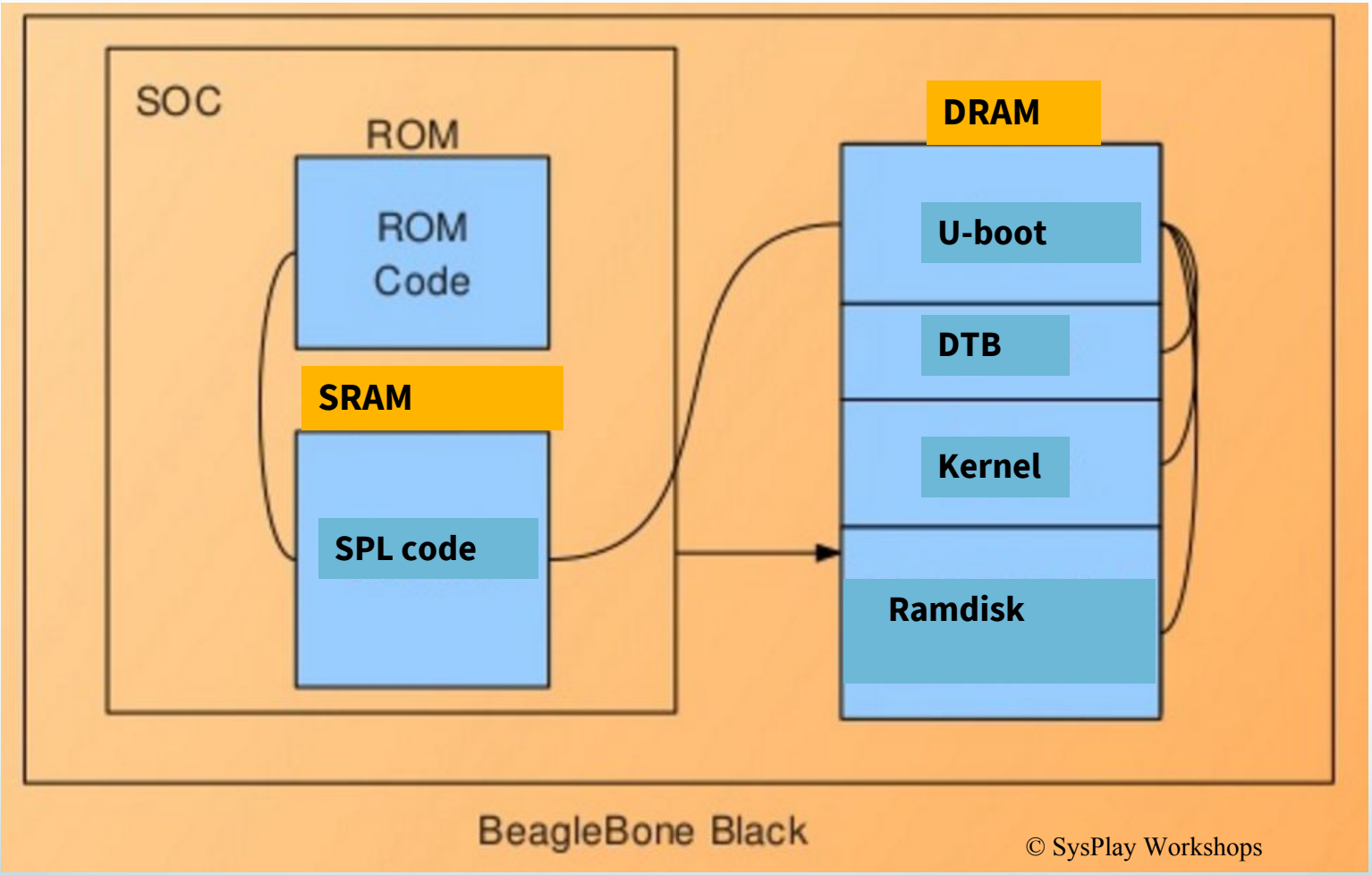
Bootloaders

- Other features:
  - Security: Loading trusted software
  - Flashing the OS onto flash memory
  - Diagnostics: Debugging/testing hardware without OS

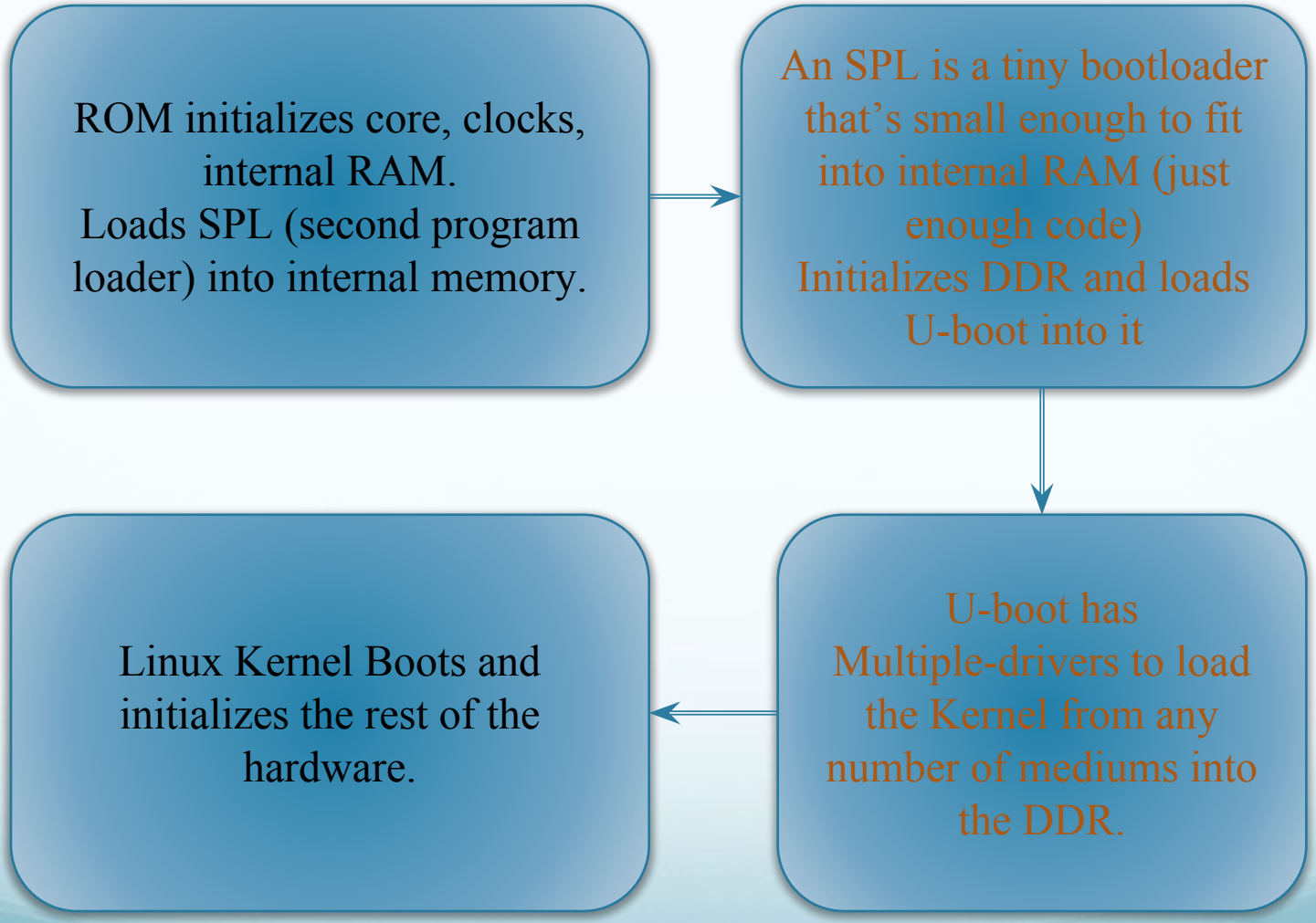
Computer Memory Hierarchy



Bootstrapping..



Example: U-boot



Example: U-boot

Boot ROM (or First-program loader):

- First program that executes from Read-only memory.
- ROM memory is typically inside an SoC and can be NOR flash.
- May read boot pins on an SoC to determine where to boot SPL from.

Example: U-boot

SPL (Second-program loader):

- Runs from internal RAM on the SoC
- Its kind of a mini U-boot with just enough code to load from 1-device (such as USB or network or Flash)
- Also called X-loader or pre-loader in some platforms
- Set up basic clocks and power
- Configures DRAM (DDR2, DDR3 etc..) memory
- Load the full-fledged U-Boot into DRAM
- Switch control to U-Boot code

Example: U-boot

U-boot (The second program which just got loaded)

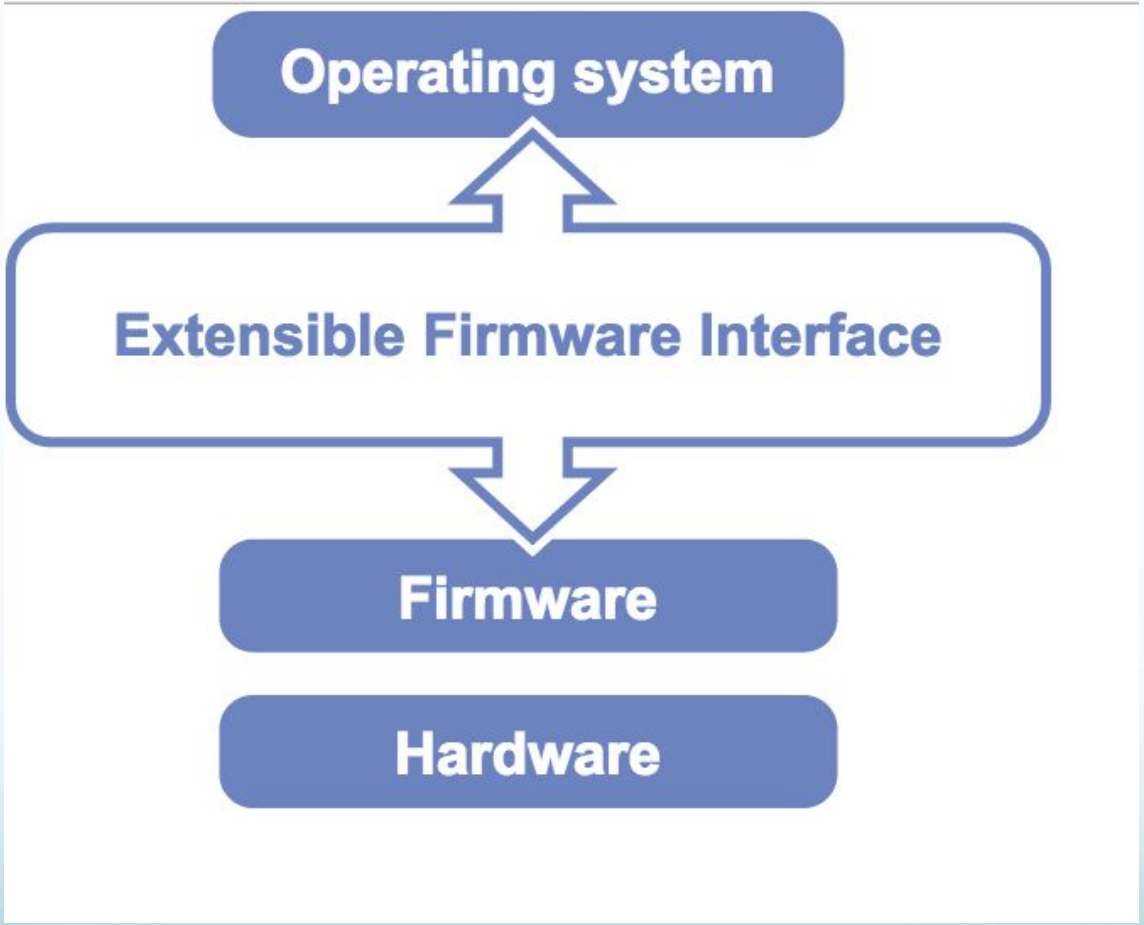
- Executes from DRAM (external memory)
- Contains lots of drivers in a single binary
- Has a scriptable environment
  - Can pass configuration scripts to it for boot flow
- Uses Virtual Memory: Initializes the TLB, MMU and Caches
  - Required for Caching to work. DCACHE speeds loading kernel
- Sets up an environment for and Loads Linux kernel

Quiz

- Can SPL load the Linux kernel directly?

- Ans: Yes! Why not. But not done very often because you'd lose U-boot's flexibility. SPL has a feature called "Falcon Mode"

Example: UEFI (BIOS)



Device-Tree

- The Device Tree is a data structure for describing hardware. Rather than hard coding every detail of a device into an operating system, many aspect of the hardware can be described in a data structure that is passed to the operating system at boot time.
- Single-kernel as much as possible philosophy..
- Data structure is stored in binary form as a blob (.dtb)
  - Can be packaged along with the kernel binary (Android does this)
  - Can be copied to memory at a certain location and the pointer of it passed to the kernel as a parameter. This is how U-boot typically does it.

Device-Tree basic format

```
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-binary-property = [0x01 0x23 0x34 0x56];
        a-cell-property = <0xbeef 123 0xabcd> // 32-bit uints
        child-node1 {
            first-child-property;
            second-child-property = <1>;
    };
};
```

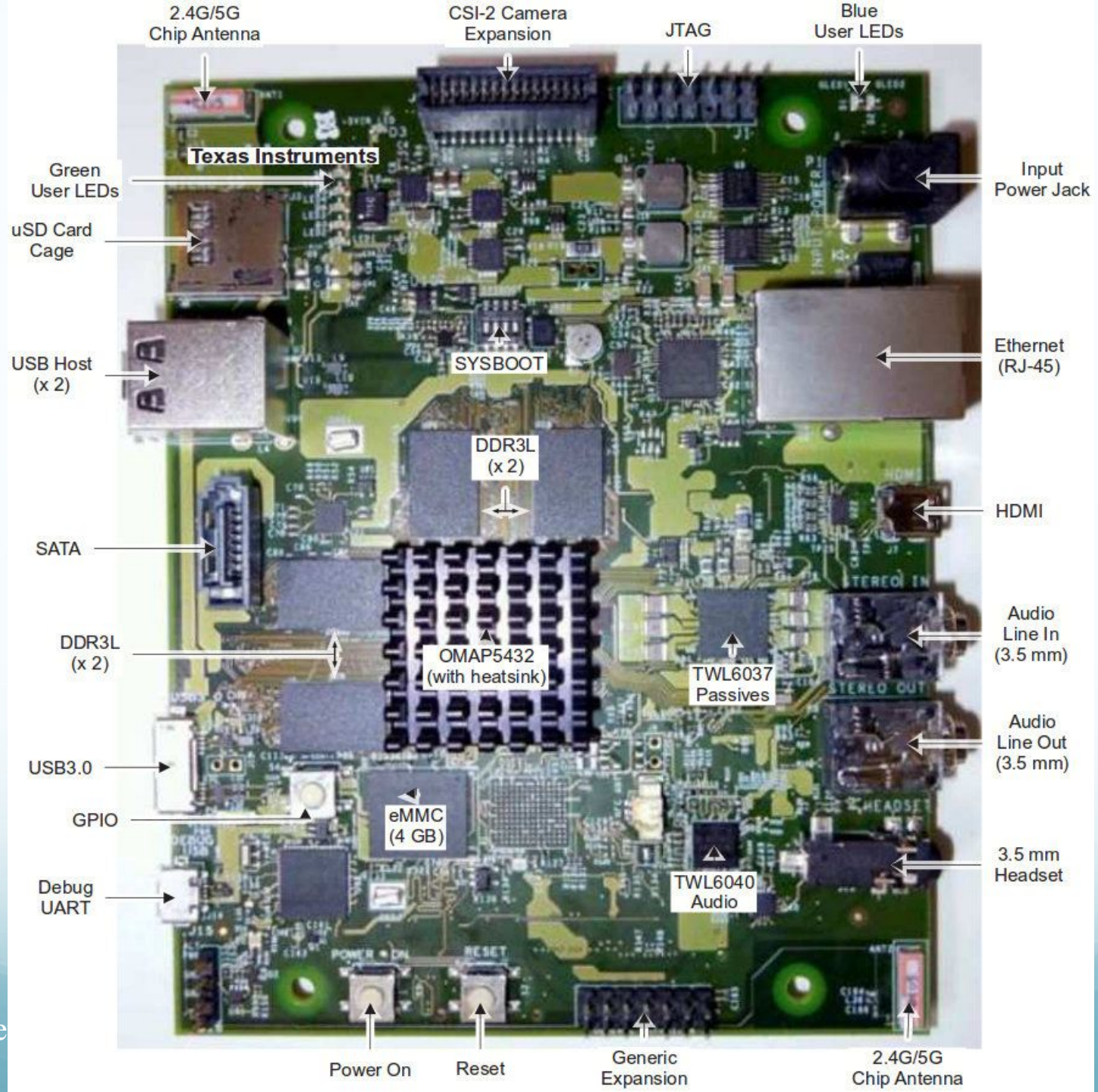
'Cells' are 32 bit unsigned integers delimited by angle brackets:
cell-property = <0xbeef 123 0xabcd1234>

binary data is delimited with square brackets:
binary-property = [0x01 0x23 0x45 0x67];

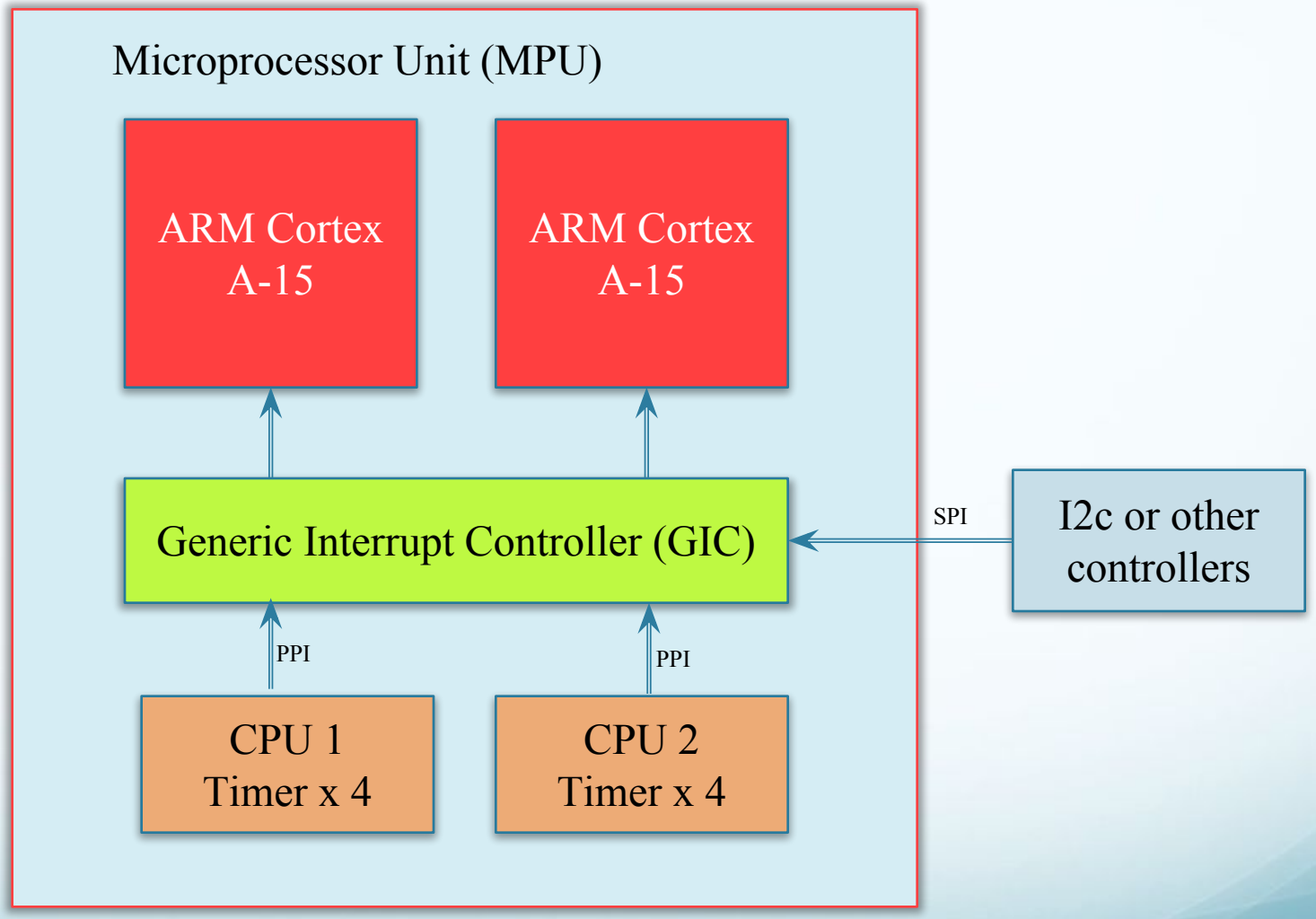
Data of differing representations can be concatenated together using a comma:
mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;

Device Tree example: OMAP5 uEVM

- Describe the OMAP5 uEVM board.
- Describe its peripherals, we will be going over:
  - CPU description
  - Interrupt Controller
  - Timers
  - I2C devices


2.4G/5G
Chip Antenna
CSI-2 Camera
Expansion
JTAG
Blue
User LEDs
Texas Instruments
Green
User LEDs
Input
Power Jack
uSD Card
Cage
USB Host
(x 2)
SYSBOOT
Ethernet
(RJ-45)
DDR3L
(x 2)
HDMI
SATA
Audio
Line In
(3.5 mm)
DDR3L
(x 2)
OMAP5432
(with heatsink)
TWL6037
Passives
Audio
Line Out
(3.5 mm)
USB3.0
GPIO
eMMC
(4 GB)
3.5 mm
Headset
Debug
UART
TWL6040
Audio
Power On
Reset
Generic
Expansion
2.4G/5G
Chip Antenna

Device Tree: OMAP5 uEVM



Device Tree: OMAP5 uEVM

A small primer on Timers on Cortex A-15

- In cortex-a15, each core has 4 timers:

- Secure Physical Timer
  - Used in Secure mode
- Non-secure Physical Timer
  - Most commonly used
- Virtual Timer
- Hypervisor Timer
  - Used in Hypervisor mode

- The CPUs have one generic timer block per CPU.
- The generic timer block contains 4 timers (there are 2 such blocks in our example, one for each A-15 core)

Device Tree: OMAP5 uEVM

A small primer on GIC (Generic Interrupt Controller)

- Interrupt Prioritization
- Interrupt Routing

3 types of interrupts

- SPI (Shared peripheral interrupt) – can be serviced by any core
  - Peripherals outside the core like i2c, usb, gpio etc.
  - We can set affinity of each interrupt source (force to a CPU).
- PPI (Private peripheral interrupt) – sent to only the core it belongs to
  - 7 PPIs per core
    - Each core has 4 timers so 4 timer PPIs per core)
- SGI (Software generated interrupts)
  - 16 per core (first 16 interrupt lines per core)
  - Used for scheduling, task-migrations, inter processor interrupts

Device Tree: OMAP5 uEVM

Describe the board first (OMAP5 uEVM):

```
/* File: arch/arm/boot/dts/omap5-uevm.dts */

/dts-v1/;

/ {
        model = "TI OMAP5 uEVM board";
        compatible = "ti,omap5-uevm", "ti,omap5"; /* identify the machine */
```

From Kernel Documentation:

The 'compatible' property contains a sorted list of strings starting with the exact name of the machine, followed by an optional list of boards it is compatible with sorted from most compatible to least. For example, the root compatible properties for the TI BeagleBoard and its successor, the BeagleBoard xM board might look like, respectively:

```
compatible = "ti,omap3-beagleboard", "ti,omap3450", "ti,omap3";
compatible = "ti,omap3-beagleboard-xm", "ti,omap3450", "ti,omap3";
```

Where "ti,omap3-beagleboard-xm" specifies the exact model, it also claims that it compatible with the OMAP 3450 SoC, and the omap3 family of SoCs in general. You'll notice that the list is sorted from most specific (exact board) to least specific (SoC family).

Compatible Property

From devicetree.org:

Every node in the tree that represents a device is required to have the compatible property. compatible is the key an operating system uses to decide which device driver to bind to a device.
compatible is a list of strings.

```
/ {
        compatible = "ti,omap5-uevm", "ti,omap5";
        timer {
    compatible = "arm,armv8-timer";
    interrupts = <GIC_PPI 13 IRQ_TYPE_LEVEL_LOW)>,
                ...
        };
    };
```

Device Tree: OMAP5 uEVM

Describe the CPUs:

```
cpus {
        #address-cells = <1>;
        #size-cells = <0>;

        cpu@0 {
                compatible = "arm,cortex-a15";
                device_type = "cpu";
                reg = <0x0>;                    /* CPU MPIDR match */

                operating-points = <
                        /* kHz    uV */
                        1000000 1060000
                        1500000 1250000
                >;

                clocks = <&dpll_mpu_ck>;
                clock-names = "cpu";
        };
        cpu@1 {
                compatible = "arm,cortex-a15";
                device_type = "cpu";
                reg = <0x1>;
        };
};
```

Device Tree: OMAP5 uEVM

Describe the Interrupt Controller (GIC):

```
gic: interrupt-controller@48211000 {
        compatible = "arm,cortex-a15-gic";
                        /* Identifies the node as an interrupt controller */
        interrupt-controller;

                    /* num of cells needed to encode an interrupt source */
        #interrupt-cells = <3>;

        /*  These are portions of memory in the physical address space,
                                which are used to program the GIC */
        reg = <0x48211000 0x1000>,
              <0x48212000 0x1000>,
              <0x48214000 0x2000>,
              <0x48216000 0x2000>;
};
```

Device Tree: OMAP5 uEVM

Finally, lets describe the Per-CPU timers.

Since its PPI, each of the 4 timer's interrupt number is the same across all CPUs. For ex, both secure timers, one for each a15 core has same IRQ number – 13 for both cores. (Kernel documentation on next slide)

```
timer {
        compatible = "arm,armv7-timer";


        /* Interrupt list for secure, non-secure, virtual and
           hypervisor timers */

        interrupts = <GIC_PPI 13 (IRQ_TYPE_LEVEL_LOW)>,
                     <GIC_PPI 14 (IRQ_TYPE_LEVEL_LOW)>,
                     <GIC_PPI 11 (IRQ_TYPE_LEVEL_LOW)>,
                     <GIC_PPI 10 (IRQ_TYPE_LEVEL_LOW)>;
        interrupt-parent = <&gic>;
};
```

Describing the Interrupt controller:

The 1st cell is the interrupt type; 0 for SPI interrupts, 1 for PPI interrupts.

The 2nd cell contains the interrupt number for the interrupt type. SPI interrupts are in the range [0-987]. PPI interrupts are in the range [0-15].

The 3rd cell is the flags, encoded as follows:

bits[3:0] trigger type and level flags.

1 = low-to-high edge triggered

2 = high-to-low edge triggered (invalid for SPIs)

4 = active high level-sensitive

8 = active low level-sensitive (invalid for SPIs).

bits[15:8] PPI interrupt cpu mask. Each bit corresponds to each of the 8 possible cpus attached to the GIC. A bit set to '1' indicated the interrupt is wired to that CPU. Only valid for PPI interrupts. Also note that the configurability of PPI interrupts is IMPLEMENTATION DEFINED and as such not guaranteed to be present (most SoC available in 2014 seem to ignore the setting of this flag and use the hardware default value).

Quiz 1

- Is device tree required for devices that can identify themselves at boot/run time?

- Ans: No! Some devices like USB and PCI identify themselves through unique identifiers. For such devices, device tree nodes are not required.

Quiz 2

- I have one kernel and 2 different DTBs available. Will they work on 2 different systems (such as an OMAP5 and a beagleboard)?

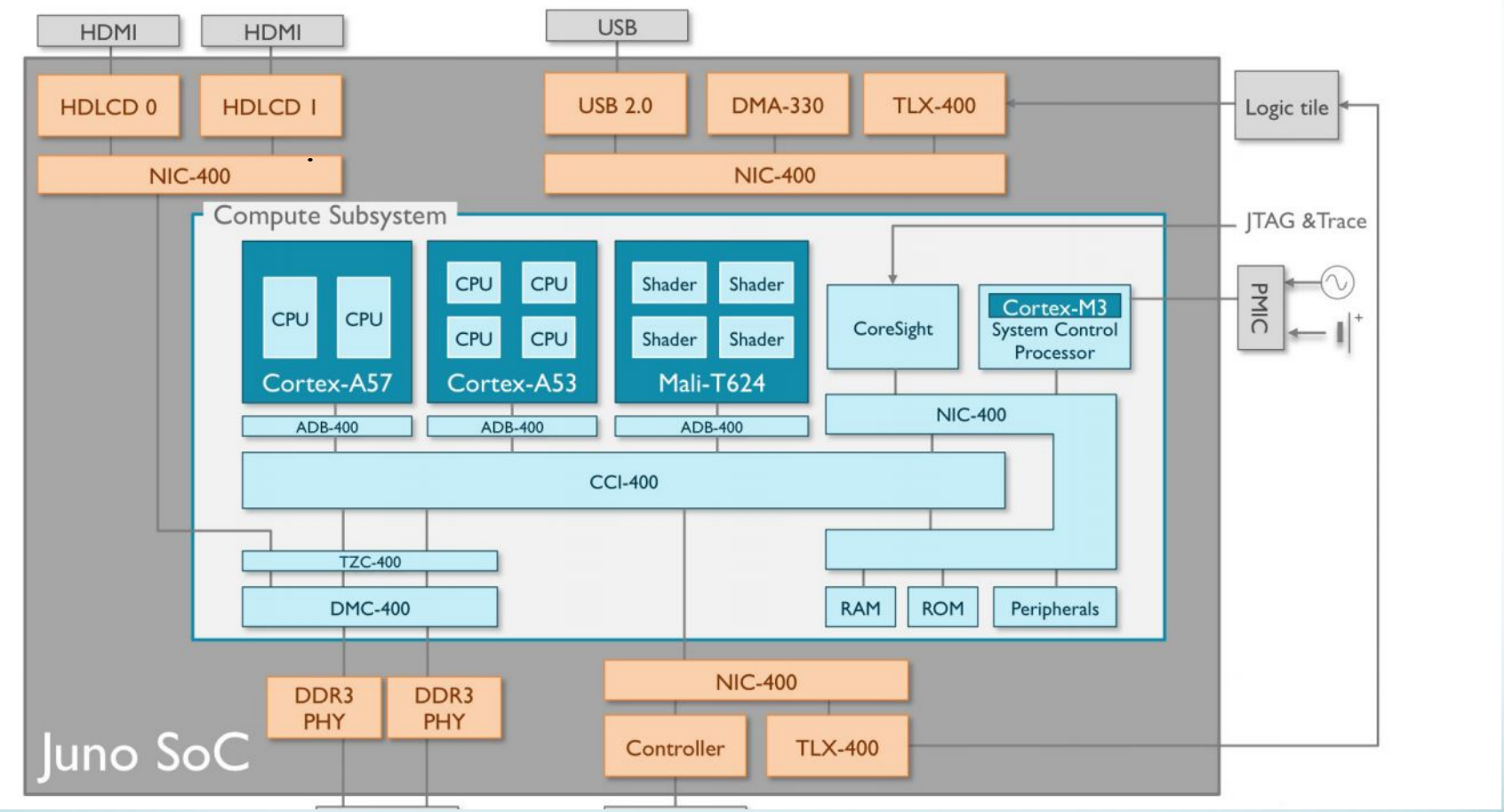
- Yes! The DTBs should account for the differences in hardware.

Device-Tree Example: ARM Juno

- Cortex A-57 cluster (x2 CPUs), 1.1GHz
  - L2: 2MB, L1: 48KB, I-cache 32KB
- Cortex A-53 cluster (x1 CPUs), 850MHz
  - L2: 2MB, L1: 32KB, No I-Cache ?
- Describe other hardware…

Device-Tree: ARM Juno

Device-Tree Example: ARM Juno



Device-Tree: ARM Juno

- First describe the machine:

  - Start with the root node “/”
  - model: name of the board
  - compatible: the board is compatible with kernel that supports “arm,juno” and “arm,vexpress” machines.

(Confirm the below from dts and DT documentation)

```
/* File: arch/arm64/boot/dts/arm-juno.dts */

/ {
        model = "ARM Juno development board (r0)";
        compatible = "arm,juno", "arm,vexpress”;
```

Describing the CPUS:

```
cpus {
        #address-cells = <2>; // length of addr field
        #size-cells = <0>;       // length of size field

        A57_0: cpu@0 {
                compatible = "arm,cortex-a57","arm,armv8";
                reg = <0x0 0x0>;   // 64-bit ID (compared to MPIDR)
                device_type = "cpu";
                enable-method = "psci";
                next-level-cache = <&A57_L2>;
        };

        A57_1: cpu@1 {
                compatible = "arm,cortex-a57","arm,armv8";
                reg = <0x0 0x1>;
                device_type = "cpu";
                enable-method = "psci";
                next-level-cache = <&A57_L2>;
        };

        A53_0: cpu@100 {
                compatible = "arm,cortex-a53","arm,armv8";
                reg = <0x0 0x100>;
                device_type = "cpu";
                enable-method = "psci";
                next-level-cache = <&A53_L2>;
        };

        A53_1: cpu@101 {
                compatible = "arm,cortex-a53","arm,armv8";
                reg = <0x0 0x101>;
                device_type = "cpu";
                enable-method = "psci";
                next-level-cache = <&A53_L2>;

       A53_2: cpu@102 {
                compatible = "arm,cortex-a53","arm,armv8";
                reg = <0x0 0x102>;
                device_type = "cpu";
                enable-method = "psci";
                next-level-cache = <&A53_L2>;
        };

        A53_3: cpu@103 {
                compatible = "arm,cortex-a53","arm,armv8";
                reg = <0x0 0x103>;
                device_type = "cpu";
                enable-method = "psci";
                next-level-cache = <&A53_L2>;
        };

        A57_L2: l2-cache0 {
                compatible = "cache";
        };

        A53_L2: l2-cache1 {
                compatible = "cache";
        };
};
```

Describing the Interrupt controller:

```
        gic: interrupt-controller@2c010000 {
                compatible = "arm,gic-400", "arm,cortex-a15-gic";
                reg = <0x0 0x2c010000 0 0x1000>,
                      <0x0 0x2c02f000 0 0x2000>,
                      <0x0 0x2c04f000 0 0x2000>,
                      <0x0 0x2c06f000 0 0x2000>;
                 ……
};
```

Describing the timers:

```
  timer {
        compatible = "arm,armv8-timer";
        interrupts = <GIC_PPI 13 IRQ_TYPE_LEVEL_LOW)>,
                     <GIC_PPI 14 IRQ_TYPE_LEVEL_LOW>,
                     <GIC_PPI 11 IRQ_TYPE_LEVEL_LOW>,
                     <GIC_PPI 10 IRQ_TYPE_LEVEL_LOW>;
       /* See next slide for interrupt documentation */
  };
```

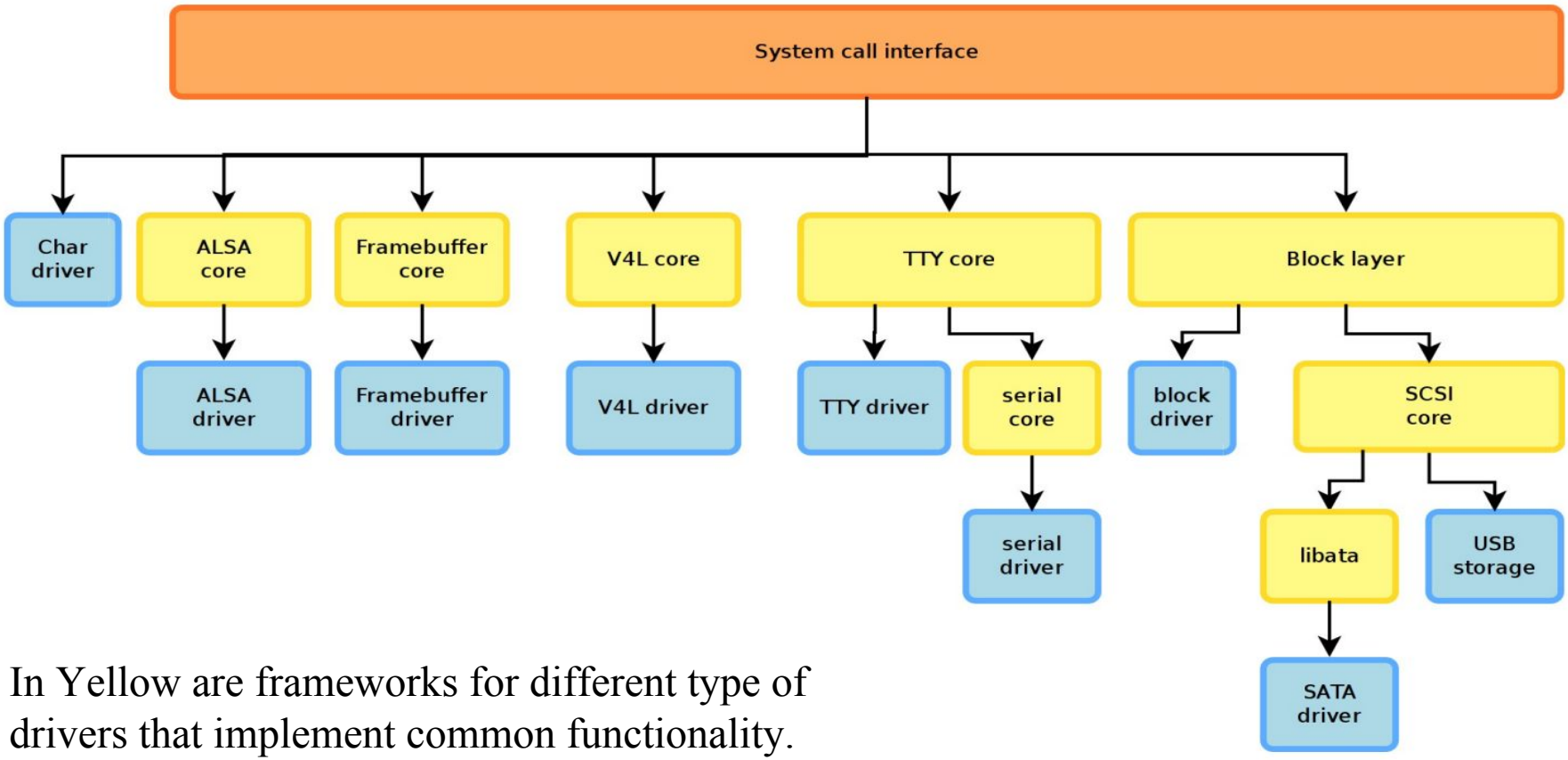
Describing other hardware: I2C

```
i2c@7ffa0000 {
        compatible = "snps,designware-i2c";
        reg = <0x0 0x7ffa0000 0x0 0x1000>;

                        /* Describe how to identify child nodes */
        #address-cells = <1>;    // all i2c address are 32-bit
        #size-cells = <0>;       // no length field required

        interrupts = <GIC_SPI 104 IRQ_TYPE_LEVEL_HIGH>;
        clock-frequency = <400000>;
        clocks = <&soc_smc50mhz>;

        dvi0: dvi-transmitter@70 {      // i2c slave 1
                compatible = "nxp,tda998x";
                reg = <0x70>;    // EDID address
        };

        dvi1: dvi-transmitter@71 {      // i2c slave 2
                compatible = "nxp,tda998x";
                reg = <0x71>;
        };
};
```

Basics of Device Drivers

- Kernel frameworks (common-code)
- Device/Driver model
- Platform bus

Kernel Frameworks



In Yellow are frameworks for different type of drivers that implement common functionality.

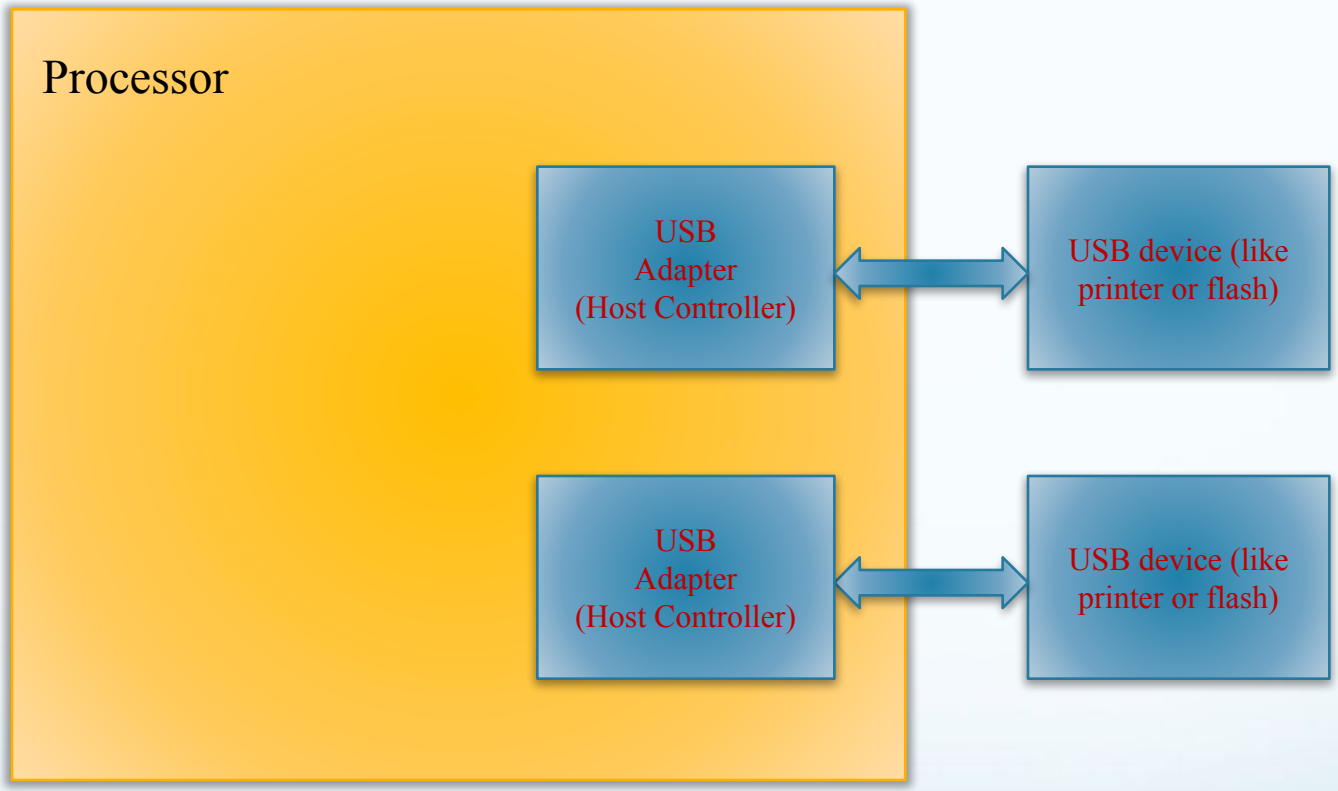
Device/Driver model

- Since 2.6 kernel, Linux has a unified device and driver model.
  - Instead of different ad-hoc mechanisms in each subsystem, the device model **unifies the vision** of the devices, drivers, their organization and relationships.
  - **Lesser code duplication**, provides common facilities and more coherency in the code organization.

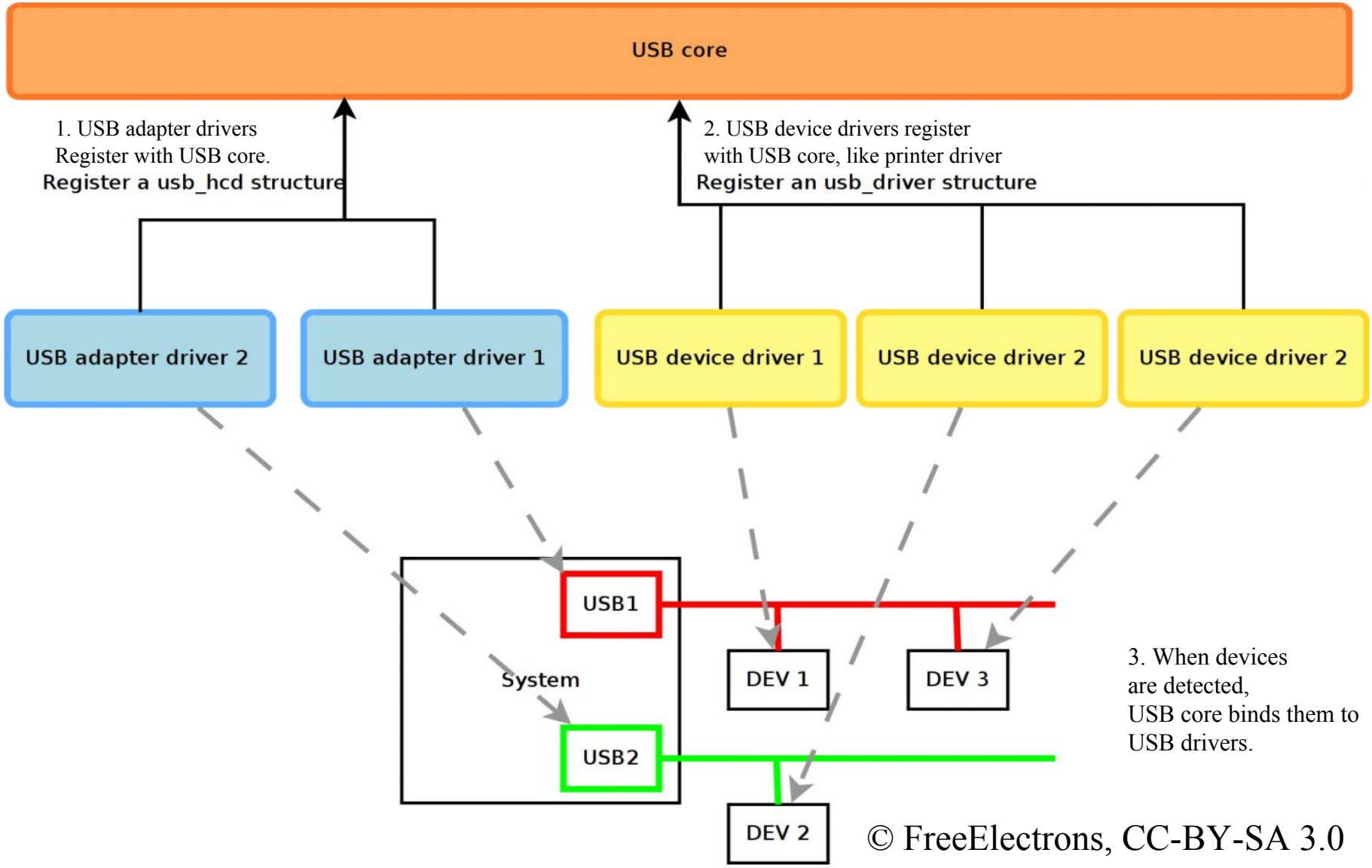
- Defines base structure types:
  - struct device
  - struct driver
  - struct bus type.

- sysfs filesystem shows all these, mounted under /sys

Example: USB device driver



Framework, drivers and devices



Concept of a Bus

- Most important element of the device/driver model
- Different drivers register a bus_type struct with driver core
  - USB
  - PCI
  - I2C
  - SPI
  - MMC

  etc..
- For each bus, the driver core will match devices and drivers

USB core registers with Linux as a “bus”

```
// File: drivers/usb/core/usb.c

struct bus_type usb_bus_type = {
        .name =         "usb",
        .match =        usb_device_match,
        .uevent =       usb_uevent,
};

static int __init usb_init() {
        ...
        retval = bus_register(&usb_bus_type);
        ...
}
```

USB host controllers (aka adapters) register with core

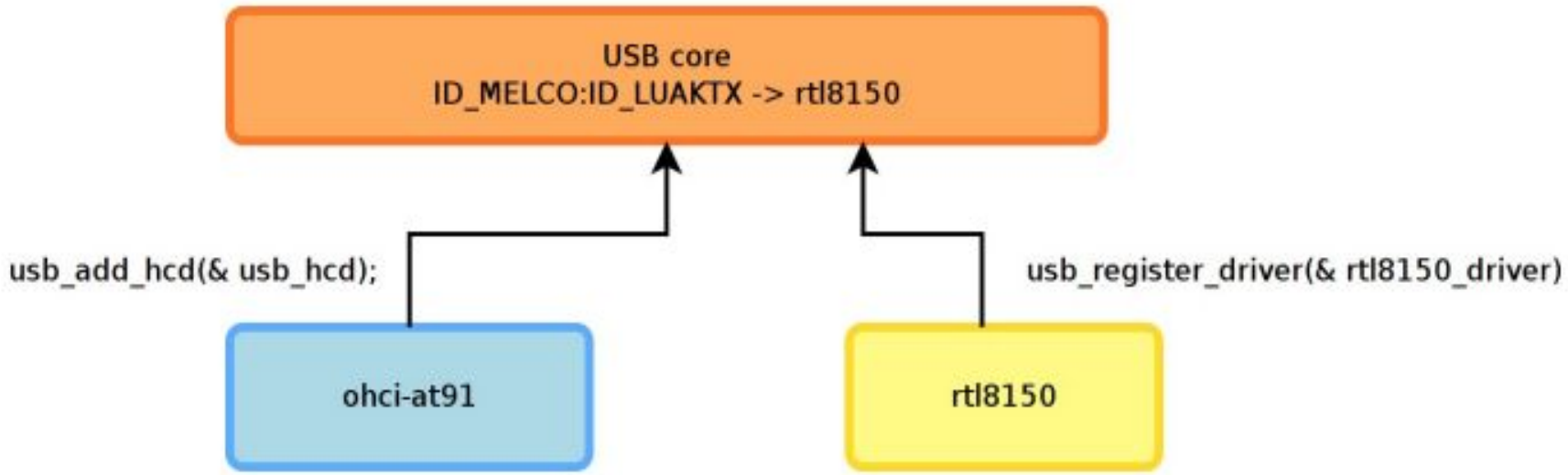
```
/*
 *      File: drivers/usb/chipidea/host.c
 */

static int host_start(struct ci_hdrc *ci)
{
        struct usb_hcd *hcd;
        ...
    hcd->rsrc_start = ci->hw_bank.phys;
    hcd->rsrc_len = ci->hw_bank.size;
    hcd->regs = ci->hw_bank.abs;
    hcd->has_tt = 1;
        ...
        ret = usb_add_hcd(hcd, 0, 0);
        ...
}
```

A USB driver registering with USB core

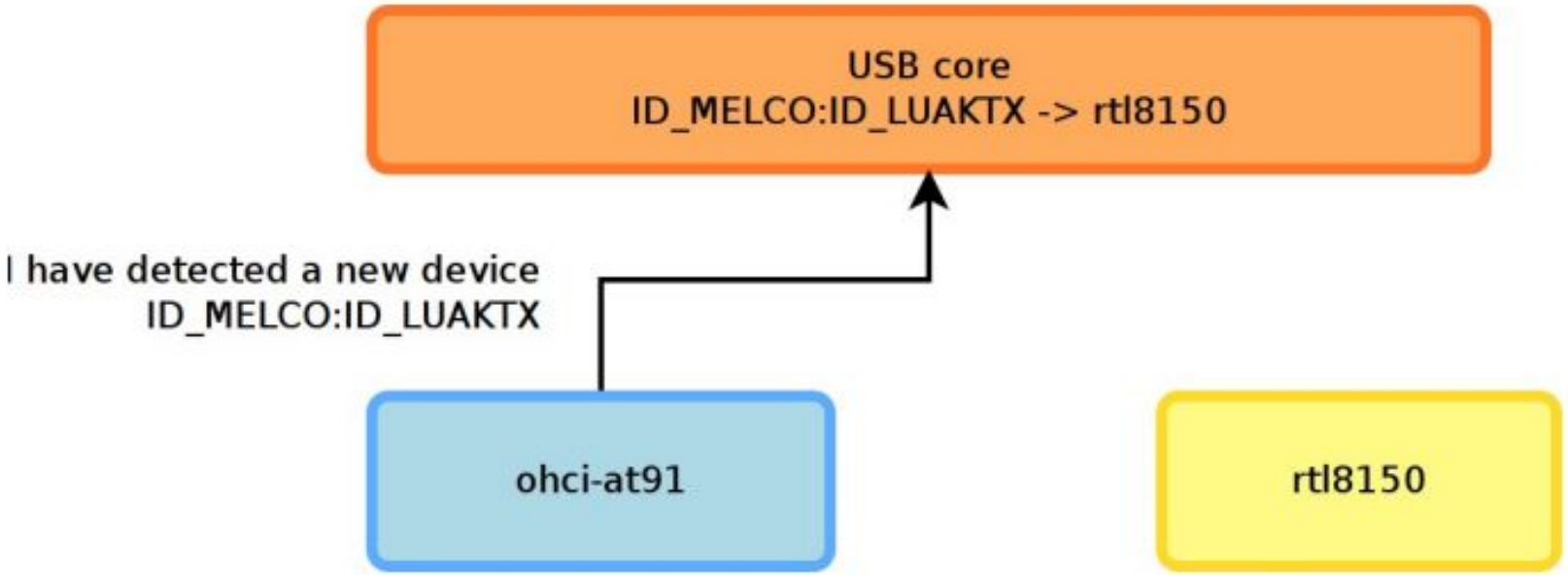
```
/* table of devices that work with this driver */
static const struct usb_device_id id_table[] = {
        { USB_DEVICE(0x0fc5, 0x1223) },
        { USB_DEVICE(0x1d34, 0x0004) },
        { USB_DEVICE(0x1d34, 0x000a) },
        { USB_DEVICE(0x1294, 0x1320) },
        { },
};

static struct usb_driver led_driver = {
        .name =         "usbled",
        .probe =        led_probe,
        .disconnect =   led_disconnect,
        .id_table =     id_table,
};

usb_register(&led_driver)
```

When a USB adapter driver notifies the USB core of a new device,
The USB core binds the new device with the driver based on the id_table

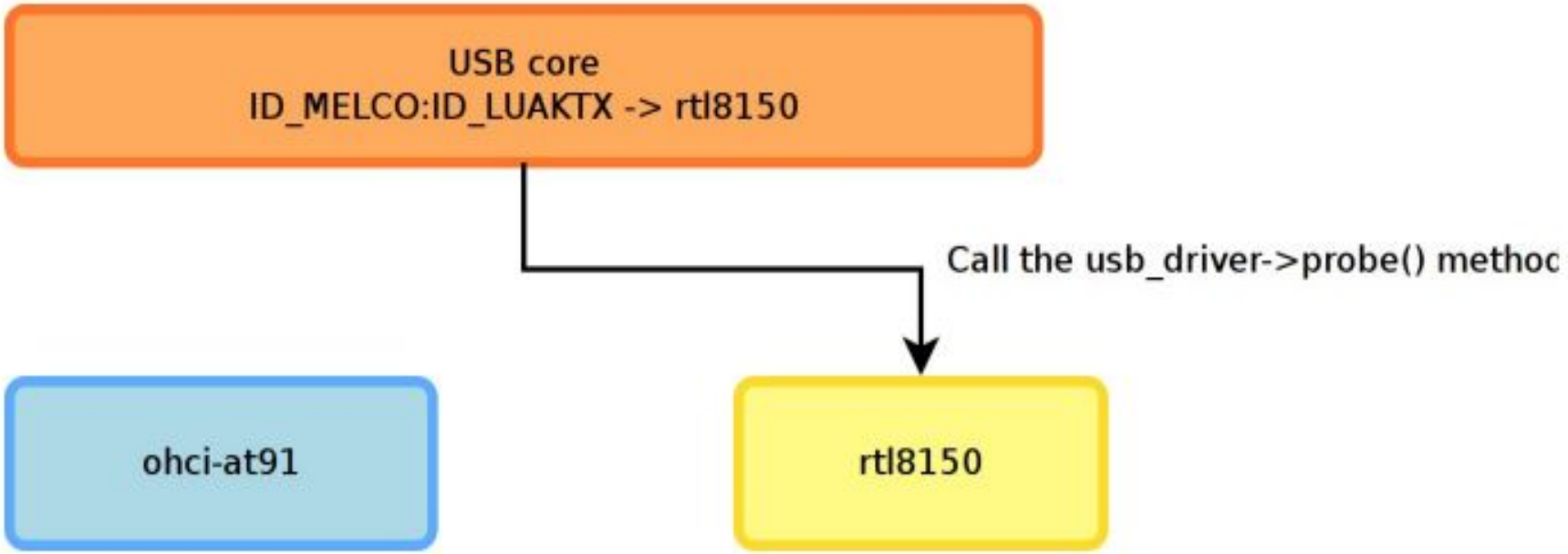
Example of USB driver probing



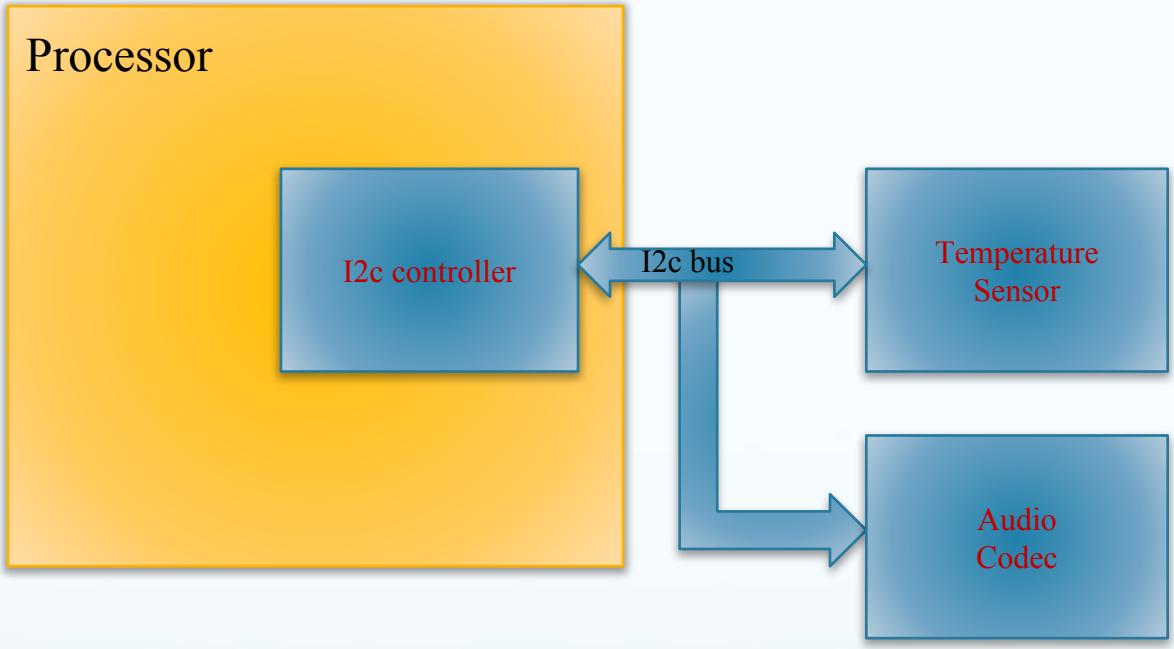
Example of USB driver probing



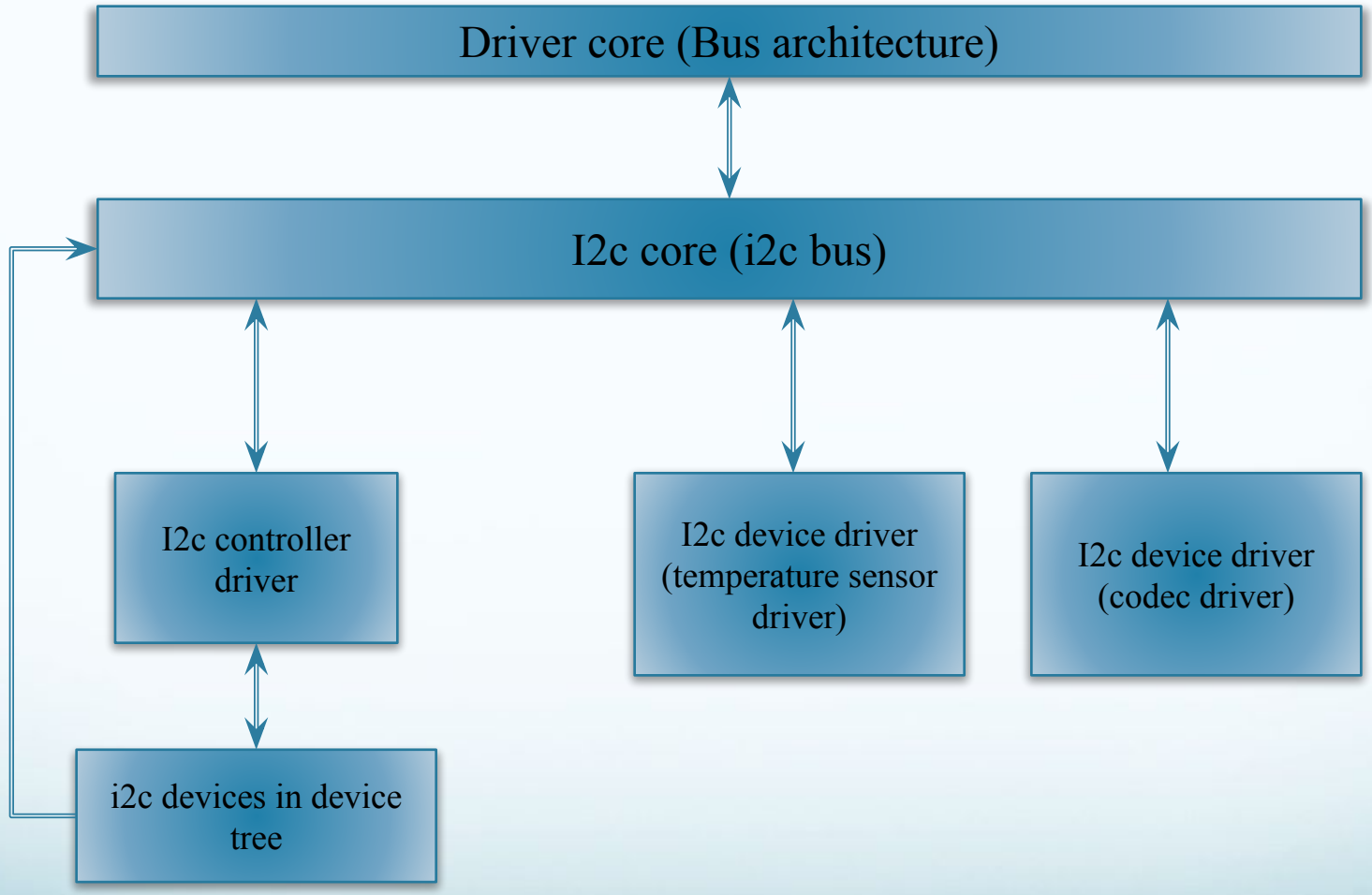
Example of USB driver probing



Another example: i2c device drivers



Another example: i2c device drivers



I2c devices are registered onto the bus after parsing Device Tree.

Another example: i2c core registers as a "BUS"

```
/* File: drivers/i2c/i2c-core.c */


struct bus_type i2c_bus_type = {
        .name           = "i2c",
        .match                  = i2c_device_match,
        .probe          = i2c_device_probe,
        .remove                 = i2c_device_remove,
        .shutdown       = i2c_device_shutdown,
        .pm             = &i2c_device_pm_ops,
};
EXPORT_SYMBOL_GPL(i2c_bus_type);


static int __init i2c_init(void)
{
        int retval;
        retval = bus_register(&i2c_bus_type);
        ...
}
```

i2c device drivers register with i2c core

```
/* File: drives/hwmon/tmp102.c */

static const struct i2c_device_id tmp102_id[] = {
        { "tmp102", 0 },
        { }
};

static struct i2c_driver tmp102_driver = {
        .driver.name= DRIVER_NAME,
        .driver.pm    = TMP102_DEV_PM_OPS,
        .probe        = tmp102_probe,
        .remove               = tmp102_remove,
        .id_table     = tmp102_id,
};


module_i2c_driver(tmp102_driver); /* Register with i2c core */
```

I2c adapter driver registers with the i2c core

```
/* File: drivers/busses/i2c/i2c-omap.c */

static const struct i2c_algorithm omap_i2c_algo = {
        .master_xfer= omap_i2c_xfer,            // send/receive data
        .functionality        = omap_i2c_func,  // what the adapter supports
};


int omap_i2c_probe(struct platform_device *pdev) {
        ...
        adap->owner = THIS_MODULE;
        strlcpy(adap->name, "OMAP I2C adapter", sizeof(adap->name));
        adap->algo = &omap_i2c_algo;

        /* i2c device drivers may be active on return from add_adapter() */
        adap->nr = pdev->id;
        r = i2c_add_numbered_adapter(adap);
        ...
}
```

i2c devices are detected through Device Tree

i2c core goes through all devices in DT and looks for matching drivers.

```
/* File: arch/arm/boot/dts/am57xx-beagle-x15.dts */

i2c2 {
        tmp102: tmp102@48 {
                compatible = "ti,tmp102";
                reg = <0x48>;
                ...
        };

        tlv320aic3104: tlv320aic3104@18 {                    compatible = "ti,tlv320aic3104";

                reg = <0x18>;

                AVDD-supply = <&vdd_3v3>;
                IOVDD-supply = <&vdd_3v3>;
                DRVDD-supply = <&vdd_3v3>;
                DVDD-supply = <&aic_dvdd>;
        };
}
```

i2c adapter detects i2c devices through DT

i2c devices are not “enumerated” like USB. So the i2c adapter driver has to manually parse the DT and let the “bus” code know about these devices

```
/* We just saw this code */

int omap_i2c_probe(struct platform_device *pdev) {
        ...
        adap->owner = THIS_MODULE;
        strlcpy(adap->name, "OMAP I2C adapter", sizeof(adap->name));
        adap->algo = &omap_i2c_algo;

        adap->nr = pdev->id;
        r = i2c_add_numbered_adapter(adap);
        ...
        of_i2c_register_devices(adap);    /* Let the Bus core code know of devices */
        ...
}
```