How does the linux kernel get to know about hardware present in a particular board?

Discoverable devices, **platform devices, device trees**, registering and probing

Platform devices and drivers

- Some buses (USB, PCI) can auto-detect devices
- Not very frequent in embedded systems
- UARTs, flash memory, LEDs, GPIOs, MMC/SD, Ethernet cannot

Solution:

1) Provide a description of devices
2) Manage them through a fake 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.

Device Tree

```
/dts-v1/;

/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        // hex is implied in byte arrays. no '0x' prefix is required
        a-byte-data-property = [01 23 34 56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

Properties

Simple key-value pairs where the value can either be empty or contain an arbitrary byte stream.

While data types are not encoded into the data structure, there are a few fundamental data representations that can be expressed in a device tree source file.

- Text strings (null terminated) are represented with double quotes:
  - string-property = "a string";
- '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>;
- Commas are also used to create lists of strings:
  - string-list = "red fish", "blue fish";

Sample Machine

(loosely based on ARM Versatile), manufactured by "Acme" and named "Coyote's Revenge"

- One 32bit ARM CPU
- processor local bus attached to **memory mapped** serial port, gpio controller, spi bus controller, i2c controller, interrupt controller, and external bus bridge
- 256MB of SDRAM based at 0
- 2 Serial ports based at 0x101F1000 and 0x101F2000
- GPIO controller based at 0x101F3000
- SPI controller based at 0x10170000 with following devices
  - MMC slot with SS pin attached to GPIO #1
- External bus bridge with following devices
  - SMC SMC91111 Ethernet device attached to external bus based at 0x10100000
  - i2c controller based at 0x10160000 with following devices
    - Maxim DS1338 real time clock. Responds to slave address 1101000 (0x58)
  - 64MB of NOR flash based at 0x30000000

Initial Structure

```
/dts-v1/;


/ {
    compatible = "acme,coyotes-revenge";
};
```

Compatible specifies the name of the system. It contains a string in the form "<manufacturer>,<model>.

It is important to specify the exact device, and to include the manufacturer name to avoid namespace collisions. Since the operating system will use the compatible value to make decisions about how to run on the machine, it is very important to put correct data into this property.

CPUs

```
/dts-v1/;

/ {
    compatible = "acme,coyotes-revenge";

    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
};
```

The compatible property in each cpu node is a string that specifies the exact cpu model in the form <manufacturer>,<model>, just like the compatible property at the top level.

Node Names

- Every node must have a name in the form <name>[@<unit-address>].
- <name> is a simple ascii string and can be up to 31 characters in length. In general, nodes are named according to what kind of device it represents. ie. A node for a 3com Ethernet adapter would be use the name ethernet, not 3com509.
- The unit-address is included if the node describes a device with an address. In general, the unit address is the primary address used to access the device, and is listed in the node's reg property.
- Sibling nodes must be uniquely named, but it is normal for more than one node to use the same generic name so long as the address is different (ie, serial@101f1000 & serial@101f2000).

A node for each device

```
/dts-v1/;

/ {
    compatible = "acme,coyotes-revenge";

    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
```

A node for each device (contd.)

```
serial@101F0000 {
    compatible = "arm,pl011";
};

serial@101F2000 {
    compatible = "arm,pl011";
};

gpio@101F3000 {
    compatible = "arm,pl061";
};

interrupt-controller@10140000 {
    compatible = "arm,pl190";
};

spi@10115000 {
    compatible = "arm,pl022";
};

external-bus {
    ethernet@0,0 {
        compatible = "smc,smc91c111";
    };

    i2c@1,0 {
        compatible = "acme,a1234-i2c-bus";
        rtc@58 {
            compatible = "maxim,ds1338";
        };
    };

    flash@2,0 {
        compatible = "samsung,k8f1315ebm", "cfi-flash";
    };
};
};
```

Compatible Property

- 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. The first string in the list specifies the exact device that the node represents in the form "<manufacturer>,<model>". The following strings represent other devices that the device is compatible with.

- For example, the Freescale MPC8349 System on Chip (SoC) has a serial device which implements the National Semiconductor ns16550 register interface. The compatible property for the MPC8349 serial device should therefore be: compatible = "fsl,mpc8349-uart", "ns16550". In this case, fsl,mpc8349-uart specifies the exact device, and ns16550 states that it is **register-level compatible** with a National Semiconductor 16550 UART.
  - Note: ns16550 doesn't have a manufacturer prefix purely for historical reasons. All new compatible values should use the manufacturer prefix.

- This practice allows **existing device drivers to be bound to a newer device**, while still uniquely identifying the exact hardware.

- Warning: Don't use wildcard compatible values, like "fsl,mpc83xx-uart" or similar. Silicon vendors will invariably make a change that breaks your wildcard assumptions the moment it is too late to change it. Instead, choose a specific silicon implementations and make all subsequent silicon compatible with it.

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),
 },
};
```

Matching and probing happens on platform driver registration

driver_register -> bus_add_driver -> driver_attach -> __driver_attach

->driver_probe_device -> really_probe -> drv->probe

- When platform driver is registered, the platform bus(platform_bus_type) will scan for the matching device.
- platform_bus_type structure has match function which does the matching of driver and device.
- Once it's found the right device, it will attach the driver to the device through function driver_attach.
- In platform_driver_register function, you can see that "platform_drv_probe" function is assigned to driver.probe function.
- This probe function will finally call your registered platform driver probe function.

I2C example

- A very commonly used low-speed bus to connect on-board and external 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.

probe()

- 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.

I2C 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;
```

I2C remove function

```
static 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);
}
```

How Addressing Works

Devices that are addressable use the following properties to encode address information into the device tree:

- reg
- #address-cells
- #size-cells

- Each addressable device gets a reg which is a list of tuples in the form
  - reg = <address1 length1 [address2 length2] [address3 length3] ... >

    Each tuple represents an address range used by the device. Each address value is a list of one or more 32 bit integers called cells. Similarly, the length value can either be a list of cells, or empty.

- Since both the address and length fields are variable of variable size, the #address-cells and #size-cells properties in the parent node are used to state how many cells are in each field. Interpreting a reg property correctly requires the parent node's #address-cells and #size-cells values.

CPU addressing

CPU nodes represent the simplest case of addressing. Each CPU is assigned a single unique ID, and there is no size associated with CPU ids.

```
cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a9";
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
        };
    };
```

In the cpus node, #address-cells is set to 1, and #size-cells is set to 0. This means that child reg values are a single uint32 that represent the address with no size field. In this case, the two cpus are assigned addresses 0 and 1. #size-cellsis 0 for cpu nodes because each cpu is only assigned a single address.

Reg value matches the value in the node name.

Memory Mapped Devices

```
/dts-v1/;

/ {
    #address-cells = <1>;
    #size-cells = <1>;

    ...

    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
    };

    serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
    };

    gpio@101f3000 {
        compatible = "arm,pl061";
        reg = <0x101f3000 0x1000
               0x101f4000 0x0010>;
    };

    interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
    };

    spi@10115000 {
        compatible = "arm,pl022";
        reg = <0x10115000 0x1000 >;
    };

    ...

};
```

- Instead of single address values like found in the cpu nodes, a memory mapped device is assigned a range of addresses that it will respond to.
- #size-cells is used to state how large the length field is in each child reg tuple.
- In this example, each address value is 1 cell (32 bits), and each length value is also 1 cell, which is typical on 32 bit systems.
- 64 bit machines may use a value of 2 for #address-cells and #size-cells to get 64 bit addressing in the device tree.
- Each device is assigned a base address, and the size of the region it is assigned. The GPIO device address in this example is assigned two address ranges; 0x101f3000...0x101f3fff and 0x101f4000..0x101f400f.

Devices on external bus

```
external-bus {
        #address-cells = <2>;
        #size-cells = <1>;

        ethernet@0,0 {
            compatible = "smc,smc91c111";
            reg = <0 0 0x1000>;
        };

        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
            };
        };

        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
            reg = <2 0 0x4000000>;
        };
    };
```

- Some devices live on a bus with a different addressing scheme. E.g. a device can be attached to an external bus with discrete chip select lines.
- Since each parent node defines the addressing domain for its children, the address mapping can be chosen to best describe the system.
- This code shows address assignment for devices attached to the external bus with the chip select number encoded into the address.
- The external-bus uses 2 cells for the address value; one for the chip select number, and one for the offset from the base of the chip select.
- The length field remains as a single cell since only the offset portion of the address needs to have a range. So, in this example, each reg entry contains 3 cells; the chipselect number, the offset, and the length.
- Since the address domains are contained to a node and its children, parent nodes are free to define whatever addressing scheme makes sense for the bus. Nodes outside of the immediate parent and child nodes do not normally have to care about the local addressing domain, and addresses have to be mapped to get from one domain to another.

Non memory mapped devices

```
i2c@1,0 {
        compatible = "acme,a1234-i2c-bus";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <1 0 0x1000>;
        rtc@58 {
            compatible = "maxim,ds1338";
            reg = <58>;
        };
    };
```

- Some devices are not memory mapped on the processor bus.
- They can have address ranges, but they are not directly accessible by the CPU. Instead the parent device's driver would perform indirect access on behalf of the CPU.
- To take the example of i2c devices, each device is assigned an address, but there is no length or range associated with it. This looks much the same as CPU address assignments.

Address Translation

```
/dts-v1/;

/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    ...
    external-bus {
        #address-cells = <2>
        #size-cells = <1>;
        ranges = <0 0  0x10100000   0x10000     // Chipselect 1, Ethernet
                  1 0  0x10160000   0x10000     // Chipselect 2, i2c controller
                  2 0  0x30000000   0x1000000>; // Chipselect 3, NOR Flash

        ethernet@0,0 {
            compatible = "smc,smc91c111";
            reg = <0 0 0x1000>;
        };

        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
                reg = <58>;
            };
        };

        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
            reg = <2 0 0x4000000>;
        };
    };
};
```

- ranges is a list of address translations.
- Each entry in the ranges table is a tuple containing the child address, the parent address, and the size of the region in the child address space.
- The size of each field is determined by taking the child's #address-cells value, the parent's #address-cells value, and the child's #size-cells value. For the external bus in our example, the child address is 2 cells, the parent address is 1 cell, and the size is also 1 cell. Three ranges are being translated:
    - Offset 0 from chip select 0 is mapped to address range 0x10100000..0x1010ffff
    - Offset 0 from chip select 1 is mapped to address range 0x10160000..0x1016ffff
    - Offset 0 from chip select 2 is mapped to address range 0x30000000..0x30ffffff
- There is no ranges property in the i2c@1,0 node. Unlike the external bus, devices on the i2c bus are not memory mapped on the CPU's address domain. Instead, the CPU indirectly accesses the rtc@58 device via the i2c@1,0 device.

Interrupt Related Information

```
/dts-v1/;

/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a9";
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
        };
    };

    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
        interrupts = < 1 0 >;
    };

    serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
        interrupts = < 2 0 >;
    };

    gpio@101f3000 {
        compatible = "arm,pl061";
        reg = <0x101f3000 0x1000
               0x101f4000 0x0010>;
        interrupts = < 3 0 >;
    };

    intc: interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
        interrupt-controller;
        #interrupt-cells = <2>;
    };
```

Four properties are used to describe interrupt connections:

- interrupt-controller - An empty property declaring a node as a device that receives interrupt signals
- #interrupt-cells - This is a property of the interrupt controller node. It states how many cells are in an interrupt specifier for this interrupt controller
- interrupt-parent - A property of a device node containing a phandle to the interrupt controller that it is attached to. Nodes that do not have an interrupt-parent property can also inherit the property from their parent node.
- interrupts - A property of a device node containing a list of interrupt specifiers, one for each interrupt output signal on the device. Each device uses an interrupt property to specify a different interrupt input line.
- #interrupt-cells is 2, so each interrupt specifier has 2 cells. This example uses the common pattern of using the first cell to encode the interrupt line number, and the second cell to encode flags such as active high vs. active low, or edge vs. level sensitive.

Device specific data

- Beyond the common properties, arbitrary properties and child nodes can be added to nodes. Any data needed by the operating system can be added as long as some rules are followed.

- First, new device-specific property names should use a manufacture prefix so that they don't conflict with existing standard property names.

- Second, the meaning of the properties and child nodes must be documented in a binding so that a device driver author knows how to interpret the data. A binding documents what a particular compatible value means, what properties it should have, what child nodes it might have, and what device it represents. Each unique compatible value should have its own binding (or claim compatibility with another compatible value).

Special nodes

aliases Node

A specific node is normally referenced by the full path, like /external-bus/ethernet@0,0, but that gets cumbersome when what a user really wants to know is, "which device is eth0?" The aliases node can be used to assign a short alias to a full device path. For example:

```
aliases {
    ethernet0 = &eth0;
    serial0 = &serial0;
};
```

The operating system is welcome to use the aliases when assigning an identifier to a device.

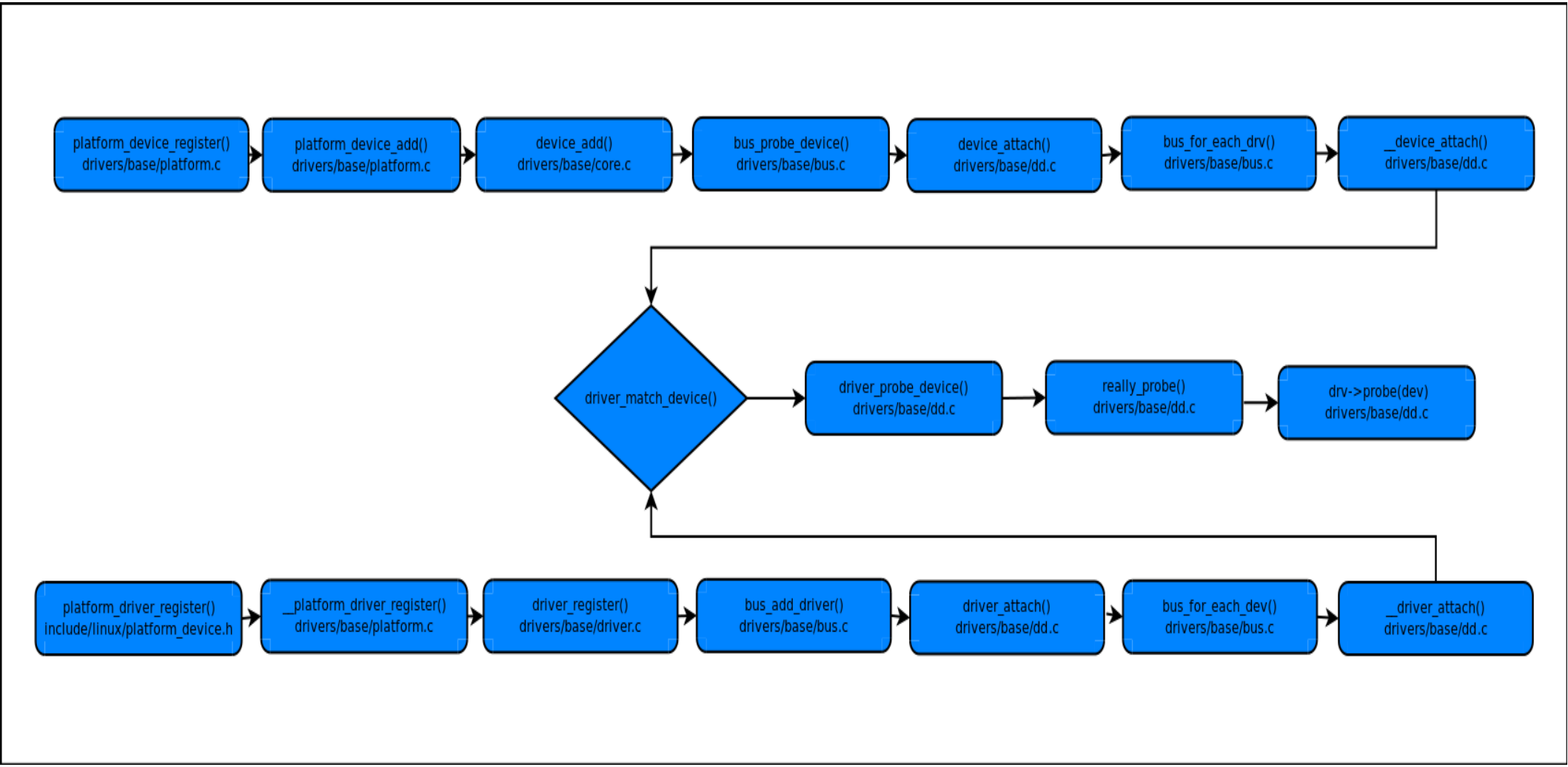
chosen Node

The chosen node doesn't represent a real device, but serves as a place for passing data between firmware and the operating system, like boot arguments. Data in the chosen node does not represent the hardware. Typically the chosen node is left empty in .dts source files and populated at boot time.

In our example system, firmware might add the following to the chosen node:

```
chosen {
    bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
};
```

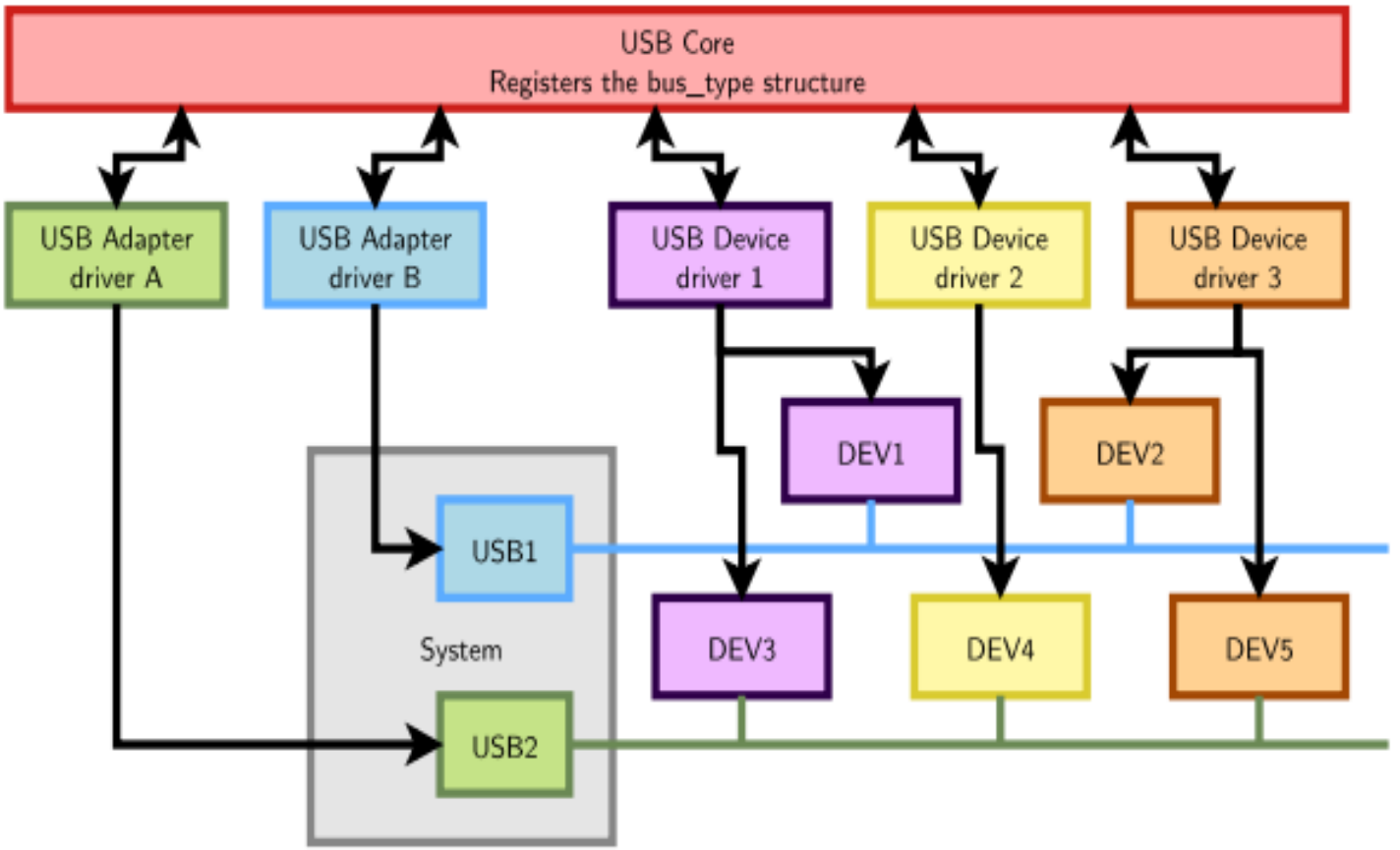
Device and Driver Matching, followed by probing



Bus driver for non-platform devices (detectable)

- E.g. USB or PCI
- 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

A high level USB controller driver



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

Device Driver

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 Driver (contd.)

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
};
```

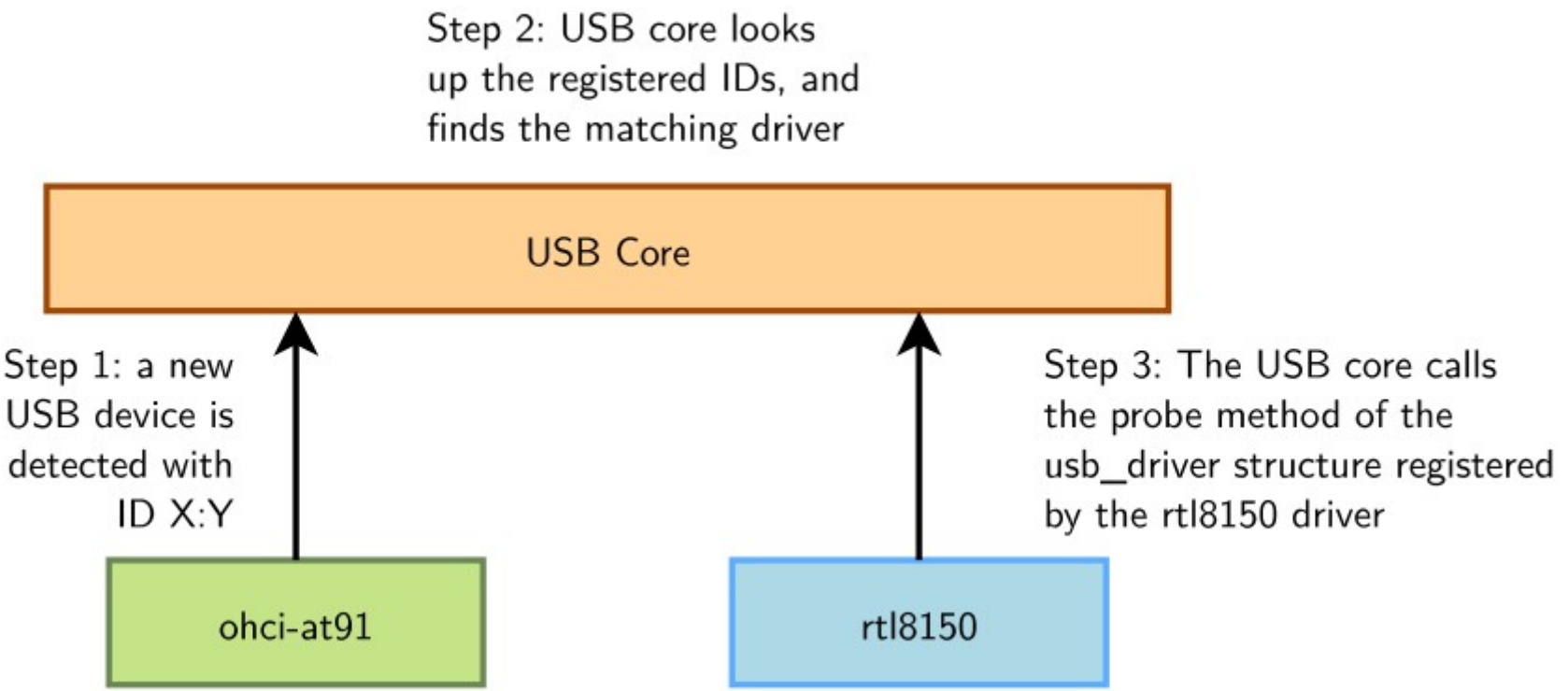
Device Driver (contd.)

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);
```

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

When a device is detected on bus



At driver loading time