Developing kernel modules

Hello module 1/2

```c
// SPDX-License-Identifier: GPL-2.0
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
  pr_alert("Good morrow to this fair assembly.\n");
  return 0;
}

static void __exit hello_exit(void)
{
  pr_alert("Alas, poor world, what treasure hast thou lost!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR("William Shakespeare");
```

Hello module 2/2

- Code marked as `__init`:
  - Removed after initialization (static kernel or module.)
  - See how init memory is reclaimed when the kernel finishes booting:

```
[    2.689854] VFS: Mounted root (nfs filesystem) on device 0:15.
[    2.698796] devtmpfs: mounted
[    2.704277] Freeing unused kernel memory: 1024K
[    2.710136] Run /sbin/init as init process
```

- Code marked as `__exit`:
  - Discarded when module compiled statically into the kernel, or when module unloading support is not enabled.
- Code of this example module available on
  https://raw.githubusercontent.com/bootlin/training-materials/master/code/hello/hello.c

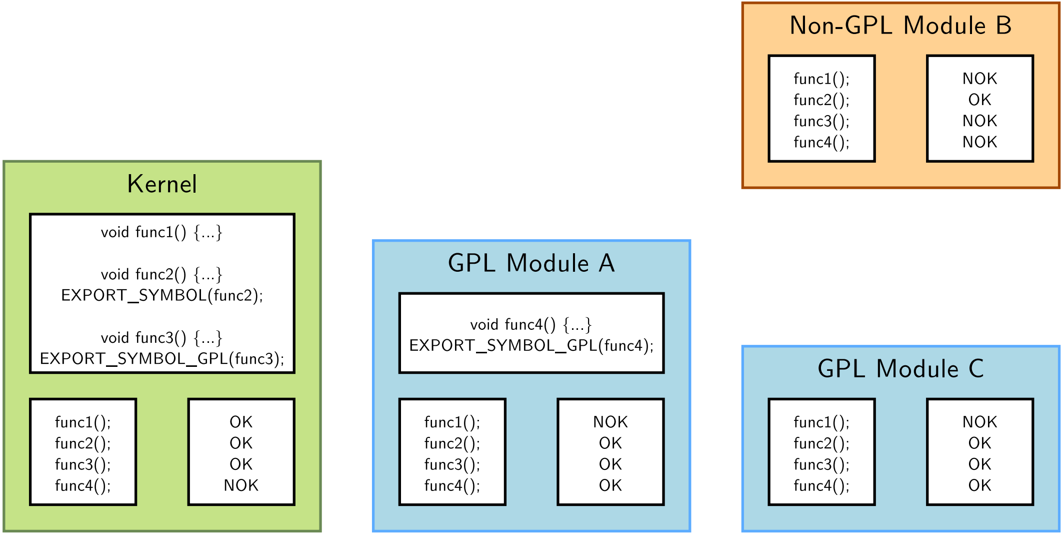
Hello module explanations

- Headers specific to the Linux kernel: `linux/xxx.h`
  - No access to the usual C library, we're doing kernel programming
- An initialization function
  - Called when the module is loaded, returns an error code (`0` on success, negative value on failure)
  - Declared by the `module_init()` macro: the name of the function doesn't matter, even though `<modulename>_init()` is a convention.
- A cleanup function
  - Called when the module is unloaded
  - Declared by the `module_exit()` macro.
- Metadata information declared using `MODULE_LICENSE()`, `MODULE_DESCRIPTION()` and `MODULE_AUTHOR()`

Symbols exported to modules 1/2

- From a kernel module, only a limited number of kernel functions can be called
- Functions and variables have to be explicitly exported by the kernel to be visible to a kernel module
- Two macros are used in the kernel to export functions and variables:
  - `EXPORT_SYMBOL(symbolname)`, which exports a function or variable to all modules
  - `EXPORT_SYMBOL_GPL(symbolname)`, which exports a function or variable only to GPL modules
  - Linux 5.3: contains the same number of symbols with `EXPORT_SYMBOL()` and symbols with `EXPORT_SYMBOL_GPL()`
- A normal driver should not need any non-exported function.

Symbols exported to modules 2/2



Module license

- Several usages
  - Used to restrict the kernel functions that the module can use if it isn't a GPL licensed module
    - Difference between `EXPORT_SYMBOL()` and `EXPORT_SYMBOL_GPL()`
  - Used by kernel developers to identify issues coming from proprietary drivers, which they can't do anything about ("Tainted" kernel notice in kernel crashes and oopses).
  - See `admin-guide/tainted-kernels` for details about `tainted` flag values.
  - Useful for users to check that their system is 100% free (for the kernel, check `/proc/sys/kernel/tainted`; run `vrms` to check installed packages)
- Values
  - GPL compatible (see `include/linux/license.h`: `GPL`, `GPL v2`, `GPL and additional rights`, `Dual MIT/GPL`, `Dual BSD/GPL`, `Dual MPL/GPL`)
  - `Proprietary`

Compiling a module

Two solutions

- *Out of tree*, when the code is outside of the kernel source tree, in a different directory
  - Not integrated into the kernel configuration/compilation process
  - Needs to be built separately
  - The driver cannot be built statically, only as a module
- Inside the kernel tree
  - Well integrated into the kernel configuration/compilation process
  - The driver can be built statically or as a module

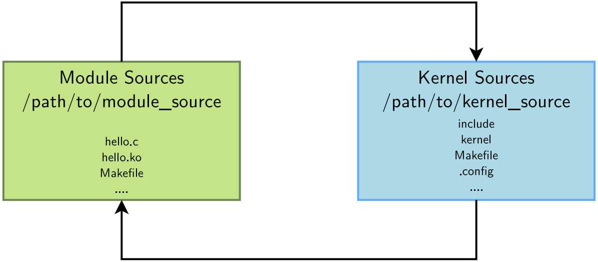
Compiling an out-of-tree module 1/2

- The below `Makefile` should be reusable for any single-file out-of-tree Linux module (see Building External Modules for other options)
- The source file is `hello.c`
- Just run `make` to build the `hello.ko` file

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources

all:
<tab>$(MAKE) -C $(KDIR) M=$$PWD
endif
```

- `KDIR`: kernel source or headers directory (see next slides)

Compiling an out-of-tree module 2/2



- The module `Makefile` is interpreted with `KERNELRELEASE` undefined, so it calls the kernel `Makefile`, passing the module directory in the `M` variable
- The kernel `Makefile` knows how to compile a module, and thanks to the `M` variable, knows where the `Makefile` for our module is. This module `Makefile` is then interpreted with `KERNELRELEASE` defined, so the kernel sees the `obj-m` definition.

Modules and kernel version

- To be compiled, a kernel module needs access to *kernel headers*, containing the definitions of functions, types and constants.
- Two solutions
  - Full kernel sources (configured + `make modules_prepare`)
  - Only kernel headers (`linux-headers-*` packages in Debian/Ubuntu distributions, or directory created by `make headers_install`).
- The sources or headers must be configured (`.config` file)
  - Many macros or functions depend on the configuration
- You also need the kernel `Makefile`, the `scripts/` directory, and a few others.
- A kernel module compiled against version X of kernel headers will not load in kernel version Y
  - `modprobe` / `insmod` will say `Invalid module format`

New driver in kernel sources 1/2

- To add a new driver to the kernel sources:
  - Add your new source file to the appropriate source directory. Example: drivers/usb/serial/navman.c
  - Single file drivers in the common case, even if the file is several thousand lines of code big. Only really big drivers are split in several files or have their own directory.
  - Describe the configuration interface for your new driver by adding the following lines to the Kconfig file in this directory:

```
config USB_SERIAL_NAVMAN
        tristate "USB Navman GPS device"
        depends on USB_SERIAL
        help
          To compile this driver as a module, choose M
          here: the module will be called navman.
```

New driver in kernel sources 2/2

- Add a line in the `Makefile` file based on the `Kconfig` setting:
  `obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o`
- It tells the kernel build system to build `navman.c` when the `USB_SERIAL_NAVMAN` option is enabled. It works both if compiled statically or as a module.
  - Run `make xconfig` and see your new options!
  - Run `make` and your new files are compiled!
  - See `kbuild/` for details and more elaborate examples like drivers with several source files, or drivers in their own subdirectory, etc.

Hello module with parameters 1/2

```c
// SPDX-License-Identifier: GPL-2.0
/* hello_param.c */
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");

static char *whom = "world";
module_param(whom, charp, 0644);
MODULE_PARM_DESC(whom, "Recipient of the hello message");

static int howmany = 1;
module_param(howmany, int, 0644);
MODULE_PARM_DESC(howmany, "Number of greetings");
```

Hello module with parameters 2/2

```c
static int __init hello_init(void)
{
    int i;

    for (i = 0; i < howmany; i++)
        pr_alert("(%d) Hello, %s\n", i, whom);
    return 0;
}

static void __exit hello_exit(void)
{
    pr_alert("Goodbye, cruel %s\n", whom);
}

module_init(hello_init);
module_exit(hello_exit);
```

Thanks to Jonathan Corbet for the examples

Source code available on: https://github.com/bootlin/training-materials/blob/master/code/hello-param/hello_param.c

Declaring a module parameter

```
module_param(
    name, /* name of an already defined variable */
    type, /* standard types (different from C types) are:
           * byte, short, ushort, int, uint, long, ulong
           * charp: a character pointer
           * bool: a bool, values 0/1, y/n, Y/N.
           * invbool: the above, only sense-reversed (N = true). */
    perm  /* for /sys/module/<module_name>/parameters/<param>,
           *  0: no such module parameter value file */
);

/* Example: drivers/block/loop.c */
static int max_loop;
module_param(max_loop, int, 0444);
MODULE_PARM_DESC(max_loop, "Maximum number of loop devices");
```

Modules parameter arrays are also possible with `module_param_array()`.

Useful general-purpose kernel APIs

Memory/string utilities

- In include/linux/string.h
  - Memory-related: memset(), memcpy(), memmove(), memscan(), memcmp(), memchr()
  - String-related: strcpy(), strcat(), strcmp(), strchr(), strrchr(), strlen() and variants
  - Allocate and copy a string: kstrdup(), kstrndup()
  - Allocate and copy a memory area: kmemdup()
- In include/linux/kernel.h
  - String to int conversion: simple_strtoul(), simple_strtol(), simple_strtoull(), simple_strtoll()
  - Other string functions: sprintf(), sscanf()

Useful macros

- Most useful: `container_of()`, `ARRAY_SIZE()`
- Bit manipulation: `BIT()`, `BIT_MASK()`, `GENMASK()`
- Math: `DIV_ROUND_UP()`, `abs()`,...
- Endianness: `cpu_to_le32()`, `cpu_to_be32()`,...
- Build-time assert: `BUILD_BUG_ON()`, `BUILD_BUG_ON_MSG()`,...
- Run-time assert: `WARN()`, `WARN_ON()`,...
- Kconfig: `IS_ENABLED()`, `IS_BUILTIN()`, `IS_MODULE()`,...
- And many more, be curious !

Linked lists

- Convenient linked-list facility in `include/linux/list.h`
  - Used in thousands of places in the kernel
- Add a `struct list_head` member to the structure whose instances will be part of the linked list. It is usually named `node` when each instance needs to only be part of a single list.
- Define the list with the `LIST_HEAD()` macro for a global list, or define a `struct list_head` element and initialize it with `INIT_LIST_HEAD()` for lists embedded in a structure.
- Then use the `list_*()` API to manipulate the list
  - Add elements: `list_add()`, `list_add_tail()`
  - Remove, move or replace elements: `list_del()`, `list_move()`, `list_move_tail()`, `list_replace()`
  - Test the list: `list_empty()`
  - Iterate over the list: `list_for_each_*()` family of macros

Linked lists examples 1/2

From `include/soc/at91/atmel_tcb.h`

```
/*
 * Definition of a list element, with a
 * struct list_head member
 */
struct atmel_tc
{
    /* some members */
    struct list_head node;
};
```

Linked lists examples 2/2

From drivers/misc/atmel_tclib.c

```
/* Define the global list */
static LIST_HEAD(tc_list);

static int __init tc_probe(struct platform_device *pdev) {
    struct atmel_tc *tc;
    tc = kzalloc(sizeof(struct atmel_tc), GFP_KERNEL);
    /* Add an element to the list */
    list_add_tail(&tc->node, &tc_list);
}

struct atmel_tc *atmel_tc_alloc(unsigned block, const char *name)
{
    struct atmel_tc *tc;
    /* Iterate over the list elements */
    list_for_each_entry(tc, &tc_list, node) {
        /* Do something with tc */
    }
    [...]
}
```

Describing hardware devices

Discoverable hardware: USB and PCI

Discoverable hardware

- Some busses have built-in hardware discoverability mechanisms
- Most common busses: USB and PCI
- Hardware devices can be enumerated, and their characteristics retrieved with just a driver or the bus controller
- Useful Linux commands
  - `lsusb`, lists all USB devices detected
  - `lspci`, lists all PCI devices detected
  - A detected device does not mean it has a kernel driver associated to it!
- Association with kernel drivers done based on product ID/vendor ID, or some other characteristics of the device: device class, device sub-class, etc.

Describing non-discoverable hardware

Non-discoverable buses

- 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 buses or SPI buses, or the devices directly part of the system-on-chip.
- However, we still want all of these devices to be part of the device model.
- Such devices, instead of being dynamically detected, must be statically described.

Describing non-discoverable hardware

1. Directly in the **OS/bootloader code**

- Using compiled data structures, typically in C
- How it was done on most embedded platforms in Linux, U-Boot.
- Considered not maintainable/sustainable on ARM32, which motivated the move to another solution.

Describing non-discoverable hardware

2. Using **ACPI** tables

- On *x86* systems, but also on a subset of ARM64 platforms
- Tables provided by the firmware

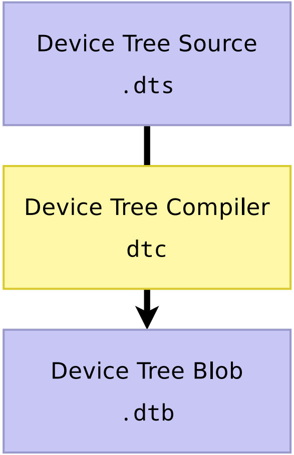
Describing non-discoverable hardware

3. Using a **Device Tree**

- Originates from **OpenFirmware**, defined by Sun, used on SPARC and PowerPC
  - That's why many Linux/U-Boot functions related to DT have a `of_` prefix
- Now used by most embedded-oriented CPU architectures that run Linux: ARC, ARM64, RISC-V, ARM32, PowerPC, Xtensa, MIPS, etc.
- Writing/tweaking a DT is necessary when porting Linux to a new board, or when connecting additional peripherals

Device Tree: from source to blob

- A tree data structure describing the hardware is written by a developer in a **Device Tree Source** file, `.dts`
- Processed by the **Device Tree Compiler**, `dtc`
- Produces a more efficient representation: **Device Tree Blob**, `.dtb`
- Additional C preprocessor pass
- `.dtb` → accurately describes the hardware platform in an **OS-agnostic** way.
- `.dtb` ≈ few dozens of kilobytes
- DTB also called **FDT**, *Flattened Device Tree*, once loaded into memory.
  - `fdt` command in U-Boot
  - `fdt_` APIs



dtc example

```
$ cat foo.dts
/dts-v1/;

/ {
        welcome = <0xBADCAFE>;
        bootlin {
                webinar = "great";
                demo = <1>, <2>, <3>;
        };
};
```

dtc example

```
$ cat foo.dts
/dts-v1/;

/ {
        welcome = <0xBADCAFE>;
        bootlin {
                webinar = "great";
                demo = <1>, <2>, <3>;
        };
};
```

```
$ dtc -I dts -O dtb -o foo.dtb foo.dts
$ ls -l foo.dt*
-rw-r--r-- 1 thomas thomas 169 ... foo.dtb
-rw-r--r-- 1 thomas thomas 102 ... foo.dts
```

dtc example

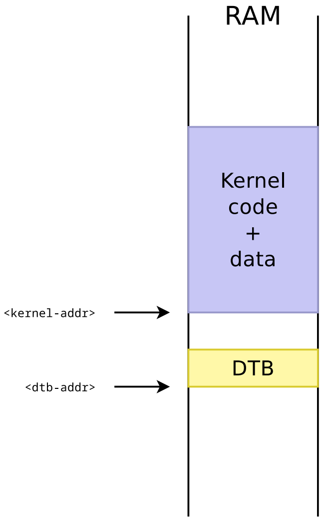
```
$ cat foo.dts
/dts-v1/;

/ {
        welcome = <0xBADCAFE>;
        bootlin {
                webinar = "great";
                demo = <1>, <2>, <3>;
        };
};
```

```
$ dtc -I dts -O dtb -o foo.dtb foo.dts
$ ls -l foo.dt*
-rw-r--r-- 1 thomas thomas 169 ... foo.dtb
-rw-r--r-- 1 thomas thomas 102 ... foo.dts
```

```
$ dtc -I dtb -O dts foo.dtb
/dts-v1/;

/ {
        welcome = <0xbadcafe>;

        bootlin {
                webinar = "great";
                demo = <0x01 0x02 0x03>;
        };
};
```

Device Tree: using the blob

- Can be **linked directly** inside a bootloader binary
  - For example: U-Boot, Barebox
- Can be **passed** to the operating system by the bootloader
  - Most common mechanism for the Linux kernel
  - U-Boot:
    `boot[z,i,m] <kernel-addr> - <dtb-addr>`
  - The bootloader can adjust the DTB before passing it to the kernel
- The DTB parsing can be done using `libfdt`, or ad-hoc code



Where are Device Tree Sources located?

- Even though they are OS-agnostic, **no central and OS-neutral** place to host Device Tree sources and share them between projects
  - Often discussed, never done
- In practice, the Linux kernel sources can be considered as the **canonical location** for Device Tree Source files
  - `arch/<ARCH>/boot/dts/<vendor>/`
  - `arch/arm/boot/dts` (on ARM 32 architecture before Linux 6.5)
  - $\approx$ 4500 Device Tree Source files (`.dts` and `.dtsi`) in Linux as of 6.0.
- Duplicated/synced in various projects
  - U-Boot, Barebox, TF-A

Device Tree base syntax

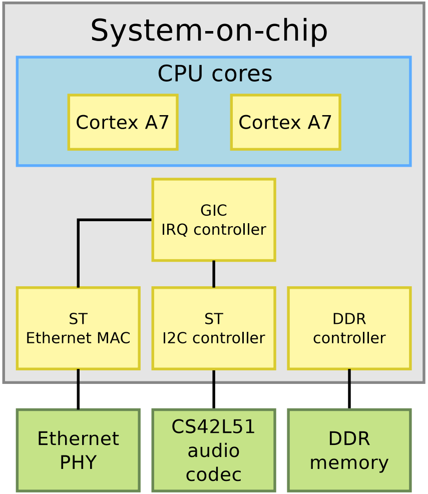
- Tree of **nodes**
- Nodes with **properties**
- Node ≈ a device or IP block
- Properties ≈ device characteristics
- Notion of **cells** in property values
- Notion of **phandle** to point to other nodes
- `dtc` only does syntax checking, no semantic validation

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];

        child-node@0 {
            first-child-property;
            second-child-property = <1>;
            a-reference-to-something = <&node1>;
        };

        child-node@1 {
        };
    };

    node1: node@1 {
        an-empty-property;
        a-cell-property = <1 2 3 4>;

        child-node@0 {
        };
    };
};
```

Node name
Unit address
Property name
Property value
Properties of node@0
Bytestring
A phandle (reference to another node)
Label
Four cells (32 bits values)

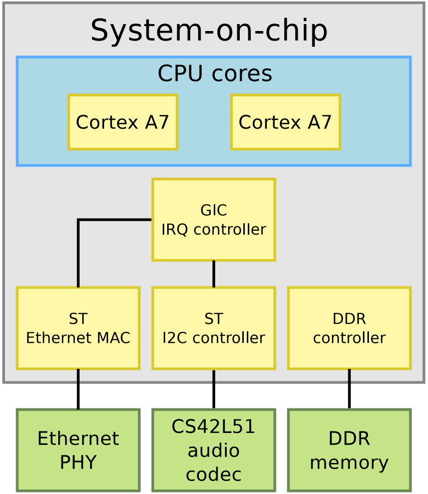
DT overall structure: simplified example

```
/ {
  #address-cells = <1>;
  #size-cells = <1>;
  model = "STMicroelectronics STM32MP157C-DK2 Discovery Board";
  compatible = "st,stm32mp157c-dk2", "st,stm32mp157";

  cpus { ... };
  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



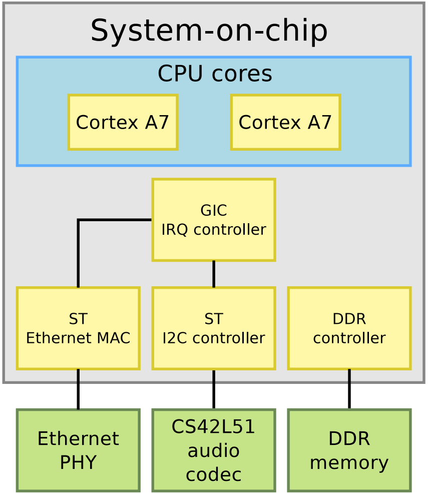
DT overall structure: simplified example

```
/ {
  cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu0: cpu@0 {
      compatible = "arm,cortex-a7";
      clock-frequency = <650000000>;
      device_type = "cpu";
      reg = <0>;
    };

    cpu1: cpu@1 {
      compatible = "arm,cortex-a7";
      clock-frequency = <650000000>;
      device_type = "cpu";
      reg = <1>;
    };
  };

  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



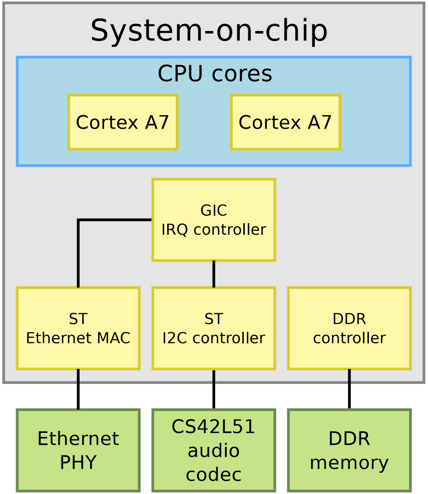
DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0 {
    device_type = "memory";
    reg = <0x0 0x20000000>;
  };

  chosen {
    bootargs = "";
    stdout-path = "serial0:115200n8";
  };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



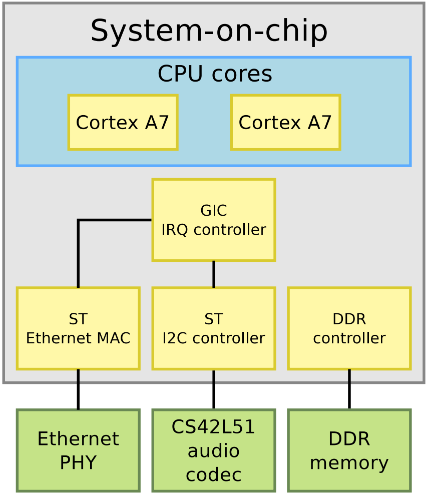
DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0 { ... };
  chosen { ... };

  intc: interrupt-controller@a0021000 {
    compatible = "arm,cortex-a7-gic";
    #interrupt-cells = <3>;
    interrupt-controller;
    reg = <0xa0021000 0x1000>,
          <0xa0022000 0x2000>;
  };

  soc {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;

    i2c1: i2c@40012000 { ... };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



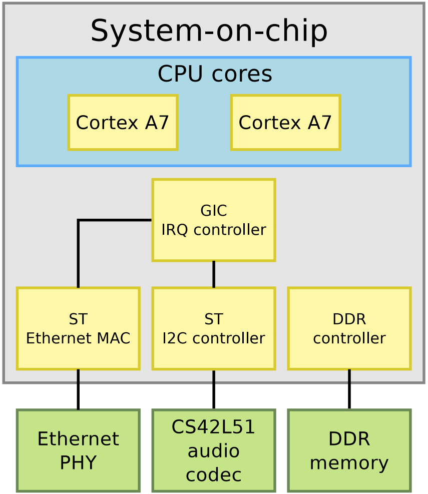
DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    i2c1: i2c@40012000 {
      compatible = "st,stm32mp15-i2c";
      reg = <0x40012000 0x400>;
      interrupts = <GIC_SPI 31 IRQ_TYPE_LEVEL_HIGH>,
                   <GIC_SPI 32 IRQ_TYPE_LEVEL_HIGH>;
      #address-cells = <1>;
      #size-cells = <0>;
      status = "okay";

      cs42l51: cs42l51@4a {
        compatible = "cirrus,cs42l51";
        reg = <0x4a>;
        reset-gpios = <&gpiog 9 GPIO_ACTIVE_LOW>;
        status = "okay";
      };
    };
    ethernet0: ethernet@5800a000 { ... };
  };
};
```



DT overall structure: simplified example

```
/ {
  cpus { ... };
  memory@0 { ... };
  chosen { ... };
  intc: interrupt-controller@a0021000 { ... };
  soc {
    compatible = "simple-bus";
    ...
    interrupt-parent = <&intc>;
    i2c1: i2c@40012000 { ... };

    ethernet0: ethernet@5800a000 {
      compatible = "st,stm32mp1-dwmac", "snps,dwmac-4.20a";
      reg = <0x5800a000 0x2000>;
      interrupts-extended = <&intc GIC_SPI 61 IRQ_TYPE_LEVEL_HIGH>;
      status = "okay";

      mdio0 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "snps,dwmac-mdio";
        phy0: ethernet-phy@0 {
          reg = <0>;
        };
      };
    };
  };
};
```



Device Tree inheritance

- Device Tree files are not monolithic, they can be split in several files, including each other.
- `.dtsi` files are included files, while `.dts` files are *final* Device Trees
  - Only `.dts` files are accepted as input to `dtc`
- Typically, `.dtsi` will contain
  - definitions of SoC-level information
  - definitions common to several boards
- The `.dts` file contains the board-level information
- The inclusion works by **overlaying** the tree of the including file over the tree of the included file, according to the order of the `#include` directives.
- Allows an including file to **override** values specified by an included file.
- Uses the C pre-processor #include directive

Device Tree inheritance example

Definition of the AM33xx SoC family

```
&l4_wkup {
    target-module@b000 {
        i2c0: i2c@0 {
            compatible = "ti,omap4-i2c";
            reg = <0x0 0x1000>;
            interrupts = <70>;
            status = "disabled";
        };
    };
};
```

am33xx-l4.dtsi

+

Definition of the Bone Black board

```
#include "am33xx-l4.dtsi"

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

    baseboard_eeprom: eeprom@50 {
        compatible = "atmel,24c256";
        reg = <0x50>;
    };
};
```

am335x-boneblack.dts

=

Compiled DTB

```
&l4_wkup {
    target-module@b000 {
        i2c0: i2c@0 {
            compatible = "ti,omap4-i2c";
            reg = <0x0 0x1000>;
            interrupts = <70>;
            pinctrl-names = "default";
            pinctrl-0 = <&i2c0_pins>;
            status = "okay";

            baseboard_eeprom: eeprom@50 {
                compatible = "atmel,24c256";
                reg = <0x50>;
            };
        };
    };
};
```

am335x-boneblack.dtb

Note 1

The actual Device Trees for this
platform are more complicated.
This example is highly simplified.

Note 2

The real DTB is in binary format.
Here we show the text equivalent of the
DTB contents.

Doing:

soc.dtsi

```
/ {
  soc {
    usart1: serial@5c000000 {
      compatible = "st,stm32h7-uart";
      reg = <0x5c000000 0x400>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

/ {
  soc {
    serial@5c000000 {
      status = "okay";
    };
  };
};
```

Inheritance and labels

Doing:

soc.dtsi

```
/ {
  soc {
    usart1: serial@5c000000 {
      compatible = "st,stm32h7-uart";
      reg = <0x5c000000 0x400>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

/ {
  soc {
    serial@5c000000 {
      status = "okay";
    };
  };
};
```

Is exactly equivalent to:

soc.dtsi

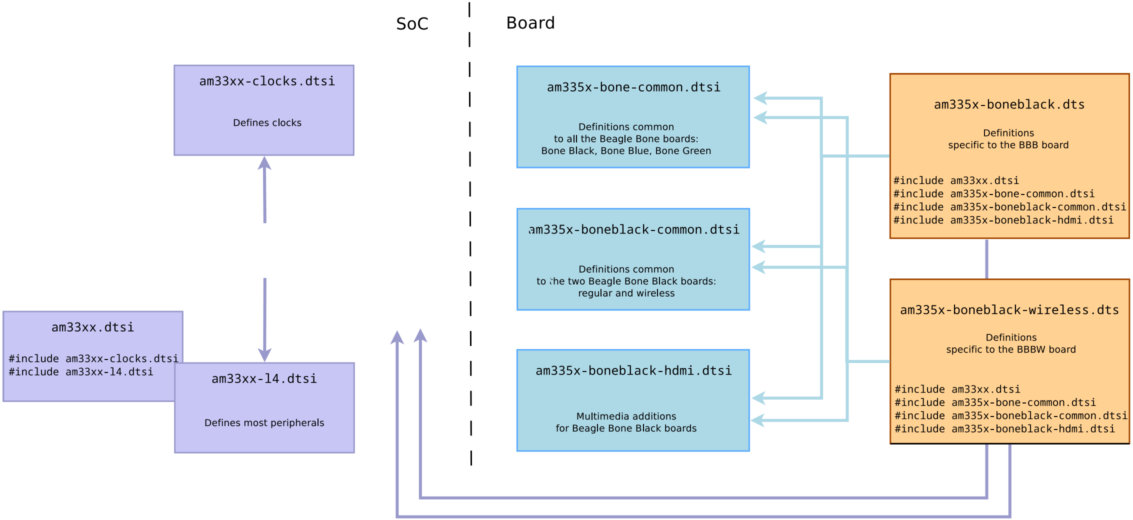
```
/ {
  soc {
    usart1: serial@5c000000 {
      compatible = "st,stm32h7-uart";
      reg = <0x5c000000 0x400>;
      status = "disabled";
    };
  };
};
```

board.dts

```
#include "soc.dtsi"

&usart1 {
  status = "okay";
};
```

$\rightarrow$ this solution is now often preferred

DT inheritance in Bone Black support

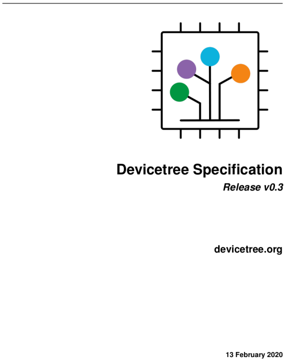


Device Tree design principles

- **Describe hardware** (how the hardware is), not configuration (how I choose to use the hardware)
- **OS-agnostic**
  - For a given piece of HW, Device Tree should be the same for U-Boot, FreeBSD or Linux
  - There should be no need to change the Device Tree when updating the OS
  - The Device Tree is de facto part of the kernel ABI
- Describe **integration of hardware components**, not the internals of hardware components
  - The details of how a specific device/IP block is working is handled by code in device drivers
  - The Device Tree describes how the device/IP block is connected/integrated with the rest of the system: IRQ lines, DMA channels, clocks, reset lines, etc.
- Like all beautiful design principles, these principles are sometimes violated.

Device Tree specifications

- How to write the correct nodes/properties to describe a given hardware platform ?
- **DeviceTree Specifications** → base Device Tree syntax + number of standard properties.
  - https://www.devicetree.org/specifications/
  - Not sufficient to describe the wide variety of hardware.
- **Device Tree Bindings** → documents that each specify how a piece of HW should be described
  - Documentation/devicetree/bindings/ in Linux kernel sources
  - Reviewed by DT bindings maintainer team
  - Legacy: human readable documents
  - New norm: YAML-written specifications


Devicetree Specification
Release v0.3

devicetree.org

13 February 2020

Device Tree binding: legacy style

Documentation/devicetree/bindings/i2c/i2c-omap.txt

```
I2C for OMAP platforms

-Required properties :
- compatible : Must be
       "ti,omap2420-i2c" for OMAP2420 SoCs
       "ti,omap2430-i2c" for OMAP2430 SoCs
       "ti,omap3-i2c" for OMAP3 SoCs
       "ti,omap4-i2c" for OMAP4+ SoCs
       "ti,am654-i2c", "ti,omap4-i2c" for AM654 SoCs
       "ti,j721e-i2c", "ti,omap4-i2c" for J721E SoCs
       "ti,am64-i2c", "ti,omap4-i2c" for AM64 SoCs
- ti,hwmods : Must be "i2c<n>", n being the instance number (1-based)
- #address-cells = <1>;
- #size-cells = <0>;

Recommended properties :
- clock-frequency : Desired I2C bus clock frequency in Hz. Otherwise
  the default 100 kHz frequency will be used.

Optional properties:
- Child nodes conforming to i2c bus binding

Note: Current implementation will fetch base address, irq and dma
from omap hwmod data base during device registration.
Future plan is to migrate hwmod data base contents into device tree
blob so that, all the required data will be used from device tree dts
file.
```

```
Examples :

i2c1: i2c@0 {
    compatible = "ti,omap3-i2c";
    #address-cells = <1>;
    #size-cells = <0>;
    ti,hwmods = "i2c1";
    clock-frequency = <400000>;
};
```

Device Tree binding: YAML style

Documentation/devicetree/bindings/i2c/ti,omap4-i2c.yaml

```yaml
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
%YAML 1.2
---
$id: http://devicetree.org/schemas/i2c/ti,omap4-i2c.yaml#
$schema: http://devicetree.org/meta-schemas/core.yaml#

title: I2C controllers on TI's OMAP and K3 SoCs

maintainers:
  - Vignesh Raghavendra <vigneshr@ti.com>

properties:
  compatible:
    oneOf:
      - enum:
          - ti,omap2420-i2c
          - ti,omap2430-i2c
          - ti,omap3-i2c
          - ti,omap4-i2c
      - items:
          - enum:
              - ti,am4372-i2c
              - ti,am64-i2c
              - ti,am654-i2c
              - ti,j721e-i2c
          - const: ti,omap4-i2c

  reg:
    maxItems: 1
```

```yaml
  interrupts:
    maxItems: 1

  clocks:
    maxItems: 1

  clock-names:
    const: fck

  clock-frequency: true

  power-domains: true

  "#address-cells":
    const: 1

  "#size-cells":
    const: 0

  ti,hwmods:
    description:
      Must be "i2c<n>", n being [...]
    $ref: /schemas/types.yaml#/definitions/string
    deprecated: true

required:
  - compatible
  - reg
  - interrupts
```

```yaml
additionalProperties: false

if:
  properties:
    compatible:
      enum:
        - ti,omap2420-i2c
        - ti,omap2430-i2c
        - ti,omap3-i2c
        - ti,omap4-i2c
then:
  properties:
    ti,hwmods:
      items:
        - pattern: "^i2c([1-9])$"
else:
  properties:
    ti,hwmods: false

examples:
  - |
    #include <dt-bindings/interrupt-controller/irq.h>
    #include <dt-bindings/interrupt-controller/arm-gic.h>

    main_i2c0: i2c@2000000 {
        compatible = "ti,j721e-i2c", "ti,omap4-i2c";
        reg = <0x2000000 0x100>;
        interrupts = <GIC_SPI 200 IRQ_TYPE_LEVEL_HIGH>;
    };
```

Validating Device Tree in Linux

- `dtc` only does syntactic validation
- YAML bindings allow to do semantic validation
- Linux kernel `make` rules:
  - `make dt_binding_check`
    verify that YAML bindings are valid
  - `make dtbs_check`
    validate DTs currently enabled against YAML bindings
  - `make DT_SCHEMA_FILES=Documentation/devicetree/bindings/trivial-devices.yaml dtbs_check`
    validate DTs against a specific YAML binding

The compatible property

- Is a list of strings
  - From the most specific to the least specific
- Describes the specific **binding** to which the node complies.
- It uniquely identifies the **programming model** of the device.
- Practically speaking, it is used by the operating system to find the **appropriate driver** for this device.
- When describing real hardware, the typical form is `vendor,model`
- Examples:
  - `compatible = "arm,armv7-timer";`
  - `compatible = "st,stm32mp1-dwmac", "snps,dwmac-4.20a";`
  - `compatible = "regulator-fixed";`
  - `compatible = "gpio-keys";`
- Special value: `simple-bus` → bus where all sub-nodes are memory-mapped devices

reg property

- Most important property after `compatible`
- **Memory-mapped** devices: base physical address and size of the memory-mapped registers. Can have several entries for multiple register areas.

```
sai4: sai@50027000 {
    reg = <0x50027000 0x4>, <0x500273f0 0x10>;
};
```

reg property

- Most important property after `compatible`
- **Memory-mapped** devices: base physical address and size of the memory-mapped registers. Can have several entries for multiple register areas.
- **I2C** devices: address of the device on the I2C bus.

```
&i2c1 {
    hdmi-transmitter@39 {
        reg = <0x39>;
    };
    cs42l51: cs42l51@4a {
        reg = <0x4a>;
    };
};
```

reg property

- Most important property after `compatible`
- **Memory-mapped** devices: base physical address and size of the memory-mapped registers. Can have several entries for multiple register areas.
- **I2C** devices: address of the device on the I2C bus.
- **SPI** devices: chip select number

```
&qspi {
        flash0: mx66l51235l@0 {
                reg = <0>;
        };
        flash1: mx66l51235l@1 {
                reg = <1>;
        };
};
```

reg property

- Most important property after `compatible`
- **Memory-mapped** devices: base physical address and size of the memory-mapped registers. Can have several entries for multiple register areas.
- **I2C** devices: address of the device on the I2C bus.
- **SPI** devices: chip select number
- The unit address must be the address of the first `reg` entry.

```
sai4: sai@50027000 {
    reg = <0x50027000 0x4>, <0x500273f0 0x10>;
};
```

cells property

- Property numbers shall fit into 32-bit containers called `cells`
- The compiler does not maintain information about the number of entries, the OS just receives 4 independent `cells`
  - Example with a `reg` property using 2 entries of 2 cells:

```
reg = <0x50027000 0x4>, <0x500273f0 0x10>;
```

  - The OS cannot make the difference with:

```
reg = <0x50027000>, <0x4>, <0x500273f0>, <0x10>;
reg = <0x50027000 0x4 0x500273f0>, <0x10>;
reg = <0x50027000>, <0x4 0x500273f0 0x10>;
reg = <0x50027000 0x4 0x500273f0 0x10>;
```

cells property

- Property numbers shall fit into 32-bit containers called `cells`
- The compiler does not maintain information about the number of entries, the OS just receives 4 independent `cells`
- Need for other properties to declare the right formatting:
  - `#address-cells`: Indicates the number of cells used to carry the address
  - `#size-cells`: Indicates the dimension of the address range. `0`: one address, `1`: address range (interval), `2`: multiple address ranges.
- The parent-node declares the children `reg` property formatting
  - Platform devices need memory ranges

```
module@a0000 {
    #address-cells = <1>;
    #size-cells = <1>;

    serial@1000 {
        reg = <0x1000 0x10>, <0x2000 0x10>;
    };
};
```

cells property

- Property numbers shall fit into 32-bit containers called `cells`
- The compiler does not maintain information about the number of entries, the OS just receives 4 independent `cells`
- Need for other properties to declare the right formatting:
  - `#address-cells`: Indicates the number of cells used to carry the address
  - `#size-cells`: Indicates the dimension of the address range. `0`: one address, `1`: address range (interval), `2`: multiple address ranges.
- The parent-node declares the children `reg` property formatting
  - Platform devices need memory ranges
  - SPI devices need chip-selects

```
spi@300000 {
    #address-cells = <1>;
    #size-cells = <0>;

    flash@1 {
        reg = <1>;
    };
}
```

Status property

- The `status` property indicates if the device is really in use or not
  - `okay` or `ok` → the device is really in use
  - any other value, by convention `disabled` → the device is not in use
- In Linux, controls if a device is instantiated
- In `.dtsi` files describing SoCs: all devices that interface to the outside world have `status = disabled`
- Enabled on a per-device basis in the board `.dts`

Resources: interrupts, clocks, DMA, reset lines, ...

- Common pattern for resources shared by multiple hardware blocks
  - Interrupt lines
  - Clock controllers
  - DMA controllers
  - Reset controllers
  - ...
- A Device Tree node describing the *controller* as a device
- References from other nodes that use resources provided by this *controller*

```
intc: interrupt-controller@a0021000 {
   compatible = "arm,cortex-a7-gic";
   #interrupt-cells = <3>;
   interrupt-controller;
   reg = <0xa0021000 0x1000>, <0xa0022000 0x2000>;
};

rcc: rcc@50000000 {
   compatible = "st,stm32mp1-rcc", "syscon";
   reg = <0x50000000 0x1000>;
   #clock-cells = <1>;
   #reset-cells = <1>;
};

dmamux1: dma-router@48002000 {
   compatible = "st,stm32h7-dmamux";
   reg = <0x48002000 0x1c>;
   #dma-cells = <3>;
   clocks = <&rcc DMAMUX>;
   resets = <&rcc DMAMUX_R>;
};

spi3: spi@4000c000 {
   interrupts = <GIC_SPI 51 IRQ_TYPE_LEVEL_HIGH>;
   clocks = <&rcc SPI3_K>;
   resets = <&rcc SPI3_R>;
   dmas = <&dmamux1 61 0x400 0x05>,  <&dmamux1 62 0x400 0x05>;
};
```

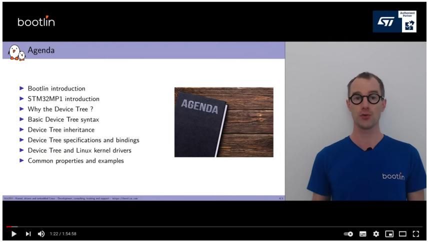
Generic suffixes

- `xxx-gpios`
  - When drivers need access to GPIOs
  - May be subsystem-specific or vendor-specific
  - Examples: `enable-gpios`, `cts-gpios`, `rts-gpios`
- `xxx-names`
  - Sometimes naming items is relevant
  - Allows drivers to perform lookups by name rather than ID
  - The order of definition of each item still matters
  - Examples: `gpio-names`, `clock-names`, `reset-names`

```
uart0@4000c000 {
    dmas = <&edma 26 0>, <&edma 27 0>;
    dma-names = "tx", "rx";
    ...
};
```

References

- Device Tree 101 webinar, Thomas Petazzoni (2021):
  Slides: https://bootlin.com/blog/device-tree-101-webinar-slides-and-videos/
  Video: https://youtu.be/a9CZ1Uk3OYQ
- Kernel documentation
  - driver-api/driver-model/
  - devicetree/
  - filesystems/sysfs
- https://devicetree.org
- The kernel source code
  - Full of examples of other drivers!



Linux device and driver model

Introduction

The need for a device model?

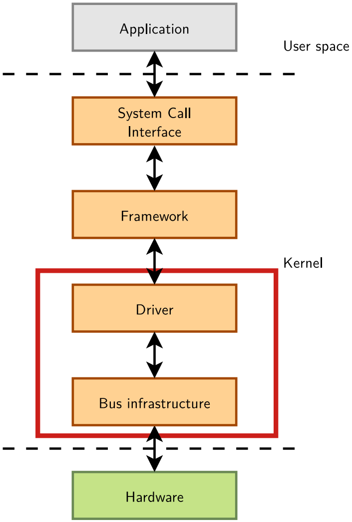
- The Linux kernel runs on a wide range of architectures and hardware platforms, and therefore needs to **maximize the reusability** of code between platforms.
- For example, we want the same *USB device driver* to be usable on a x86 PC, or an ARM platform, even though the USB controllers used on these 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.

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 *bus infrastructure*, while *kernel frameworks* are covered later in this training.



Device Model data structures

- The *device model* is organized around three main data structures:
  - The `struct bus_type` structure, which represents 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.
- In order to explore the device model, we will
  - First look at a popular bus that offers dynamic enumeration, the *USB bus*
  - Continue by studying how buses that do not offer dynamic enumeration are handled.

Bus Drivers

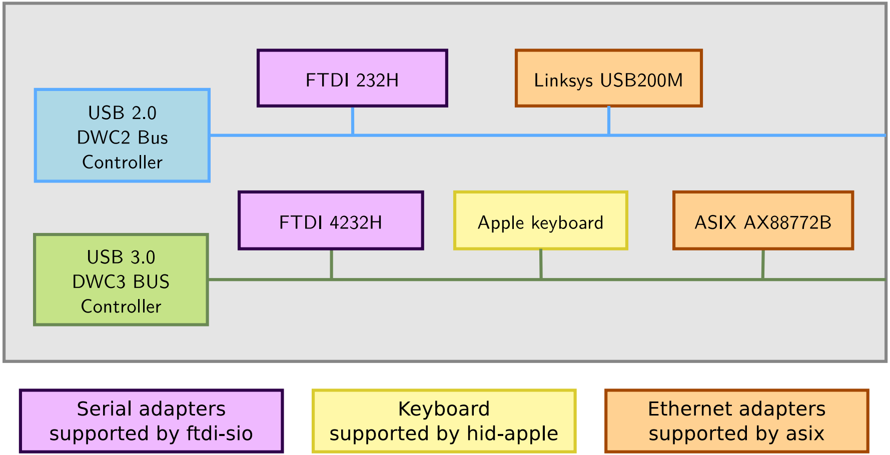
- The first component of the device model is the bus driver
  - One bus driver for each type of bus: USB, PCI, SPI, MMC, I2C, etc.
- It is responsible for
  - Registering the bus type (`struct bus_type`)
  - Allowing the registration of adapter drivers (USB controllers, I2C adapters, etc.), able to detect the connected devices (if possible), 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 implement both adapter drivers and device drivers
  - Defining driver and device specific structures, mainly `struct usb_driver` and `struct usb_interface`

sysfs

- The bus, device, drivers, etc. structures are internal to the kernel
- The `sysfs` virtual filesystem offers a mechanism to export such information to user space
- Used for example by `udev` to provide automatic module loading, firmware loading, mounting of external media, 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 the framework they are registered to (`net`, `input`, `block`...), whatever bus they are connected to. Very useful!

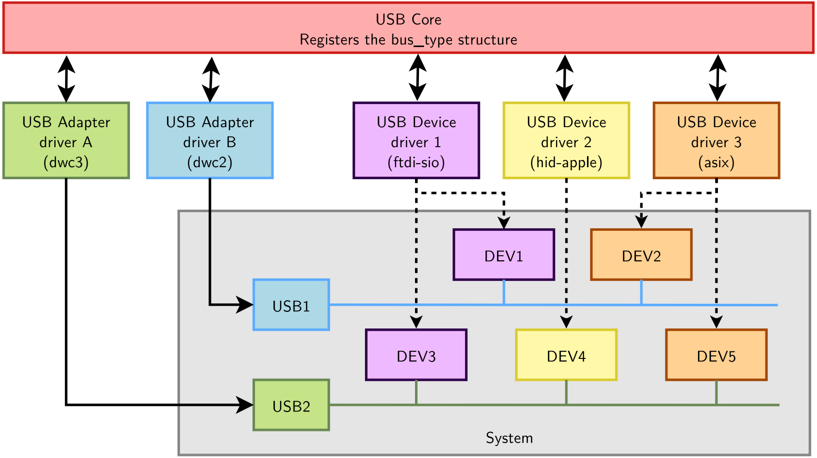
Example of the USB bus

Example: USB Bus 1/3



Hardware view of the bus

Example: USB Bus 2/3



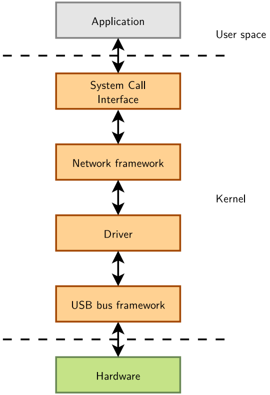
Device model view of the bus

Example: USB Bus 3/3

- 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/`)

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 exposes a network device, so it has to be a network driver
  - 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` (run by `make modules_install`) to extract the relationship 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) },
    { 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 device_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 registration and unregistration

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

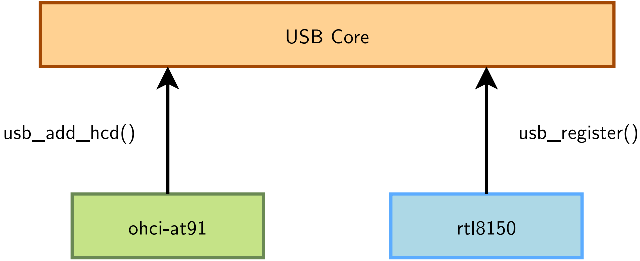
```
static int __init usb_rtl8150_init(void)
{
    return usb_register(&rtl8150_driver);
}

static void __exit usb_rtl8150_exit(void)
{
    usb_deregister(&rtl8150_driver);
}

module_init(usb_rtl8150_init);
module_exit(usb_rtl8150_exit);
```

- All this code is actually replaced by a call to the `module_usb_driver()` macro:

```
module_usb_driver(rtl8150_driver);
```

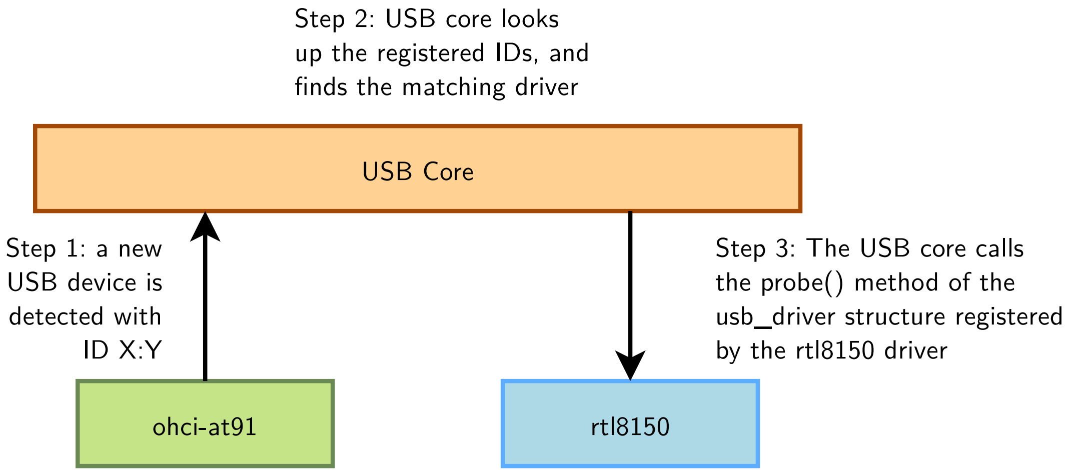
At Initialization

- 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



Probe Method

- Invoked **for each device** bound to a driver
- The `probe()` method receives as argument a structure describing the device, usually specialized by the bus infrastructure (`struct pci_dev`, `struct 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.

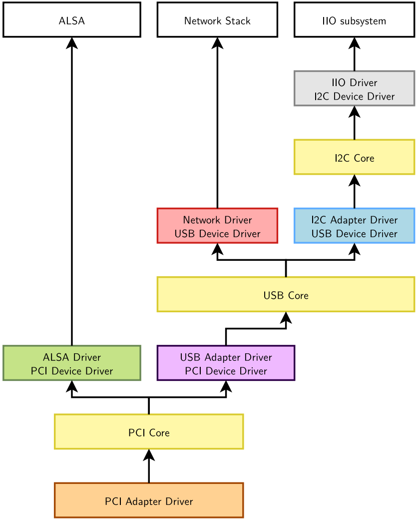
Example: probe() and disconnect() methods

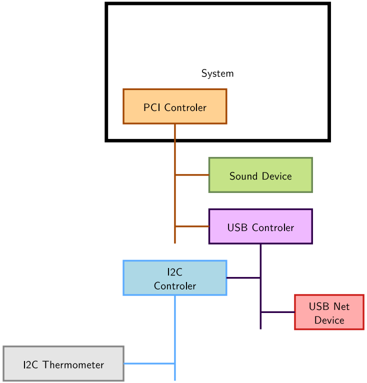
```
static int rtl8150_probe(struct usb_interface *intf,
    const struct usb_device_id *id)
{
    rtl8150_t *dev;
    struct net_device *netdev;

    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;
}
```

```
static void rtl8150_disconnect(struct usb_interface *intf)
{
        rtl8150_t *dev = usb_get_intfdata(intf);

        usb_set_intfdata(intf, NULL);
        if (dev) {
                set_bit(RTL8150_UNPLUG, &dev->flags);
                tasklet_kill(&dev->tl);
                unregister_netdev(dev->netdev);
                unlink_all_urbs(dev);
                free_all_urbs(dev);
                free_skb_pool(dev);
                if (dev->rx_skb)
                        dev_kfree_skb(dev->rx_skb);
                kfree(dev->intr_buff);
                free_netdev(dev->netdev);
        }
}
```

Source: drivers/net/usb/rtl8150.c

The Model is Recursive





Platform drivers

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 of a Platform Driver (1)

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

```
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,
        },
};
```

Implementation of a Platform Driver (2)

... and registers its driver to the platform driver infrastructure

```
static int __init imx_serial_init(void) {
    return platform_driver_register(&serial_imx_driver);
}

static void __exit imx_serial_cleanup(void) {
    platform_driver_unregister(&serial_imx_driver);
}

module_init(imx_serial_init);
module_exit(imx_serial_cleanup);
```

Most drivers actually use the `module_platform_driver()` macro when they do nothing special in `init()` and `exit()` functions:

```
module_platform_driver(serial_imx_driver);
```

Platform device instantiation

- As platform devices cannot be detected dynamically, they are defined statically
  - Legacy way: by direct instantiation of `struct platform_device` structures, as done on a few old ARM platforms. The device was part of a list, and the list of devices was added to the system during board initialization.
  - Current way: by parsing an "external" description, like a *device tree* on most embedded platforms today, from which `struct platform_device` structures are created.

compatible property and Linux kernel drivers

- Linux identifies as **platform devices**:
  - Top-level DT nodes with a `compatible` string
  - Sub-nodes of `simple-bus`
    - Instantiated automatically at boot time
- Sub-nodes of I2C controllers → *I2C devices*
- Sub-nodes of SPI controllers → *SPI devices*
- Each Linux driver has a table of compatible strings it supports
  - `struct of_device_id[]`
- When a DT node compatible string matches a given driver, the device is *bound* to that driver.

```
/ {
  timer {                          Platform device
    compatible = "...";
  };
  soc {
    compatible = "simple-bus";
    uart@1000 {                    Platform device
      compatible = "...";
    };
    i2c@2000 {                     Platform device
      compatible = "...";
      eeprom@65 {                  I2C device
        compatible = "...";
      };
    };
  };
};
```

Matching with drivers in Linux: platform driver

drivers/i2c/busses/i2c-omap.c

```
static const struct of_device_id omap_i2c_of_match[] = {
	{
		.compatible = "ti,omap4-i2c",
		.data = &omap4_pdata,
	},
	{
		.compatible = "ti,omap3-i2c",
		.data = &omap3_pdata,
	},
	[...]
	{ },
};
MODULE_DEVICE_TABLE(of, omap_i2c_of_match);

[...]

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 with drivers in Linux: I2C driver

sound/soc/codecs/cs42151.c

```
const struct of_device_id cs42l51_of_match[] = {
        { .compatible = "cirrus,cs42l51", },
        { }
};
MODULE_DEVICE_TABLE(of, cs42l51_of_match);
```

sound/soc/codecs/cs42151-i2c.c

```
static struct i2c_driver cs42l51_i2c_driver = {
        .driver = {
                .name = "cs42l51",
                .of_match_table = cs42l51_of_match,
                .pm = &cs42l51_pm_ops,
        },
        .probe = cs42l51_i2c_probe,
        .remove = cs42l51_i2c_remove,
        .id_table = cs42l51_i2c_id,
};
```

Using additional hardware resources

- Regular DT descriptions contain many information, including phandles (pointers) towards additional hardware blocks or hardware details which cannot be discovered.
  - Some of them are available through a generic array of resourses, like addresses for the I/O registers and IRQ lines:
    - Such information can be represented using `struct resource`, and an array of `struct resource` is associated to each `struct platform_device`.
  - Common information/dependencies are parsed by the relevant subsystems, like clocks, GPIOs, or DMA channels:
    - Each subsystem is responsible of instantiating its components, and offering an API to retrieve these objects and use them from device drivers.
  - Specific information might be directly be retrieved by device drivers, through (expensive) direct DT lookups (old drivers use `struct platform_data`).
- All these methods allow the same driver to be used with multiple devices functioning similarly, but with different addresses, IRQs, etc.

Using Resources

- The platform driver has access to the resources:

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

- As well as the various common dependencies through individual APIs:
  - clk_get()
  - gpio_request()
  - dma_request_channel()

Driver data

- In addition to the per-device resources and information, drivers may require driver-specific information to behave slighlty differently when different flavors of an IP block are driven by the same driver.
- A `const void *data` pointer can be used to store per-compatible specificities:

```
static const struct of_device_id marvell_nfc_of_ids[] = {
        {
                .compatible = "marvell,armada-8k-nand-controller",
                .data = &marvell_armada_8k_nfc_caps,
        },
};
```

- Which can be retrieved in the probe with:

```
        /* Get NAND controller capabilities */
        if (pdev->id_entry) /* legacy way */
                nfc->caps = (void *)pdev->id_entry->driver_data;
        else /* current way */
                nfc->caps = of_device_get_match_data(&pdev->dev);
```