# Device tree

In [computing](https://en.wikipedia.org/wiki/Computing), a **device tree** (also written **devicetree**) is a [data structure](https://en.wikipedia.org/wiki/Data_structure) describing the hardware components of a particular computer so that the [operating system](https://en.wikipedia.org/wiki/Operating_system)'s [kernel](https://en.wikipedia.org/wiki/Kernel_(operating_system)) can use and manage those components, including the [CPU](https://en.wikipedia.org/wiki/Central_processing_unit) or CPUs, the [memory](https://en.wikipedia.org/wiki/Computer_memory), the [buses](https://en.wikipedia.org/wiki/Bus_(computing)) and the [peripherals](https://en.wikipedia.org/wiki/Peripheral).

[Personal computers](https://en.wikipedia.org/wiki/Personal_computer) with the [x86](https://en.wikipedia.org/wiki/X86) architecture generally do not use device trees, relying instead on various auto configuration protocols to discover hardware. Systems which use device trees usually pass a static device tree (perhaps stored in [ROM](https://en.wikipedia.org/wiki/Read-only_memory)) to the operating system, but can also generate a device tree in the early stages of [booting](https://en.wikipedia.org/wiki/Booting). As an example, [Das U-Boot](https://en.wikipedia.org/wiki/Das_U-Boot) and [kexec](https://en.wikipedia.org/wiki/Kexec) can pass a device tree when launching a new operating system. On systems with a boot loader that does not support device trees, a static device tree may be installed along with the operating system; the [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel) supports this approach.

## Device Tree formats

A device tree can hold any kind of data as internally it is a [tree](https://en.wikipedia.org/wiki/Tree_structure) of named nodes and [properties](https://en.wikipedia.org/wiki/Property_(programming)). Nodes contain properties and child nodes, while properties are [name–value pairs](https://en.wikipedia.org/wiki/Attribute–value_pair).

Device trees have both a [binary format](https://en.wikipedia.org/wiki/Binary_format) for operating systems to use and a textual format for convenient editing and management.[[1]](https://en.wikipedia.org/wiki/Device_tree" \l "cite_note-dtv02-1)

# Device Tree Tutorial (ARM)

## Overview

The linux kernel requires the entire description of the hardware, like which board it is booting(machine type), which all devices it is using there addresses(device/bus addresses), there interrupts numbers(irq), mfp pins configuration(pin muxing/gpios)  also some board level information like memory size, kernel command line etc etc …

Before device tree, all these information use to be set in a huge cluster of board files. And, Information like command line, memory size etc use to be passed by bootloaders as part of ATAGS through register R2(ARM). Machine type use to be set separately in register R1(ARM).  
At this time each kernel compilation use to be for only one specific chip an a specific board.

So there was a long pending wish to compile the kernel for all ARM processors, and let the kernel somehow detect its hardware and apply the right drivers as needed just like your PC.  
But how? On a PC, the initial registers are hardcoded, and the rest of the information is supplied by the BIOS. But ARM processors don’t have a BIOS.  
The solution chosen was device tree, also referred to as Open Firmware (abbreviated OF) or Flattened Device Tree (FDT). This is essentially a data structure in byte code format which contains information that is helpful to the kernel when booting up.

The bootloader now loads two binaries: the kernel image and the DTB.  
DTB is the device tree blob. The bootloader passes the DTB address through R2 instead of ATAGS and R1 register is not required now.

For a one line bookish definition “A device tree is a tree data structure with nodes that describe the physical devices in a system”

Currently device tree is supported by ARM, x86, Microblaze, PowerPC, and Sparc architectures.

## I. Device Tree Compilation

Device tree compiler and its source code  located at scripts/dtc/.  
On ARM all device tree source are located at /arch/arm/boot/dts/.  
The Device Tree Blob(.dtb) is produced by the compiler, and it is the binary that gets loaded by the bootloader and parsed by the kernel at boot time.

$ scripts/dtc/dtc -I dts -O dtb -o /path/my\_tree.dtb /arch/arm/boot/dts/my\_tree.dts

This will result my\_tree.dtb

For creating the dts from dtb

$ scripts/dtc/dtc -I dtb -O dts -o /path/my\_tree.dts /path/my\_tree.dtb

This will result my\_tree.dts

## II. Device Tree Basics

Each module in device tree is defined by a node and all its properties are defined under that node. Depending on the driver it can have child nodes or parent node.  
For example a device connected by i2c bus, will have i2c as its parent node, and that device will be one of the child node of i2c node, i2c may have apd bus as its parent and so on. All leads up to root node, which is parent of all. (Don’t worry an example after this section will make it more clear.)  
Under the root of the Device Tree, one typically finds the following most common top-level nodes:

* cpus: its each sub-nodes describing each CPU in the system.
* memory : defines location and size of the RAM.
* chosen : defines parameters chosen or defined by the system firmware at boot time. In practice, one of its usage is to pass the kernel command line.
* aliases: shortcuts to certain nodes.
* One or more nodes defining the buses in the SoC
* One or mode nodes defining on-board devices

## III. Device Tree Structure example

Here will take the example of a dummy dts code for explanation

 #include "pxa910.dtsi"

/ {

compatible = "mrvl,pxa910-dkb", "mrvl,pxa910";

chosen {

bootargs = "<boot args here>";

};

memory {

reg = <0x00000000 0x10000000>;

};

soc {

apb@d4000000 {

uart1: uart@d4017000 {

status = "okay";

};

twsi1: i2c@d4011000 {

#address-cells = <1>

#size-cells = <0>

status = "okay";

pmic: 88pm860x@34 {

compatible = "marvell,88pm860x";

reg = <0x34>;

interrupts = <4>;

interrupt-parent = <&intc>;

interrupt-controller;

#interrupt-cells = <1>;

Figure 1

Each module is defined in one curly bracket area under one node, any sub modules can be defined further inside.

Explaning the above tree starting from the first line :

#include : including any headed file, just like any C file  
.dtsi : extended dts file, single dts can have any number of dtsi, but couldn’t include other dts file  
/: root node, device tree structure starts here

## IV. Properties

There are data define in dts as form of property which are read by the kernel code, lets read about some of the major properties

### Compatible

The top-level compatible property typically defines a compatible string for the board. Priority always given with the most-specific first, to least-specific last. It used to match with the dt\_compat field of the DT\_MACHINE structure.  
Inside a driver or bus node , it is the most crucial one, as it is the link between the hardware and its driver.Each node belongs to one compatible string and based on compatible string only kernel matches the device driver with its data in device tree node.  
The connection between a kernel driver and the “compatible” entries it should be attached to, is made by a code segment as follows in the driver’s source code:

static struct of\_device\_id dummy\_of\_match[] = {

{ .compatible = "marvell,88pm860x", },

{}

};

MODULE\_DEVICE\_TABLE(of, dummy\_of\_match);

 The above code in driver matches it to the pmic node shown in device tree structure shown in figure 1.

### reg

defines the address for that node/device

### #address-cells

property indicate how many cells (i.e 32 bits values) are needed to form the base address part in the reg property

### #size-cells

the size part of the reg property

### interrupt-controller

is a boolean property that indicates that the current node is an interrupt controller

### #interrupt-cells

indicates the number of cells in the interrupts property for the interrupts managed by the selected interrupt controller

### interrupt-parent

is a phandle that points to the interrupt controller for the current node. There is generally a top-level interrupt-parent definition for the main interrupt controller.

### The label and node name

First, the label (”pmic”) and entry’s name (”88pm860x@34″). The label could have been omitted altogether, and the entry’s node name should stick to this format (some-name@address). This tells the kernel that this driver name 88pm860x and connected to its parent bus(i2c in this case) with the adress 34 (i2c slave address here). PMIC is the label which could be use as a phandle to refer this node inside dts.

## V. Getting the resources from DTS

Below are the few major APIs in current kernel (4.3) for reading the various properties from DTS.

of\_address\_to\_resource: Reads the memory address of device defined by res property

irq\_of\_parse\_and\_map: Attach the interrupt handler, provided by the properties interrupt and interrupt-parent

of\_find\_property(np, propname, NULL): To find if property named in argument2 is present or not.

of\_property\_read\_bool: To read a bool property named in argument 2, as it is a bool property it just like searching if that property present or not. Returns true or false

of\_get\_property: For reading any property named in argument 2

of\_property\_read\_u32: To read a 32 bit property, populate into 3rd argument. Doesn’t set anything to 3rd argument in case of error.

of\_property\_read\_string: To read string property

of\_match\_device: Sanity check for device that device is matching with the node, highly optional, I don’t see much use of it.

let me know if you have any doubts related to device tree in comment section below or an personal email to me.