



Device Tree (DT)

See this article:

[A Tutorial on the Device Tree \(Zynq\) -- Part I](#)

Source: [Device Tree Usage Model](#)

“The "Open Firmware Device Tree", or simply **Device Tree (DT)**, is a data structure and language for describing hardware. More specifically, it is a description of hardware that is readable by an operating system so that the operating system doesn't need to hard code details of the machine.

...

An operating system used the Device Tree to discover the topology of the hardware at runtime, and thereby support a majority of available hardware without hard coded information (assuming drivers were available for all devices).”

...

<<

Common Abbreviations

OF = Open Firmware

DT = Device Tree

FDT = Flattened Device Tree

DTB = Device Tree Binary -or- Device Tree Blob

dts = Device Tree Source

dtso = Device Tree Source files for Included files (usually contain SoC-level definitions)

dtc = device tree compiler (tool)

>>

2.1 High Level View

The most important thing to understand is that the DT is simply a data structure that describes the hardware. There is nothing magical about it, and it doesn't magically make all hardware configuration problems go away. What it does do is provide **a language for decoupling the hardware configuration from the board and device driver support in the Linux kernel** (or any other operating system for that matter). Using it allows board and device support to **become data driven**; to make setup decisions based on data passed into the kernel instead of on per-machine hard coded selections.


Ideally, data driven platform setup should result in less code duplication and make it easier to support a wide range of hardware with a single kernel image.

Linux uses DT data for three major purposes:

- 1) platform identification,
- 2) runtime configuration, and
- 3) device population.

...

[Source \(below\)](#)

 **Booting**

- ▶ The kernel no longer contains the description of the hardware, it is located in a separate binary: the *device tree blob*
- ▶ The bootloader loads two binaries: the kernel image and the *DTB*
 - ▶ Kernel image remains *uImage* or *zImage*
 - ▶ DTB located in *arch/arm/boot/dts*, one per board
- ▶ The bootloader passes the DTB address through *r2*. It is supposed to adjust the DTB with memory information, kernel command line, and potentially other info.
- ▶ No more *machine type*.
- ▶ U-Boot command:
`boot[mz] <kernel img addr> - <dtb addr>`
- ▶ Barebox variables: *bootm.image*, *bootm.oftree*

Device Trees in ARM Linux

More and more manufacturers are adopting device trees in linux. Let's see What and Why device trees?

If you notice in the recent past the *arch/arm/mach-xxx/* folder is becoming very clumsy as the number of boards every vendor is supporting is increasing drastically and every board has its own board specific files for maintaining the data and the number files in this folder is increasing. Also, the maintenance is becoming increasingly difficult.

Most of the drivers available today are written in a way that they are **board independent**. All they **need is the board (platform) specific information to be passed** to them when the *xxx_probe()* is called. The most popular way to send this was using the ***platform_data (pdata)* data structure**.

<<

For example, on a recent 3.10.6 vanilla Linux kernel source tree:

```
$ grep 'platform_data.*=.*{' arch/arm/mach-*/*.ch | wc -l
1356
$ grep 'platform_data.*=.*{' arch/arm/mach-*/*.ch
arch/arm/mach-at91/at91rm9200_devices.c:472:static struct i2c_gpio_platform_data pdata = {
arch/arm/mach-at91/at91sam9260_devices.c:382:static struct i2c_gpio_platform_data pdata = {
arch/arm/mach-at91/at91sam9261_devices.c:278:static struct i2c_gpio_platform_data pdata = {
arch/arm/mach-at91/at91sam9263_devices.c:560:static struct i2c_gpio_platform_data pdata = {

--snip--

arch/arm/mach-vexpress/v2m.c:141:static struct isp1760_platform_data v2m_usb_config = {
arch/arm/mach-vexpress/v2m.c:206:static struct mmci_platform_data v2m_mmci_data = {
arch/arm/mach-w90x900/dev.c:251:static struct flash_platform_data nuc900_spi_flash_data = {
arch/arm/mach-w90x900/dev.c:395:static struct w90p910_keypad_platform_data nuc900_keypad_info = {
$
```

>>

And this is populated in the board specific files (*arch/arm/mach-xxxx/board-xxx.x* << or equivalent >>) for every board. So, every different board that is made in the same family of SoC needs board specific files. This also leads to a situation that one kernel built for a board does not work when used with another board using the same SoC family. Successful companies making a lot of products need to maintain a lot of different ROM's.

Device trees came in to solve these problems stated above. Device trees have been existing in kernel for a while now. Recently it is being used in ARM based SoC's as well.

When device trees are used, the platform specific data is not passed from the board specific files. It is **passed at boot time** like the ATAG parameters. This ensures that **one kernel built for the platform can work on several different boards just by changing the device tree data passed** to them in one or other method of passing device tree.

The device tree files are called as **dtb** (device tree blob). These are maintained in a separate partition, and read and put into the RAM by the boot loader. Or, it can as well be packaged along with the kernel.

<<

[*Source: "Platform devices and device trees", LWN*](#)

“... It is possible for platform devices to work on a device-tree-enabled system **with no extra work at all**, especially once [Grant Likely's improvements](#) are merged.

If the device tree includes a platform device (where such devices, in the device tree context, are those which are direct children of the root or are attached to a "simple bus"), that device will be instantiated and matched against a driver. The memory-mapped I/O and interrupt resources will be **marshalled from the device tree description and made available to the device's *probe()* function in the usual way.** The driver need not know that the device was instantiated out of a device tree rather than from a hard-coded platform device definition.

...”

>>

During boot, the device trees are parsed as part of the *init/main.c:start_kernel()* routine. The *setup_arch()* function calls the *setup_machine_fdt()* in the *arch/arm/kernel* folder where the device tree is parsed and put into data structures.

The device drivers use functions like *of_find_property()*, *of_property_read_xx()* etc to get the data from corresponding to it's device node. This way the board specific *platform_data* files are **eliminated**.

But, what is described above is a ideal scenario. **Not all drivers support device trees yet** and hence board specific files **may not** be completely eliminated at this time. Over a period of time I guess most drivers will move to device trees. The format of the tree is standard and can found in the websites.

There are tools that are available that can generate device trees from FPGA designs. If such tools become popular and work well, then device trees could become even more popular.

Posted by [Narayanan Gopalakrishnan](#).

Now read this:

- [Device Tree Usage](#)
- [6. Devicetree Source \(DTS\) Format \(version 1\)](#) – nicely shown and explained
- [Raspberry Pi : DEVICE TREES, OVERLAYS AND PARAMETERS](#) [btw, HAT=Hardware Attached on Top, a new feature of the R Pi B+]
- Linux kernel documentation:
[Device Tree Usage Model](#) ← “This article describes how Linux uses the device tree.”

Raspberry Pi:

<https://www.raspberrypi.org/documentation/configuration/device-tree.md>

TIP-

Runtime check:

Dump the currently used DTB in source form (dts) like this:

```
dtc -I fs /proc/device-tree
```

dtc is the DT Compiler.

-I : input style format; fs => from proc fs */proc/device-tree*

DT usage

- Identification of the platform at system boot

ARM:

- kernel looks to match an entry in the machine descriptor's (in DT_MACHINE_START) `dt_compat` list to a `compatible` string entry in the DT
- DT matching done at early boot in *arch/arm/kernel/setup.c:setup_arch()*

Eg. Raspberry Pi; the SoC is the Broadcom BCM2835 / 2836 / 2837 (latter two for Rpi 3 ARMv7)

```
arch/arm/mach-bcm/board_bcm2835.c:
static const char * const bcm2835_compat[] = {
#ifdef CONFIG_ARCH_MULTI_V6
    "brcm,bcm2835",
#endif
#ifdef CONFIG_ARCH_MULTI_V7
    "brcm,bcm2836",
    "brcm,bcm2837",           << R Pi 3 matches this compatible string prop! >>
#endif
    NULL
};

DT_MACHINE_START(BCM2835, "BCM2835")
    .dt_compat = bcm2835_compat,
    .smp = smp_ops(bcm2836_smp_ops),
MACHINE_END
```

The Raspberry Pi 3 Model B+ DT is here:

arch/arm/boot/dts/bcm2837-rpi-3-b-plus.dts

```
...
/ {
    compatible = "raspberrypi,3-model-b-plus", "brcm,bcm2837";
    model = "Raspberry Pi 3 Model B+";
```

```

chosen {
    /* 8250 auxiliary UART instead of pl011 */
    stdout-path = "serial1:115200n8";    << passed as part of kernel
                                         cmdline! >>
    << can use 'bootargs = "..."' to pass kernel cmdline args >>
};
...

```

<https://www.kernel.org/doc/Documentation/devicetree/bindings/chosen.txt>

“The **chosen** node does not 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. ...”

Useful resource for DT syntax:

[6. Devicetree Source \(DTS\) Format \(version 1\)](#) – nicely shown and explained

All slides (with Tux Sr & Jr @ upper-left :-)) below are from here:

[Device Trees for Dummies, Thomas Petazzoni](#)



Basic Device Tree syntax

```

/ {
    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 {
        };
    };
};

```

Diagram annotations:

- Node name:** node@0
- Unit address:** 0
- Property name:** a-string-property
- Property value:** "A string"
- Properties of node@0:** a-string-property, a-string-list-property, a-byte-data-property
- Bytestring:** [0x01 0x23 0x34 0x56]
- A phandle (reference to another node):** <&node1>
- Label:** node1
- Four cells (32 bits values):** <1 2 3 4>



DT is hardware description, not configuration

- ▶ The Device Tree is really a hardware description language.
- ▶ It should **describe the hardware layout**, and how it works.
- ▶ But it should **not describe which particular hardware configuration** you're interested in.
- ▶ As an example:
 - ▶ You may describe in the DT whether a particular piece of hardware supports DMA or not.
 - ▶ But you may not describe in the DT if you *want* to use DMA or not.



DT bindings as an ABI

- ▶ Since the DT is OS independent, it should also be stable.
- ▶ The original idea is that DTBs can be flashed on some devices by the manufacturer, so that the user can install whichever operating system it wants.
- ▶ Once a Device Tree binding is defined, and used in DTBs, it should no longer be changed anymore. It can only be extended.
- ▶ This normally means that **Device Tree bindings become part of the kernel ABI**, and it should be handled with the same care.
- ▶ However, kernel developers are realizing that this is really hard to achieve and slowing down the integration of drivers.
 - ▶ The ARM Kernel Mini-summit discussions have relaxed those rules.
 - ▶ There will be additional discussions during the Kernel Summit, with final conclusions published afterwards.



Basic guidelines for binding design

- ▶ **A precise compatible string is better than a vague one**
 - ▶ You have a driver that covers both variants T320 and T330 of your hardware. You may be tempted to use `foo,t3xx` as your compatible string.
 - ▶ Bad idea: what if T340 is slightly different, in an incompatible way? You'd better use `foo,t320` for both T320 and T330.
- ▶ **Do not encode too much hardware details in the DT**
 - ▶ When two hardware variants are quite similar, some developers are tempted to encode all the differences in the DT, including register offsets, bit masks or offsets.
 - ▶ Bad idea: it makes the binding more complex, and therefore less resilient to future changes. Instead, use two different compatible strings and handle the differences in the driver.

Resources

[Platform devices and device trees \[LWN\]](#)

[Device Trees for Dummies, Thomas Petazzoni](#)

[Device Tree Usage](#) – learn the device tree format, with an example building up a device tree describing a simple hardware platform (OS-agnostic)

Linux kernel documentation:

[Device Tree Usage Model](#)

[Bootimg without OF \(OpenFirmware\)](#)

← “This article describes how Linux uses the device tree.”

<http://xillybus.com/tutorials/device-tree-zynq-1> [5-part tutorial]

Misc

<https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt>

...
 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).

...

<<

Useful to read the example DT snippet and explanation of the same in the above documentation

>>

...
 The trick is that the kernel starts at the root of the tree and looks for nodes that have a 'compatible' property. First, it is generally assumed that any node with a 'compatible' property represents a device of some kind, and second, it can be assumed that any node at the root of the tree is either directly attached to the processor bus, or is a miscellaneous system device that cannot be described any other way. **For each of these nodes, Linux allocates and registers a platform_device, which in turn may get bound to a platform_driver.**

...

Source

...

compatible properties are the link between the hardware description and the driver software. When an OS encounters a node with a **compatible** property, it looks it up in its database of device drivers to find the best match. In Linux, this **usually results in the driver module being automatically loaded**, provided it has been appropriately labelled and not blacklisted.

The **status** property indicates whether a device is enabled or disabled. If the **status** is ok, okay or absent, then the device is enabled. Otherwise, **status** should be **disabled**, so the device will be disabled. It can be useful to place devices in a **.dtsi** file with the status set to **disabled**. A derived configuration can then include that **.dtsi** and set the status for the devices which are needed to **okay**.

...

- **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>".

- *DT snippet :*

```
...
memory {
    device_type = "memory";
    reg = <0x0 0x80000000>;           // 128M at 0x0
};
...
flash@0,0 {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "cfi-flash";
    reg = <0x0 0x0 0x1000000>;
    bank-width = <2>;
    device-width = <2>;
    partition@0x0 {
        label = "free space";
        reg = <0x00000000 0x00f80000>;
    };
    partition@0x100000 {
        label = "bootloader";
        reg = <0x00f80000 0x00080000>;
        read-only;
    };
};
```

<<

The properties called `#address-cells` and `#size-cells` are worth explaining. A "cell" in this context is simply a 32-bit quantity. `#address-cells` and `#size-cells` simply indicate the number of cells (32-bit fields) required to specify an address (or size) in the child node.

>>

How does the device driver interact with the DT?

Ref: https://elinux.org/Device_Tree_Linux

There are several ways to; some are:

- Hardware Version in struct of `_device_id.data`
- Function Call Table pointer in struct of `_device_id.data`
- Hardware Description pointer in struct of `_device_id.data`
- FDT built into kernel as data

(See examples in the reference web page).

Also, see the ‘Device Tree for Dummies’ book pages 13 – 15.

Where are the device tree “source files” on Linux?

One can locate **.dts** (device tree source) within the *arch/<architecture>/boot/dts* folder. For example on ARM:

```
$ find arch/arm -name '*.dts' |wc -l
208
$ find arch/arm -name '*.dts' | sort -t '/' -k5
arch/arm/boot/dts/aks-cdu.dts
arch/arm/boot/dts/am335x-bone.dts
arch/arm/boot/dts/am335x-evm.dts
...
arch/arm/boot/dts/versatile-ab.dts
arch/arm/boot/dts/versatile-pb.dts
arch/arm/boot/dts/vexpress-v2p-ca15_a7.dts
arch/arm/boot/dts/vexpress-v2p-ca9.dts
...
arch/arm/boot/dts/xenvm-4.2.dts
arch/arm/boot/dts/zynq-zc702.dts
$
```

<< On 4.16.0, the number of *dts* files under *arch/arm* has rocketed to 1055! >>

- ... The device tree source code is **compiled by the dtc tool** to the compact binary form expected by the *of_xxx()* routines. The tool finds and publicizes errors, albeit sometimes with cryptic messages.

... u-boot takes the binary device tree, adds decorations to reflect the hardware (e.g., describing the amount of installed system RAM), then passes it to the Linux kernel.

...

- [Source](#) [Documentation/devicetree/usage-model.txt]

...

```
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).**

...

2.4 Device population

 After the board has been identified, and after the early configuration data has been parsed, then kernel initialization can proceed in the normal way. At some point in this process, `unflatten_device_tree()` is called to convert the data into a more efficient runtime representation. This is also when machine-specific setup hooks will get called, like the `machine_desc .init_early()`, `.init_irq()` and `.init_machine()` hooks on ARM.

...

The most interesting hook in the DT context is `.init_machine()` which is primarily responsible for **populating the Linux device model with data about the platform**. Historically this has been implemented on embedded platforms by defining a set of static clock structures, `platform_devices`, and other data in the board support `.c` file, and registering it en-masse in `.init_machine()`. When DT is used, then instead of hard coding static devices for each platform, the list of devices can be obtained by parsing the DT, and allocating device structures dynamically.

The simplest case is when `.init_machine()` is only responsible for registering a block of `platform_devices`. A `platform_device` is a concept used by Linux for memory or I/O mapped devices which cannot be detected by hardware, and for 'composite' or 'virtual' devices ...

...

The trick is that the kernel starts at the root of the tree and **looks** for nodes that have a **'compatible' property**. First, it is generally assumed that any node with a 'compatible' property represents a device of some kind, and second, it can be assumed that any node at the root of the tree is either directly attached to the processor bus, or is a miscellaneous system device that cannot be described any other way. **For each of these nodes, Linux allocates and registers a `platform_device`, which in turn may get bound to a `platform_driver`.**

...

Linux board support code calls `of_platform_populate(NULL, NULL, NULL, NULL)` to kick off discovery of devices at the root of the tree.

...

<<

More recently, it's the function
`drivers/of/platform.c:of_platform_default_populate()`
 >>

In the Tegra example, this accounts for the `/soc` and `/sound` nodes, but what about the children of the SoC node? Shouldn't they be registered as platform devices too? For Linux DT support, the generic behaviour

is for child devices to be registered by the parent's device driver at driver .probe() time. So, an i2c bus device driver will register a i2c_client for each child node, an SPI bus driver will register its spi_device children, and similarly for other bus_types. According to that model, a driver could be written that binds to the SoC node and simply registers platform_devices for each of its children. The board support code would allocate and register an SoC device, a (theoretical) SoC device driver could bind to the SoC device, and register platform_devices for /soc/interrupt-controller, /soc/serial, /soc/i2s, and /soc/i2c in its .probe() hook. Easy, right?

Actually, it turns out that registering children of some platform_devices as more platform_devices is a common pattern, and the device tree support code reflects that and makes the above example simpler. The second argument to of_platform_populate() is an of_device_id table, and any node that matches an entry in that table will also get its child nodes registered.

...

DT Binding: how exactly does a driver “parse” a DT and extract relevant hardware information from it?

We use the ARM matrix keypad (input) platform driver as an example:

drivers/input/keyboard/matrix_keypad.c

<< the hookup to the DT >>

```
#ifdef CONFIG_OF
static const struct of_device_id matrix_keypad_dt_match[] = {
    { .compatible = "gpio-matrix-keypad" },
    { }
};
MODULE_DEVICE_TABLE(of, matrix_keypad_dt_match);
#endif

static struct platform_driver matrix_keypad_driver = {
    .probe      = matrix_keypad_probe,
    .remove     = matrix_keypad_remove,
    .driver     = {
        .name   = "matrix-keypad",
        .pm     = &matrix_keypad_pm_ops,
        .of_match_table = of_match_ptr(matrix_keypad_dt_match),
    },
};
module_platform_driver(matrix_keypad_driver);
```

<< the actual DT parsing function (called from the probe method) >>

```
...
#ifdef CONFIG_OF
static struct matrix_keypad_platform_data *
```

```

matrix_keypad_parse_dt(struct device *dev)
{
    struct matrix_keypad_platform_data *pdata;
    struct device_node *np = dev->of_node;
    unsigned int *gpios;
    int ret, i, nrow, ncol;

    if (!np) {
        dev_err(dev, "device lacks DT data\n");
        return ERR_PTR(-ENODEV);
    }

    pdata = devm_kzalloc(dev, sizeof(*pdata), GFP_KERNEL);
    if (!pdata) {
        dev_err(dev, "could not allocate memory for platform data\n");
        return ERR_PTR(-ENOMEM);
    }

    pdata->num_row_gpios = nrow = of_gpio_named_count(np, "row-gpios");
    pdata->num_col_gpios = ncol = of_gpio_named_count(np, "col-gpios");
    if (nrow <= 0 || ncol <= 0) {
        dev_err(dev, "number of keypad rows/columns not specified\n");
        return ERR_PTR(-EINVAL);
    }

    if (of_get_property(np, "linux,no-autorepeat", NULL))
        pdata->no_autorepeat = true;

    pdata->wakeup = of_property_read_bool(np, "wakeup-source") ||
        of_property_read_bool(np, "linux,wakeup"); /* legacy */

    if (of_get_property(np, "gpio-activelow", NULL))
        pdata->active_low = true;

    pdata->drive_inactive_cols =
        of_property_read_bool(np, "drive-inactive-cols");

    of_property_read_u32(np, "debounce-delay-ms", &pdata->debounce_ms);
    of_property_read_u32(np, "col-scan-delay-us",
        &pdata->col_scan_delay_us);
    ...
}

```

[From the Raspberry Pi documentation:](#)

...

The loader will skip over missing overlays and bad parameters, but if there are serious errors, such as a missing or corrupt base DTB or a failed overlay merge, then the loader will fall back to a non-DT boot. If this happens, or if your settings don't behave as you expect, it is worth checking for warnings or errors from the loader:

```
sudo vcdbg log msg
```

Extra debugging can be enabled by adding `dtdebug=1` to `config.txt` .

...

Small demo on the Raspberry Pi 3B+ device

dtdemo_platdrv : the platform driver.

Obtain the original Raspberry Pi DTS file:

(One way): generate the DTS by doing (on the device; BTW, here we're running the Raspberry Pi 3B+ on a 64-bit Linux kernel and rootfs, [courtesy Ubuntu](#)):

```
dtc -I fs -O dts /proc/device-tree/ > ~/aarch64_raspi2.dts
```

Edit the DTS file:

```
[ ... ]
soc {
    dma-ranges = <0xc0000000 0x0 0x3f000000>;
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    compatible = "simple-bus";
    ranges = <0x7e000000 0x3f000000 0x10000000 0x40000000 0x40000000
0x1000>;
    phandle = <0x31>;

    /* our demo pseudo h/w chip
    From DTSpec: https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3-rc2/devicetree-specification-v0.3-rc2.pdf
    "This list of strings should be used by a client program for device driver
    selection. The property value consists of a concatenated list of null terminated
    strings, from most specific to most general. They allow a device to express its
    compatibility with a family of similar devices, potentially allowing a single
    device driver to match against several devices. " */
    dtdemo_chip {
        compatible = "dtdemo,dtdemo_platdev";
        aproperty = "my prop 1";
    };

    local_intc@40000000 {
[ ... ]
```

Add to the DT (under /boot/firmware), build it:

Generate the new DTB now:

```
rpi # dtc <dtb-name>.dts > <dtb-name>.dtb
<stdout>: Warning (unit_address_vs_reg): Node /soc has a reg or ranges property, but no
unit name
<stdout>: Warning (unit_address_vs_reg): Node /soc/gpiomem has a reg or ranges property,
...
```



```
<stdout>: Warning (simple_bus_reg): Node /soc/dtdemo missing or empty reg/ranges property
...
rpi #
```

Keep the dtdemo_platdrv kernel module built and install it (sudo make install).

Reboot and check:

```
rpi # dtc -I fs /proc/device-tree |grep -C3 "dtdemo"
<stdout>: Warning (unit_address_vs_reg): Node /soc has a reg or ranges property, but no
unit name
...
                reg = <0x7e215080 0x40>;
            };

            dtdemo_chip {
                compatible = "dtdemo,dtdemo_platdev";
                aproperty = "my prop 1";
            };

            aux@0x7e215000 {
rpi #
```

The KEY point:

the string in the compatible property in the DT and in the driver source's of_device_id[] array must be IDENTICAL. Really!!! Even a mismatched space will cause the match to fail.

After rebooting with the new DTB we have a new node here:
/sys/bus/platform/devices/soc:dtdemo_chip

```
rpi64 ~ $ ls -l /sys/bus/platform/devices/soc\:dtdemo_chip/
total 0
lrwxrwxrwx 1 root root    0 Jul 23 07:25 driver ->
../../../../bus/platform/drivers/dtdemo_platdev/
-rw-r--r-- 1 root root 4096 Jul 23 07:25 driver_override
-r--r--r-- 1 root root 4096 Jul 23 07:25 modalias
lrwxrwxrwx 1 root root    0 Jul 23 07:25 of_node ->
../../../../firmware/devicetree/base/soc/dtdemo_chip/
drwxr-xr-x 2 root root    0 Jul 23 07:25 power/
lrwxrwxrwx 1 root root    0 Jan  1  1970 subsystem -> ../../../../../../bus/platform/
-rw-r--r-- 1 root root 4096 Jan  1  1970 uevent
rpi64 ~ $ cat /sys/bus/platform/devices/soc\:dtdemo_chip/uevent
DRIVER=dtdemo_platdev
OF_NAME=dtdemo_chip
OF_FULLNAME=/soc/dtdemo_chip
OF_COMPATIBLE_0=dtdemo,dtdemo_platdev
OF_COMPATIBLE_N=1
MODALIAS=of:Ndtdemo_chipT<NULL>Cdtdemo,dtdemo_platdev
rpi64 ~ $
```

Notice how, because the tweaked DT has the new (pseudo) platform device *dtdemo_chip*, the device got auto-instantiated at boot, and because of it's *compatible* property, **the kernel auto-loaded the *dtdemo_platdrv* kernel module** (as it perfectly matched the compatible property) and thus got bound:

```

rpi64 ~ $ dmesg |grep "dtdemo"
[ 11.656895] dtdemo_platdrv: loading out-of-tree module taints kernel.
[ 11.657072] dtdemo_platdrv: module verification failed: signature and/or required
key missing - tainting kernel
[ 11.658926] dtdemo_platdrv: inserted
[ 11.659209] dtdemo_platdrv: platform driver probe enter
[ 11.659218] dtdemo_platdrv: DT property 'aproperty' = my prop 1 (len=10)
rpi64 ~ $ lsmod |grep dtdemo
dtdemo_platdrv      16384  0
rpi64 ~ $

```

How does the kernel parse the DT properties once the platform driver is registered (typically via the `platform_driver_register()` API)?

Lets use `ftrace` to find out! To make it easier, we use the super `trace-cmd(1)` front-end to `ftrace`.

Get `trace-cmd`:

git clone <https://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git>

Build & install:

make && sudo make install

Use it: wanted to capture the work of `platform_driver_register()` within the `insmod`:

First, lets make it easier by *ourselves* emitting a message into the `ftrace` log buffer (via the `trace_printk()`), around the area of interest; this way, we can just search the (typically huge) `trace-cmd` report for this string!

```

[...]
    trace_printk("@@@@ %s: MARKER 1: platform_driver_register() begin\n", DRVNAME);
    ret_val = platform_driver_register(&my_platform_driver);
    trace_printk("@@@@ %s: MARKER 2: platform_driver_register() done\n", DRVNAME);

```

```

trace-cmd record -p function_graph -F taskset 01 insmod ./dtdemo_platdrv.ko
trace-cmd report -I -S -l > ftrc_rep.txt

```

Notice how we use `taskset(1)` to ensure that the process we're interested in `ftrace`-ing (`insmod`) runs on exactly one CPU.

NOTE!

Actually, doing a `sudo make install` of our platform driver ensures it's discoverable at boot; it does indeed get auto-loaded at boot because our (pseudo) platform device is instantiated by the kernel – because our tweaked DT has a new (pseudo) platform device ('dtdemo')!

\$ vi ftrc_rep.txt *<< it's 3.6 MB ; below, we have prevented the output from wrapping >>*

```

[...]
insmod-1851  1.... 441.946436: funcgraph_exit:      ! 280.834 us |      }
insmod-1851  1.... 441.946443: bprint:          init_module: @@@@ dtdemo_platdev: MARKER 1: platform_driver_register()
begin

```

```

insmod-1851  1....  441.946445: funcgraph_entry:      |
insmod-1851  1....  441.946447: funcgraph_entry:      |
...
insmod-1851  1....  441.947200: funcgraph_entry:      |
insmod-1851  1....  441.947202: funcgraph_entry:      |
insmod-1851  1....  441.947203: funcgraph_entry:      |
insmod-1851  1....  441.947205: funcgraph_entry:      |
insmod-1851  1....  441.947206: funcgraph_entry:      | 0.677 us |
insmod-1851  1d..1  441.947209: funcgraph_entry:      |
insmod-1851  1d..1  441.947210: funcgraph_entry:      |
__of_device_is_compatible() {
insmod-1851  1d..1  441.947212: funcgraph_entry:      | 3.021 us |
insmod-1851  1d..1  441.947217: funcgraph_entry:      | 0.521 us |
insmod-1851  1d..1  441.947220: funcgraph_entry:      | 0.208 us |
insmod-1851  1d..1  441.947222: funcgraph_exit:       | + 11.563 us |
insmod-1851  1d..1  441.947223: funcgraph_exit:       | + 14.219 us |
insmod-1851  1d..1  441.947224: funcgraph_entry:      | 0.261 us |
__raw_spin_unlock_irqrestore();
insmod-1851  1....  441.947226: funcgraph_exit:       | + 20.625 us |
insmod-1851  1....  441.947227: funcgraph_exit:       | + 22.969 us |
insmod-1851  1....  441.947229: funcgraph_exit:       | + 26.667 us |
insmod-1851  1....  441.947230: funcgraph_exit:       | + 29.167 us |
...

    << code of of_*(()) runs several times ... >>

insmod-1851  1....  441.947700: funcgraph_entry:      |
insmod-1851  1....  441.947701: funcgraph_entry:      |
of_parse_phandle_with_args() {
insmod-1851  1....  441.947703: funcgraph_entry:      |
__of_parse_phandle_with_args() {
insmod-1851  1....  441.947704: funcgraph_entry:      |
of_phandle_iterator_init() {
insmod-1851  1....  441.947705: funcgraph_entry:      |
insmod-1851  1....  441.947706: funcgraph_entry:      | 0.261 us |
__raw_spin_lock_irqsave();
...

insmod-1851  1....  441.947764: funcgraph_exit:       | ! 316.615 us |
insmod-1851  1....  441.947766: funcgraph_entry:      |
...

insmod-1851  1....  441.947952: funcgraph_entry:      |
insmod-1851  1....  441.947953: funcgraph_entry:      |
insmod-1851  1....  441.947954: funcgraph_entry:      |
{
insmod-1851  1....  441.947956: funcgraph_entry:      |
{
insmod-1851  1....  441.947956: funcgraph_entry:      |
...

insmod-1851  1....  441.948017: funcgraph_entry:      |
    << then the printk() we emitted runs ... >>
insmod-1851  1....  441.948019: funcgraph_entry:      |
insmod-1851  1....  441.948020: funcgraph_entry:      |
...

```

```

insmod-1851  1.... 441.948088: funcgraph_entry:      |
insmod-1851  1.... 441.948089: funcgraph_entry:      |
insmod-1851  1.... 441.948089: funcgraph_entry:      | 0.260 us |
_raw_spin_lock_irqsave();
insmod-1851  1d..1 441.948091: funcgraph_entry:      | 0.521 us |
insmod-1851  1d..1 441.948093: funcgraph_entry:      | 0.260 us |
_raw_spin_unlock_irqrestore();
insmod-1851  1.... 441.948095: funcgraph_exit:       | 5.833 us |
insmod-1851  1.... 441.948096: funcgraph_exit:       | 7.656 us |
insmod-1851  1.... 441.948097: funcgraph_entry:      |
...
insmod-12410 0.... 6164.582970: funcgraph_entry:      |
insmod-12410 0.... 6164.582971: funcgraph_entry:      |
{
insmod-12410 0.... 6164.582973: funcgraph_entry:      |
vprintk_default() {
insmod-12410 0.... 6164.582974: funcgraph_entry:      |
vprintk_emit() {
insmod-12410 0d... 6164.582975: funcgraph_entry:      | 0.261 us |
__printk_safe_enter();
...

```

Can see the '*of_**' - Open Firmware – code running within the *platform_driver_register()* code path.

- **Q. How does the DT (Device Tree) get passed to the Linux kernel?**

A. This is usually done via the bootloader; the bootloader loads both the kernel image file and the DTB (Device Tree Blob) into memory (at pre-defined addresses), from where the OS can pick up the information.

So, how is the DTB generated? Short answer- the *.dts* (Device Tree Source) files are “compiled” into the binary DTB format using a userspace tool called '*dtc*' (Device Tree Compiler).

```

plat-xyz.dts → plat-xyz.dtb → Linux kernel
              dtc              u-boot

```

<<

One can see in the ARM kernel start-up code, the kernel expects the *r2* register to be populated with *either* the ATAGS physical address -or- the DTB physical address.

File : ***arch/arm/kernel/head.S***

```

...
58 /*
59  * Kernel startup entry point.
60  * -----
61  *
62  * This is normally called from the decompressor code. The requirements
63  * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
64  * r1 = machine nr, r2 = atags or dtb pointer.
>>

```

In the kernel configuration menu (Kconfig) system, we see:

Under “**Boot Options**”:

```
.config - Linux/arm 4.16.0 Kernel Configuration
-> Boot options

                                Boot options
Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M>
modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

-- Flattened Device Tree support
[*] Support for the traditional ATAGS boot data passing (NEW)
[ ] Provide old way to pass kernel parameters (NEW)
(0) Compressed ROM boot loader base address (NEW)
(0) Compressed ROM boot loader BSS address (NEW)
[ ] Use appended device tree blob to zImage (EXPERIMENTAL) (NEW)
() Default kernel command string (NEW)
[*] Kexec system call (EXPERIMENTAL)
[*] Export atags in procfs (NEW)
[*] Build kdump crash kernel (EXPERIMENTAL)
-- Auto calculation of the decompressed kernel image address
[*] UEFI runtime support
[*] Enable support for SMBIOS (DMI) tables

<Select>  < Exit >  < Help >  < Save >  < Load >
```

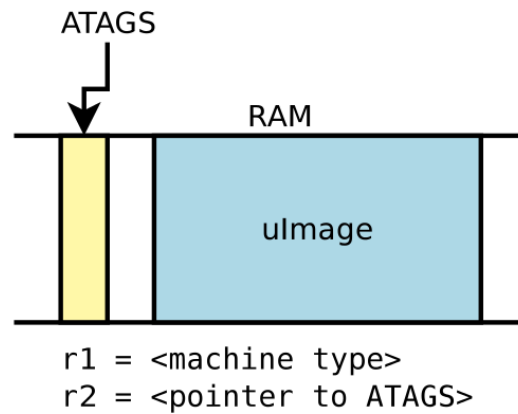
Check out the first few menu entries...

<<

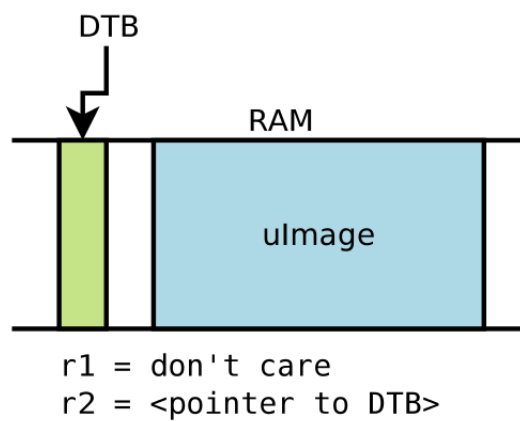
[Source: Device Trees for Dummies, Thomas Petazzoni](#)



User perspective: before the Device Tree



User perspective: booting with a Device Tree





From source to binary

- ▶ On ARM, all **Device Tree Source** files (DTS) are for now located in `arch/arm/boot/dts`
 - ▶ `.dts` files for board-level definitions
 - ▶ `.dtsi` files for included files, generally containing SoC-level definitions
- ▶ A tool, the **Device Tree Compiler** compiles the source into a binary form.
 - ▶ Source code located in `scripts/dtc`
- ▶ The **Device Tree Blob** is produced by the compiler, and is the binary that gets loaded by the bootloader and parsed by the kernel at boot time.
- ▶ `arch/arm/boot/dts/Makefile` lists which DTBs should be generated at build time.

```
dtb=$(CONFIG_ARCH_MVEBU) += armada-370-db.dtb \
    armada-370-mirabox.dtb \
    ...
```

>>

More information available here:

[“Bootloaders in Embedded Linux Systems”, Chris Hallinan](#)

- Q. How to build the DTB?

A. In Linux source tree root (for ARM):

make dtbs - Build device tree blobs for enabled boards

Example (on kernel ver 3.10.6, setup for the ARM Cortex-A9 Versatile Express board):

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- vexpress_defconfig
#
# configuration written to .config
#
$
```

Build the Device Tree Blob files for the Vexpress platform

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- dtbs
scripts/kconfig/conf --silentoldconfig Kconfig
CC      scripts/mod/empty.o
```

```

HOSTCC scripts/mod/mk_elfconfig
MKELF scripts/mod/elfconfig.h
CC scripts/mod/devicetable-offsets.s
GEN scripts/mod/devicetable-offsets.h
HOSTCC scripts/mod/file2alias.o
HOSTCC scripts/mod/modpost.o
HOSTCC scripts/mod/sumversion.o
HOSTLD scripts/mod/modpost
HOSTCC scripts/kallsyms
HOSTCC scripts/pnmtologo
HOSTCC scripts/conmakehash
HOSTCC scripts/bin2c
HOSTCC scripts/sortextable
DTC arch/arm/boot/dts/vexpress-v2p-ca5s.dtb
DTC arch/arm/boot/dts/vexpress-v2p-ca9.dtb
DTC arch/arm/boot/dts/vexpress-v2p-ca15-tc1.dtb
DTC arch/arm/boot/dts/vexpress-v2p-ca15_a7.dtb
$ ls -l arch/arm/boot/dts/vexpress-v2*.dt[sb]
-rw-rw-r-- 1 kaiwan kaiwan 13K Aug 13 16:38 arch/arm/boot/dts/vexpress-v2p-ca15_a7.dtb
[...]
-rw-rw-r-- 1 kaiwan kaiwan 12K Aug 13 16:38 arch/arm/boot/dts/vexpress-v2p-ca9.dtb
-rw-rw-r-- 1 kaiwan kaiwan 7.2K Aug 12 07:19 arch/arm/boot/dts/vexpress-v2p-ca9.dts
$
$ file arch/arm/boot/dts/vexpress-v2*.dt[sb]
arch/arm/boot/dts/vexpress-v2p-ca15_a7.dtb: data
arch/arm/boot/dts/vexpress-v2p-ca15_a7.dts: ASCII text
arch/arm/boot/dts/vexpress-v2p-ca15-tc1.dtb: data
arch/arm/boot/dts/vexpress-v2p-ca15-tc1.dts: ASCII text
arch/arm/boot/dts/vexpress-v2p-ca5s.dtb: data
arch/arm/boot/dts/vexpress-v2p-ca5s.dts: ASCII text
arch/arm/boot/dts/vexpress-v2p-ca9.dtb: data
arch/arm/boot/dts/vexpress-v2p-ca9.dts: ASCII text
$

```


Eg. 1

<< Lets try booting a Qemu-emulated ARM-Versatile-Express CA-9 platform, supplying the DTB on the bootloader command-line. As we're emulating a system with Qemu, it itself acts as the bootloader (else typically passed via U-Boot) >>

```
$ qemu-system-arm -m 256 \
-M vexpress-a9 \ << emulated machine: Versatile Express Cortex-A9 >>
-kernel <...>/zImage \
-drive file=<...>/rfs.img,if=sd,format=raw \
-append "console=ttyAMA0 rootfstype=ext4 root=/dev/mmcblk0 init=/sbin/init " \
-nographic \
-dtb <...>/vexpress-v2p-ca9.dtb << pass the -dtb path-to-the-dtb file >>

...
Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpuset
Linux version 3.18.22 (kaiwan@kaiwan-T460) (gcc version 4.8.3 20140320 (prerelease) (Sourcery
CodeBench Lite 2014.05-29) ) #3 SMP Fri Mar 24 06:47:34 IST 2017
CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7), cr=10c5387d

[...]

Machine model: V2P-CA9
Memory policy: Data cache writeback
CPU: All CPU(s) started in SVC mode.
PERCPU: Embedded 10 pages/cpu @cfdc6000 s11200 r8192 d21568 u40960
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 65024
Kernel command line: console=ttyAMA0 rootfstype=ext4 root=/dev/mmcblk0 init=/sbin/init
[...]
L2C: device tree omits to specify unified cache
L2C: DT/platform modifies aux control register: 0x02020000 -> 0x02420000

[...]
```

Eg. 2

<< Lets try booting a Yocto built ARM system: again, it's a Qemu-emulated ARM-Versatile-PB ARMv5 platform, supplying the DTB on the bootloader command-line. As we're emulating a system with Qemu, it itself acts as the bootloader (else typically passed via U-Boot) >>

...in the Yocto-build folder...

```
$ runqemu qemuarm nographic
runqemu - INFO - Assuming MACHINE = qemuarm
runqemu - INFO - Running MACHINE=qemuarm bitbake -e...
runqemu - INFO - MACHINE: qemuarm
runqemu - INFO - DEPLOY_DIR_IMAGE:
/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/deploy/images/qemuarm
runqemu - INFO - Running ls -t
/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/deploy/images/qemuarm/
*.qemuboot.conf...
runqemu - INFO - CONFFILE:
/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/deploy/images/qemuarm/core-
image-minimal-qemuarm-20161212064558.qemuboot.conf
runqemu - INFO - Continuing with the following parameters:

KERNEL: [/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/deploy/images/
qemuarm/zImage]
DTB: [/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/deploy/images/qemuarm/
zImage-versatile-pb.dtb]
```

```

MACHINE: [qemuarm]
FSTYPE: [ext4]
ROOTFS: [/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/ deploy/images/
qemuarm/core-image-minimal-qemuarm-20161212064558.rootfs.ext4]
CONFFILE: [/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/ deploy/images/
qemuarm/core-image-minimal-qemuarm-20161212064558.qemuboot.conf]

runqemu - INFO - Running /sbin/ip link...
runqemu - INFO - Setting up tap interface under sudo
runqemu - INFO - Acquiring lockfile /tmp/qemu-tap-locks/tap0.lock...
runqemu - INFO - Created tap: tap0
runqemu - INFO - Running ldd /home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/
sysroots/x86_64-linux/usr/bin/qemu-system-arm...
runqemu - INFO - Running
/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/sysroots/x86_64-linux/usr/
bin/qemu-system-arm -device virtio-net-pci,netdev=net0,mac=52:54:00:12:34:02 -
netdev tap,id=net0,ifname=tap0,script=no,downscript=no -nographic -machine
versatilepb -m 256 -drive
file=/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/ deploy/images/qemuarm/
core-image-minimal-qemuarm-20161212064558.rootfs.ext4,if=virtio,format=raw -
show-cursor -usb -usbdevice tablet -device virtio-rng-pci -kernel
/home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/ deploy/images/qemuarm/zImage
-append 'root=/dev/vda rw highres=off console=ttyS0 mem=256M
ip=192.168.7.2::192.168.7.1:255.255.255.0 console=ttyAMA0,115200 console=tty' -
dtb /home/yocto/yocto_poky/poky-morty-16.0.0/build/tmp/ deploy/images/qemuarm/
zImage-versatile-pb.dtb
vpb_sic_write: Bad register offset 0x2c
[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Linux version 4.8.3-yocto-standard (yocto@yocto-VirtualBox) (gcc
version 6.2.0 (GCC) ) #1 PREEMPT Mon Dec 12 14:25:33 IST 2016
[ 0.000000] CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177
[ 0.000000] CPU: VIVT data cache, VIVT instruction cache
[ 0.000000] OF: fdt:Machine model: ARM Versatile PB
[ 0.000000] Memory policy: Data cache writeback

[...]
```



Exploring the DT on the target

- In `/sys/firmware/devicetree/base`, there is a directory/file representation of the Device Tree contents

```
# ls -l /sys/firmware/devicetree/base/
total 0
-r--r--r--    1 root    root    4 Jan  1 00:00 #address-cells
-r--r--r--    1 root    root    4 Jan  1 00:00 #size-cells
drwxr-xr-x    2 root    root    0 Jan  1 00:00 chosen
drwxr-xr-x    3 root    root    0 Jan  1 00:00 clocks
-r--r--r--    1 root    root   34 Jan  1 00:00 compatible
[...]
-r--r--r--    1 root    root    1 Jan  1 00:00 name
drwxr-xr-x   10 root    root    0 Jan  1 00:00 soc
```

- If `dtc` is available on the target, possible to "unpack" the Device Tree using:

```
dtc -I fs /sys/firmware/devicetree/base
```

<< On a Qemu-emulated Vexpress CA9 ARM board ... >>

```
ARM # ls -l /sys/firmware/
total 0
drwxr-xr-x    3 0        0          0 Jul  6 06:59 devicetree/
-r-----    1 0        0      49384 Jul  6 06:59 fdt
ARM # ls -l /sys/firmware/devicetree/
total 0
drwxr-xr-x   24 0        0          0 Jul  6 06:59 base/
ARM # ls -l /sys/firmware/devicetree/base/
total 0
-r--r--r--    1 0        0          4 Jul  6 06:59 #address-cells
-r--r--r--    1 0        0          4 Jul  6 06:59 #size-cells
drwxr-xr-x    2 0        0          0 Jul  6 06:59 aliases/
-r--r--r--    1 0        0          4 Jul  6 06:59 arm,hbi
-r--r--r--    1 0        0          4 Jul  6 06:59 arm,vexpress,site
drwxr-xr-x    2 0        0          0 Jul  6 06:59 cache-controller@1e00a000/
drwxr-xr-x    2 0        0          0 Jul  6 06:59 chosen/
drwxr-xr-x    4 0        0          0 Jul  6 06:59 clcd@10020000/
-r--r--r--    1 0        0      34 Jul  6 06:59 compatible
drwxr-xr-x    6 0        0          0 Jul  6 06:59 cpus/
drwxr-xr-x   15 0        0          0 Jul  6 06:59 dcc/
drwxr-xr-x    2 0        0          0 Jul  6 06:59 hsb@e0000000/
drwxr-xr-x    2 0        0          0 Jul  6 06:59 interrupt-controller@1e001000/
-r--r--r--    1 0        0          4 Jul  6 06:59 interrupt-parent
drwxr-xr-x    2 0        0          0 Jul  6 06:59 memory-controller@100e0000/
drwxr-xr-x    2 0        0          0 Jul  6 06:59 memory-controller@100e1000/
drwxr-xr-x    2 0        0          0 Jul  6 06:59 memory@60000000/
-r--r--r--    1 0        0          8 Jul  6 06:59 model
-r--r--r--    1 0        0          1 Jul  6 06:59 name
drwxr-xr-x    2 0        0          0 Jul  6 06:59 pmu/
```

```

drwxr-xr-x  2 0      0      0 Jul  6 06:59 scu@1e000000/
drwxr-xr-x  3 0      0      0 Jul  6 06:59 smb@4000000/
drwxr-xr-x  2 0      0      0 Jul  6 06:59 timer@100e4000/
drwxr-xr-x  2 0      0      0 Jul  6 06:59 timer@1e000600/
drwxr-xr-x  2 0      0      0 Jul  6 06:59 virtio_mmio@10013000/
drwxr-xr-x  2 0      0      0 Jul  6 06:59 virtio_mmio@10013200/
drwxr-xr-x  2 0      0      0 Jul  6 06:59 virtio_mmio@10013400/
drwxr-xr-x  2 0      0      0 Jul  6 06:59 virtio_mmio@10013600/
drwxr-xr-x  2 0      0      0 Jul  6 06:59 watchdog@100e5000/
drwxr-xr-x  2 0      0      0 Jul  6 06:59 watchdog@1e000620/
ARM #

```

OPTIONAL / FYI

Pinmux – pin multiplexing; pl refer:

[Linux device driver development: The pin control subsystem, John Madieu](#)

Kernel doc: [PINCTRL \(PIN CONTROL\) subsystem](#)

Src: [Update](#) (for kernel ver 3.19):

...

1.7. Device Tree overlays

The Device Tree is a data structure for describing hardware that is passed to the operating system at boot time, rather than hard coding every detail of a device into an operating system. It is used most often in architectures like powerpc and ARM. The Device Tree is designed with static systems in mind, and has troubles adapting to custom expansion busses commonly found on consumer development boards like the [BeagleBone](#) or Raspberry Pi.

This release introduces Device Tree overlay support. **Overlays are a method to dynamically modify part of the kernel's device tree and make changes to properties in a existing tree.** This makes easier to support devices such as the [BeagleBone](#) or Raspberry Pi.

Recommended LWN article: [Device tree overlays](#)

Code: [commit](#)

Linux kernel documentation: [Device Tree Overlay Notes](#)

[Device Tree for the Raspberry Pi](#)

[Information on using Device Tree for the Raspberry Pi](#)

[btw, HAT=Hardware Attached on Top, a new feature of the R Pi B+]

Additional Resources:

- https://elinux.org/Device_Tree_Reference
- https://elinux.org/Device_Tree_Linux – Linux specific info regarding the DT



User perspective: compatibility mode for DT booting

- ▶ Some bootloaders have no specific support for the Device Tree, or the version used on a particular device is too old to have this support.
- ▶ To ease the transition, a *compatibility* mechanism was added: `CONFIG_ARM_APPENDED_DTB`.
 - ▶ It tells the kernel to look for a DTB right *after* the kernel image.
 - ▶ There is no built-in Makefile rule to produce such kernel, so one must manually do:

```
cat arch/arm/boot/zImage arch/arm/boot/dts/myboard.dtb > my-zImage
mkimage ... -d my-zImage my-uImage
```

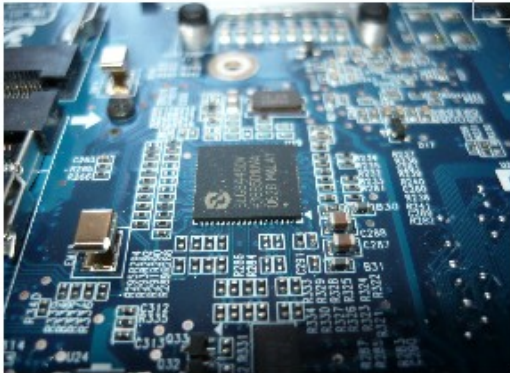
- ▶ In addition, the additional option `CONFIG_ARM_ATAG_DTB_COMPAT` tells the kernel to read the *ATAGS* information from the bootloader, and update the DT using them.

Tips

- The *compatible* string is used to bind a device with a driver (of course, the string in the DT and driver's *compatible* property must be a perfect match)
 - If manual parsing of the DT is required in the driver, do so in the *probe* method
 - can use the `of_property_read_[bool|u|s[8|16|32]|string[_helper]]()`, `platform_get_resource()`, etc APIs
 - DT basic syntax
 - **6. Devicetree Source (DTS) Format (version 1)** – nicely shown and explained
 - NXP 'App Note – DT Intro' <https://www.nxp.com/docs/en/application-note/AN5125.pdf>
 - "The DTC can also be used to reverse compile DTBs and make them human-readable again:

`dtc -I dtb -O dts bsc9131rdb.dtb > bsc9131rdb_output.dts`
or
`dtc -I fs -O dts /proc/device-tree/`
 - "Device tree bindings describe the syntax used to describe specific types and classes of devices. The **compatible property** of a device node describes the specific binding, or bindings, to which the node complies. Device tree bindings recognized by the kernel are documented in *Documentation/devicetree/bindings*."
 - Excellent article: [Platform devices and device trees, LWN, Jon Corbet](#)
 - Blog article: [Device Tree Tutorial \(ARM\), Saurabh Singh Sengar](#)
-

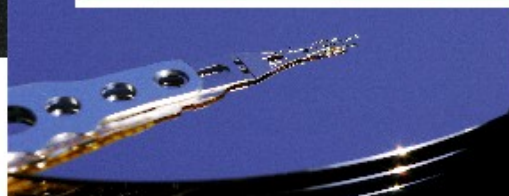
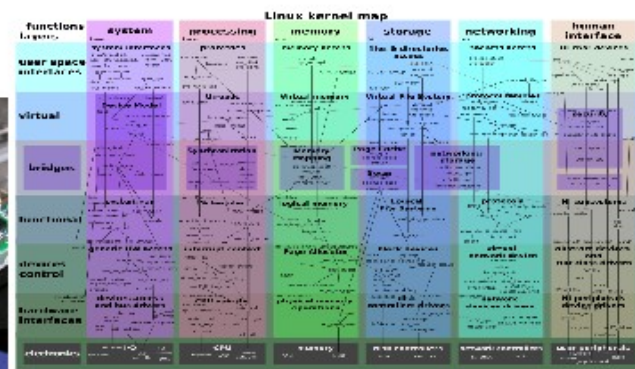
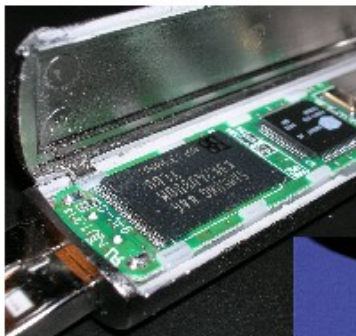
Linux Operating System Specialized



The highest quality Training on:

Linux Fundamentals, CLI and Scripting
Linux Systems Programming
Linux Kernel Internals
Linux Device Drivers
Embedded Linux
Linux Debugging Techniques
New! Linux OS for Technical Managers

Please do visit our website for details:
<http://kaiwantech.in>



<http://kaiwantech.in>