Linux Devicetree Documentation

The kernel development community

Jan 15, 2023

CONTENTS

CHAPTER

ONE

KERNEL DEVICETREE USAGE

1.1 Linux and the Devicetree

The Linux usage model for device tree data

Author Grant Likely <grant.likely@secretlab.ca>

This article describes how Linux uses the device tree. An overview of the device tree data format can be found on the device tree usage page at devicetree.org[1].

The "Open Firmware Device Tree", or simply Devicetree (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.

Structurally, the DT is a tree, or acyclic graph with named nodes, and nodes may have an arbitrary number of named properties encapsulating arbitrary data. A mechanism also exists to create arbitrary links from one node to another outside of the natural tree structure.

Conceptually, a common set of usage conventions, called 'bindings', is defined for how data should appear in the tree to describe typical hardware characteristics including data busses, interrupt lines, GPIO connections, and peripheral devices.

As much as possible, hardware is described using existing bindings to maximize use of existing support code, but since property and node names are simply text strings, it is easy to extend existing bindings or create new ones by defining new nodes and properties. Be wary, however, of creating a new binding without first doing some homework about what already exists. There are currently two different, incompatible, bindings for i2c busses that came about because the new binding was created without first investigating how i2c devices were already being enumerated in existing systems.

1.1.1 1. History

The DT was originally created by Open Firmware as part of the communication method for passing data from Open Firmware to a client program (like to an operating system). 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).

Since Open Firmware is commonly used on PowerPC and SPARC platforms, the Linux support for those architectures has for a long time used the Device Tree.

[1] https://www.devicetree.org/specifications/

In 2005, when PowerPC Linux began a major cleanup and to merge 32-bit and 64-bit support, the decision was made to require DT support on all powerpc platforms, regardless of whether or not they used Open Firmware. To do this, a DT representation called the Flattened Device Tree (FDT) was created which could be passed to the kernel as a binary blob without requiring a real Open Firmware implementation. U-Boot, kexec, and other bootloaders were modified to support both passing a Device Tree Binary (dtb) and to modify a dtb at boot time. DT was also added to the PowerPC boot wrapper (`arch/powerpc/boot/*`) so that a dtb could be wrapped up with the kernel image to support booting existing non-DT aware firmware.

Some time later, FDT infrastructure was generalized to be usable by all architectures. At the time of this writing, 6 mainlined architectures (arm, microblaze, mips, powerpc, sparc, and x86) and 1 out of mainline (nios) have some level of DT support.

1.1.2 2. Data Model

If you haven't already read the Device Tree Usage[Page 1, 1] page, then go read it now. It's okay, I'll wait….

1.1.3 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.

1.1.4 2.2 Platform Identification

First and foremost, the kernel will use data in the DT to identify the specific machine. In a perfect world, the specific platform shouldn't matter to the kernel because all platform details would be described perfectly by the device tree in a consistent and reliable manner. Hardware is not perfect though, and so the kernel must identify the machine during early boot so that it has the opportunity to run machine-specific fixups.

In the majority of cases, the machine identity is irrelevant, and the kernel will instead select setup code based on the machine's core CPU or SoC. On ARM for example, setup_arch() in arch/arm/kernel/setup.c will call setup_machine_fdt() in arch/arm/kernel/devtree.c which searches through the machine_desc table and selects the machine_desc which best matches the device tree data. It determines the best match by looking at the 'compatible' property in the root device tree node, and comparing it with the dt_compat list in struct machine_desc (which is defined in arch/arm/include/asm/mach/arch.h if you're curious).

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).

Astute readers might point out that the Beagle xM could also claim compatibility with the original Beagle board. However, one should be cautioned about doing so at the board level since there is typically a high level of change from one board to another, even within the same product line, and it is hard to nail down exactly what is meant when one board claims to be compatible with another. For the top level, it is better to err on the side of caution and not claim one board is compatible with another. The notable exception would be when one board is a carrier for another, such as a CPU module attached to a carrier board.

One more note on compatible values. Any string used in a compatible property must be documented as to what it indicates. Add documentation for compatible strings in Documentation/devicetree/bindings.

Again on ARM, for each machine_desc, the kernel looks to see if any of the dt_compat list entries appear in the compatible property. If one does, then that machine_desc is a candidate for driving the machine. After searching the entire table of machine_descs, setup_machine_fdt() returns the 'most compatible' machine_desc based on which entry in the compatible property each machine_desc matches against. If no matching machine_desc is found, then it returns NULL.

The reasoning behind this scheme is the observation that in the majority of cases, a single machine_desc can support a large number of boards if they all use the same SoC, or same family of SoCs. However, invariably there will be some exceptions where a specific board will require special setup code that is not useful in the generic case. Special cases could be handled by explicitly checking for the troublesome board(s) in generic setup code, but doing so very quickly becomes ugly and/or unmaintainable if it is more than just a couple of cases.

Instead, the compatible list allows a generic machine_desc to provide support for a wide common set of boards by specifying "less compatible" values in the dt_compat list. In the example above, generic board support can claim compatibility with "ti,omap3" or "ti,omap3450". If a bug was discovered on the original beagleboard that required special workaround code during early boot, then a new machine_desc could be added which implements the workarounds and only matches on "ti,omap3-beagleboard".

PowerPC uses a slightly different scheme where it calls the .probe() hook from each machine_desc, and the first one returning TRUE is used. However, this approach does not take into account the priority of the compatible list, and probably should be avoided for new architecture support.

1.1.5 2.3 Runtime configuration

In most cases, a DT will be the sole method of communicating data from firmware to the kernel, so also gets used to pass in runtime and configuration data like the kernel parameters string and the location of an initrd image.

Most of this data is contained in the /chosen node, and when booting Linux it will look something like this:

```
chosen {
        bootargs = "console=ttyS0,115200 loglevel=8";
        initrd-start = <0xc8000000>;
        initrd-end = <0xc8200000>;
};
```

The bootargs property contains the kernel arguments, and the initrd-* properties define the address and size of an initrd blob. Note that initrd-end is the first address after the initrd image, so this doesn't match the usual semantic of struct resource. The chosen node may also optionally contain an arbitrary number of additional properties for platform-specific configuration data.

During early boot, the architecture setup code calls of_scan_flat_dt() several times with different helper callbacks to parse device tree data before paging is setup. The of_scan_flat_dt() code scans through the device tree and uses the helpers to extract information required during early boot. Typically the early_init_dt_scan_chosen() helper is used to parse the chosen node including kernel parameters, early_init_dt_scan_root() to initialize the DT address space model, and early_init_dt_scan_memory() to determine the size and location of usable RAM.

On ARM, the function setup_machine_fdt() is responsible for early scanning of the device tree after selecting the correct machine_desc that supports the board.

1.1.6 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 remainder of this section uses examples from the ARM implementation, but all architectures will do pretty much the same thing when using a DT.

As can be guessed by the names, .init_early() is used for any machine- specific setup that needs to be executed early in the boot process, and .init_irq() is used to set up interrupt handling. Using a DT doesn't materially change the behaviour of either of these functions. If a DT is provided, then both .init_early() and .init_irq() are able to call any of the DT query functions (of_* in include/linux/of*.h) to get additional data about the platform.

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 (more on those later). While there is no 'platform device' terminology for the DT, platform devices roughly correspond to device nodes at the root of the tree and children of simple memory mapped bus nodes.

About now is a good time to lay out an example. Here is part of the device tree for the NVIDIA Tegra board:

```
/{
	compatible = "nvidia,harmony", "nvidia,tegra20";
	#address-cells = <1>;
	#size-cells = <1>;
	interrupt-parent = <&intc>;

	chosen { };
	aliases { };

	memory {
		device_type = "memory";
		reg = <0x00000000 0x40000000>;
	};

	soc {
		compatible = "nvidia,tegra20-soc", "simple-bus";
		#address-cells = <1>;
		#size-cells = <1>;
		ranges;

		intc: interrupt-controller@50041000 {
			compatible = "nvidia,tegra20-gic";
			interrupt-controller;
			#interrupt-cells = <1>;
			reg = <0x50041000 0x1000>, < 0x50040100 0x0100 >;
		};

		serial@70006300 {
			compatible = "nvidia,tegra20-uart";
			reg = <0x70006300 0x100>;
			interrupts = <122>;
		};

		i2s1: i2s@70002800 {
			compatible = "nvidia,tegra20-i2s";
			reg = <0x70002800 0x100>;
			interrupts = <77>;
			codec = <&wm8903>;
		};

		i2c@7000c000 {
			compatible = "nvidia,tegra20-i2c";
```

```
                        #address-cells = <1>;
                        #size-cells = <0>;
                        reg = <0x7000c000 0x100>;
                        interrupts = <70>;

                        wm8903: codec@1a {
                                compatible = "wlf,wm8903";
                                reg = <0x1a>;
                                interrupts = <347>;
                        };
                };
        };

        sound {
                compatible = "nvidia,harmony-sound";
                i2s-controller = <&i2s1>;
                i2s-codec = <&wm8903>;
        };
};
```

At .init_machine() time, Tegra board support code will need to look at this DT and decide which nodes to create platform_devices for. However, looking at the tree, it is not immediately obvious what kind of device each node represents, or even if a node represents a device at all. The /chosen, /aliases, and /memory nodes are informational nodes that don't describe devices (although arguably memory could be considered a device). The children of the /soc node are memory mapped devices, but the codec@1a is an i2c device, and the sound node represents not a device, but rather how other devices are connected together to create the audio subsystem. I know what each device is because I'm familiar with the board design, but how does the kernel know what to do with each node?

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.

Why is using a platform_device for these nodes a safe assumption? Well, for the way that Linux models devices, just about all bus_types assume that its devices are children of a bus controller. For example, each i2c_client is a child of an i2c_master. Each spi_device is a child of an SPI bus. Similarly for USB, PCI, MDIO, etc. The same hierarchy is also found in the DT, where I2C device nodes only ever appear as children of an I2C bus node. Ditto for SPI, MDIO, USB, etc. The only devices which do not require a specific type of parent device are platform_devices (and amba_devices, but more on that later), which will happily live at the base of the Linux /sys/devices tree. Therefore, if a DT node is at the root of the tree, then it really probably is best registered as a platform_device.

Linux board support code calls of_platform_populate(NULL, NULL, NULL, NULL) to kick off discovery of devices at the root of the tree. The parameters are all NULL because when starting from the root of the tree, there is no need to provide a starting node (the first NULL), a parent struct device (the last NULL), and we're not using a match table (yet). For a board that only needs to register devices, .init_machine() can be completely empty except for the *of_platform_populate()* call.

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. In the Tegra case, the code can look something like this:

```
static void __init harmony_init_machine(void)
{
      /* ... */
      of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);
}
```

"simple-bus" is defined in the Devicetree Specification as a property meaning a simple memory mapped bus, so the *of_platform_populate()* code could be written to just assume simple-bus compatible nodes will always be traversed. However, we pass it in as an argument so that board support code can always override the default behaviour.

[Need to add discussion of adding i2c/spi/etc child devices]

1.1.7 Appendix A: AMBA devices

ARM Primecells are a certain kind of device attached to the ARM AMBA bus which include some support for hardware detection and power management. In Linux, struct amba_device and the amba_bus_type is used to represent Primecell devices. However, the fiddly bit is that not all devices on an AMBA bus are Primecells, and for Linux it is typical for both amba_device and platform_device instances to be siblings of the same bus segment.

When using the DT, this creates problems for *of_platform_populate()* because it must decide whether to register each node as either a platform_device or an amba_device. This unfortunately complicates the device creation model a little bit, but the solution turns out not to be too invasive. If a node is compatible with "arm,amba-primecell", then *of_platform_populate()* will register it as an amba_device instead of a platform_device.

1.2 Open Firmware Devicetree Unittest

Author: Gaurav Minocha <gaurav.minocha.os@gmail.com>

1.2.1 1. Introduction

This document explains how the test data required for executing OF unittest is attached to the live tree dynamically, independent of the machine's architecture.

It is recommended to read the following documents before moving ahead.

(1) *Linux and the Devicetree*

(2) http://www.devicetree.org/Device_Tree_Usage

OF Selftest has been designed to test the interface (include/linux/of.h) provided to device driver developers to fetch the device information..etc. from the unflattened device tree data structure. This interface is used by most of the device drivers in various use cases.

1.2.2 2. Verbose Output (EXPECT)

If unittest detects a problem it will print a warning or error message to the console. Unittest also triggers warning and error messages from other kernel code as a result of intentionally bad unittest data. This has led to confusion as to whether the triggered messages are an expected result of a test or whether there is a real problem that is independent of unittest.

'EXPECT : text' (begin) and 'EXPECT / : text' (end) messages have been added to unittest to report that a warning or error is expected. The begin is printed before triggering the warning or error, and the end is printed after triggering the warning or error.

The EXPECT messages result in very noisy console messages that are difficult to read. The script scripts/dtc/of_unittest_expect was created to filter this verbosity and highlight mismatches between triggered warnings and errors vs expected warnings and errors. More information is available from 'scripts/dtc/of_unittest_expect –help'.

1.2.3 3. Test-data

The Device Tree Source file (drivers/of/unittest-data/testcases.dts) contains the test data required for executing the unit tests automated in drivers/of/unittest.c. Currently, following Device Tree Source Include files (.dtsi) are included in testcases.dts:

```
drivers/of/unittest-data/tests-interrupts.dtsi
drivers/of/unittest-data/tests-platform.dtsi
drivers/of/unittest-data/tests-phandle.dtsi
drivers/of/unittest-data/tests-match.dtsi
```

When the kernel is build with OF_SELFTEST enabled, then the following make rule:

```
$(obj)/%.dtb: $(src)/%.dts FORCE
        $(call if_changed_dep, dtc)
```

is used to compile the DT source file (testcases.dts) into a binary blob (testcases.dtb), also referred as flattened DT.

After that, using the following rule the binary blob above is wrapped as an assembly file (testcases.dtb.S):

```
$(obj)/%.dtb.S: $(obj)/%.dtb
        $(call cmd, dt_S_dtb)
```

The assembly file is compiled into an object file (testcases.dtb.o), and is linked into the kernel image.

3.1. Adding the test data

Un-flattened device tree structure:

Un-flattened device tree consists of connected device_node(s) in form of a tree structure described below:

```
// following struct members are used to construct the tree
struct device_node {
    ...
    struct  device_node *parent;
    struct  device_node *child;
    struct  device_node *sibling;
    ...
};
```

Figure 1, describes a generic structure of machine's un-flattened device tree considering only child and sibling pointers. There exists another pointer, `*parent`, that is used to traverse the tree in the reverse direction. So, at a particular level the child node and all the sibling nodes will have a parent pointer pointing to a common node (e.g. child1, sibling2, sibling3, sibling4's parent points to root node):

```
root ('/')
|
child1 -> sibling2 -> sibling3 -> sibling4 -> null
|         |           |           |
|         |           |          null
|         |           |
|         |        child31 -> sibling32 -> null
|         |           |          |
|         |          null       null
|         |
|      child21 -> sibling22 -> sibling23 -> null
|         |          |            |
|        null       null         null
|
child11 -> sibling12 -> sibling13 -> sibling14 -> null
|           |           |            |
|           |           |           null
|           |           |
```

```
null        null       child131 -> null
                         |
                        null
```

Figure 1: Generic structure of un-flattened device tree

Before executing OF unittest, it is required to attach the test data to machine's device tree (if present). So, when selftest_data_add() is called, at first it reads the flattened device tree data linked into the kernel image via the following kernel symbols:

```
__dtb_testcases_begin - address marking the start of test data blob
__dtb_testcases_end   - address marking the end of test data blob
```

Secondly, it calls *of_fdt_unflatten_tree()* to unflatten the flattened blob. And finally, if the machine's device tree (i.e live tree) is present, then it attaches the unflattened test data tree to the live tree, else it attaches itself as a live device tree.

attach_node_and_children() uses of_attach_node() to attach the nodes into the live tree as explained below. To explain the same, the test data tree described in Figure 2 is attached to the live tree described in Figure 1:

```
root ('/')
    |
testcase-data
    |
test-child0 -> test-sibling1 -> test-sibling2 -> test-sibling3 -> null
    |               |                |                 |
test-child01      null             null              null
```

Figure 2: Example test data tree to be attached to live tree.

According to the scenario above, the live tree is already present so it isn't required to attach the root('/') node. All other nodes are attached by calling of_attach_node() on each node.

In the function of_attach_node(), the new node is attached as the child of the given parent in live tree. But, if parent already has a child then the new node replaces the current child and turns it into its sibling. So, when the testcase data node is attached to the live tree above (Figure 1), the final structure is as shown in Figure 3:

```
root ('/')
|
testcase-data -> child1 -> sibling2 -> sibling3 -> sibling4 -> null
|                |          |           |           |
(...)              |          |           |          null
                 |          |         child31 -> sibling32 -> null
                 |          |           |           |
                 |          |          null        null
                 |          |
                 |        child21 -> sibling22 -> sibling23 -> null
                 |          |           |            |
                 |         null        null         null
                 |
                child11 -> sibling12 -> sibling13 -> sibling14 -> null
```

```
                |          |            |            |
                null       null          |           null
                                        |
                                        child131 -> null
                                        |
                                        null
-----------------------------------------------------------------------

root ('/')
|
testcase-data -> child1 -> sibling2 -> sibling3 -> sibling4 -> null
|               |          |           |           |
|             (...)      (...)       (...)        null
|
test-sibling3 -> test-sibling2 -> test-sibling1 -> test-child0 -> null
|                |                   |                |
null             null                null         test-child01
```

Figure 3: Live device tree structure after attaching the testcase-data.

Astute readers would have noticed that test-child0 node becomes the last sibling compared to the earlier structure (Figure 2). After attaching first test-child0 the test-sibling1 is attached that pushes the child node (i.e. test-child0) to become a sibling and makes itself a child node, as mentioned above.

If a duplicate node is found (i.e. if a node with same full_name property is already present in the live tree), then the node isn't attached rather its properties are updated to the live tree's node by calling the function update_node_properties().

3.2. Removing the test data

Once the test case execution is complete, selftest_data_remove is called in order to remove the device nodes attached initially (first the leaf nodes are detached and then moving up the parent nodes are removed, and eventually the whole tree). selftest_data_remove() calls detach_node_and_children() that uses `of_detach_node()` to detach the nodes from the live device tree.

To detach a node, `of_detach_node()` either updates the child pointer of given node's parent to its sibling or attaches the previous sibling to the given node's sibling, as appropriate. That is it :)

1.3 DeviceTree Kernel API

1.3.1 Core functions

struct device_node ***of_find_all_nodes**(struct device_node *prev)
    Get next node in global list

Parameters

struct device_node *prev Previous node or NULL to start iteration *of_node_put()* will be called on it

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done.

struct device_node ***of_get_cpu_node**(int cpu, unsigned int *thread)
Get device node associated with the given logical CPU

Parameters

int cpu CPU number(logical index) for which device node is required

unsigned int *thread if not NULL, local thread number within the physical core is returned

Description

The main purpose of this function is to retrieve the device node for the given logical CPU index. It should be used to initialize the of_node in cpu device. Once of_node in cpu device is populated, all the further references can use that instead.

CPU logical to physical index mapping is architecture specific and is built before booting secondary cores. This function uses arch_match_cpu_phys_id which can be overridden by architecture specific implementation.

Return

A node pointer for the logical cpu with refcount incremented, use *of_node_put()* on it when done. Returns NULL if not found.

int **of_cpu_node_to_id**(struct device_node *cpu_node)
Get the logical CPU number for a given device_node

Parameters

struct device_node *cpu_node Pointer to the device_node for CPU.

Return

The logical CPU number of the given CPU device_node or -ENODEV if the CPU is not found.

struct device_node ***of_get_cpu_state_node**(struct device_node *cpu_node, int index)
Get CPU's idle state node at the given index

Parameters

struct device_node *cpu_node The device node for the CPU

int index The index in the list of the idle states

Description

Two generic methods can be used to describe a CPU's idle states, either via a flattened description through the "cpu-idle-states" binding or via the hierarchical layout, using the "power-domains" and the "domain-idle-states" bindings. This function check for both and returns the idle state node for the requested index.

Return

An idle state node if found at **index**. The refcount is incremented for it, so call *of_node_put()* on it when done. Returns NULL if not found.

int **`of_machine_is_compatible`**(const char *compat)
Test root of device tree for a given compatible value

Parameters

`const char *compat` compatible string to look for in root node's compatible property.

Return

A positive integer if the root node has the given value in its compatible property.

bool **`of_device_is_available`**(const struct device_node *device)
check if a device is available for use

Parameters

`const struct device_node *device` Node to check for availability

Return

True if the status property is absent or set to "okay" or "ok", false otherwise

bool **`of_device_is_big_endian`**(const struct device_node *device)
check if a device has BE registers

Parameters

`const struct device_node *device` Node to check for endianness

Return

True if the device has a "big-endian" property, or if the kernel was compiled for BE *and* the device has a "native-endian" property. Returns false otherwise.

Callers would nominally use ioread32be/iowrite32be if *`of_device_is_big_endian()`* == true, or readl/writel otherwise.

struct device_node ***`of_get_parent`**(const struct device_node *node)
Get a node's parent if any

Parameters

`const struct device_node *node` Node to get parent

Return

A node pointer with refcount incremented, use *`of_node_put()`* on it when done.

struct device_node ***`of_get_next_parent`**(struct device_node *node)
Iterate to a node's parent

Parameters

`struct device_node *node` Node to get parent of

Description

This is like *`of_get_parent()`* except that it drops the refcount on the passed node, making it suitable for iterating through a node's parents.

Return

A node pointer with refcount incremented, use *`of_node_put()`* on it when done.

struct device_node ***of_get_next_child**(const struct device_node *node, struct device_node *prev)

Iterate a node childs

Parameters

const struct device_node *node parent node

struct device_node *prev previous child of the parent node, or NULL to get first

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done. Returns NULL when prev is the last child. Decrements the refcount of prev.

struct device_node ***of_get_next_available_child**(const struct device_node *node, struct device_node *prev)

Find the next available child node

Parameters

const struct device_node *node parent node

struct device_node *prev previous child of the parent node, or NULL to get first

Description

This function is like *of_get_next_child()*, except that it automatically skips any disabled nodes (i.e. status = "disabled").

struct device_node ***of_get_next_cpu_node**(struct device_node *prev)

Iterate on cpu nodes

Parameters

struct device_node *prev previous child of the /cpus node, or NULL to get first

Description

Unusable CPUs (those with the status property set to "fail" or "fail-…") will be skipped.

Return

A cpu node pointer with refcount incremented, use *of_node_put()* on it when done. Returns NULL when prev is the last child. Decrements the refcount of prev.

struct device_node ***of_get_compatible_child**(const struct device_node *parent, const char *compatible)

Find compatible child node

Parameters

const struct device_node *parent parent node

const char *compatible compatible string

Description

Lookup child node whose compatible property contains the given compatible string.

Return

a node pointer with refcount incremented, use *of_node_put()* on it when done; or NULL if not found.

struct device_node ***of_get_child_by_name**(const struct device_node *node, const char *name)
Find the child node by name for a given parent

Parameters

const struct device_node *node parent node

const char *name child name to look for.

Description

This function looks for child node for given matching name

Return

A node pointer if found, with refcount incremented, use *of_node_put()* on it when done. Returns NULL if node is not found.

struct device_node ***of_find_node_opts_by_path**(const char *path, const char **opts)
Find a node matching a full OF path

Parameters

const char *path Either the full path to match, or if the path does not start with '/', the name of a property of the /aliases node (an alias). In the case of an alias, the node matching the alias' value will be returned.

const char **opts Address of a pointer into which to store the start of an options string appended to the end of the path with a ':' separator.

Description

Valid paths:

- /foo/bar Full path
- foo Valid alias
- foo/bar Valid alias + relative path

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done.

struct device_node ***of_find_node_by_name**(struct device_node *from, const char *name)
Find a node by its "name" property

Parameters

struct device_node *from The node to start searching from or NULL; the node you pass will not be searched, only the next one will. Typically, you pass what the previous call returned. *of_node_put()* will be called on **from**.

const char *name The name string to match against

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done.

struct device_node ***of_find_node_by_type**(struct device_node *from, const char *type)
Find a node by its "device_type" property

Parameters

struct device_node *from The node to start searching from, or NULL to start searching the entire device tree. The node you pass will not be searched, only the next one will; typically, you pass what the previous call returned. *of_node_put()* will be called on from for you.

const char *type The type string to match against

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done.

struct device_node ***of_find_compatible_node**(struct device_node *from, const char *type, const char *compatible)

Find a node based on type and one of the tokens in its "compatible" property

Parameters

struct device_node *from The node to start searching from or NULL, the node you pass will not be searched, only the next one will; typically, you pass what the previous call returned. *of_node_put()* will be called on it

const char *type The type string to match "device_type" or NULL to ignore

const char *compatible The string to match to one of the tokens in the device "compatible" list.

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done.

struct device_node ***of_find_node_with_property**(struct device_node *from, const char *prop_name)

Find a node which has a property with the given name.

Parameters

struct device_node *from The node to start searching from or NULL, the node you pass will not be searched, only the next one will; typically, you pass what the previous call returned. *of_node_put()* will be called on it

const char *prop_name The name of the property to look for.

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done.

const struct of_device_id ***of_match_node**(const struct of_device_id *matches, const struct device_node *node)

Tell if a device_node has a matching of_match structure

Parameters

const struct of_device_id *matches array of of device match structures to search in

const struct device_node *node the of device structure to match against

Description

Low level utility function used by device matching.

struct device_node ***of_find_matching_node_and_match**(struct device_node *from, const struct of_device_id *matches, const struct of_device_id **match)

Find a node based on an of_device_id match table.

Parameters

struct device_node *from The node to start searching from or NULL, the node you pass will not be searched, only the next one will; typically, you pass what the previous call returned. *of_node_put()* will be called on it

const struct of_device_id *matches array of of device match structures to search in

const struct of_device_id **match Updated to point at the matches entry which matched

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done.

int **of_modalias_node**(struct device_node *node, char *modalias, int len)
Lookup appropriate modalias for a device node

Parameters

struct device_node *node pointer to a device tree node

char *modalias Pointer to buffer that modalias value will be copied into

int len Length of modalias value

Description

Based on the value of the compatible property, this routine will attempt to choose an appropriate modalias value for a particular device tree node. It does this by stripping the manufacturer prefix (as delimited by a ',') from the first entry in the compatible list property.

Return

This routine returns 0 on success, <0 on failure.

struct device_node ***of_find_node_by_phandle**(phandle handle)
Find a node given a phandle

Parameters

phandle handle phandle of the node to find

Return

A node pointer with refcount incremented, use *of_node_put()* on it when done.

int **of_parse_phandle_with_args_map**(const struct device_node *np, const char *list_name, const char *stem_name, int index, struct of_phandle_args *out_args)
Find a node pointed by phandle in a list and remap it

Parameters

const struct device_node *np pointer to a device tree node containing a list

const char *list_name property name that contains a list

const char *stem_name stem of property names that specify phandles' arguments count

int index index of a phandle to parse out

struct of_phandle_args *out_args optional pointer to output arguments structure (will be filled)

Description

This function is useful to parse lists of phandles and their arguments. Returns 0 on success and fills out_args, on error returns appropriate errno value. The difference between this function and *of_parse_phandle_with_args()* is that this API remaps a phandle if the node the phandle points to has a <**stem_name**>-map property.

Caller is responsible to call *of_node_put()* on the returned out_args->np pointer.

Example:

```
phandle1: node1 {
    #list-cells = <2>;
};

phandle2: node2 {
    #list-cells = <1>;
};

phandle3: node3 {
    #list-cells = <1>;
    list-map = <0 &phandle2 3>,
               <1 &phandle2 2>,
               <2 &phandle1 5 1>;
    list-map-mask = <0x3>;
};

node4 {
    list = <&phandle1 1 2 &phandle3 0>;
};
```

To get a device_node of the `node2` node you may call this: of_parse_phandle_with_args(node4, "list", "list", 1, `args`);

int **of_count_phandle_with_args**(const struct device_node *np, const char *list_name, const char *cells_name)

Find the number of phandles references in a property

Parameters

const struct device_node *np pointer to a device tree node containing a list

const char *list_name property name that contains a list

const char *cells_name property name that specifies phandles' arguments count

Return

The number of phandle + argument tuples within a property. It is a typical pattern to encode a list of phandle and variable arguments into a single property. The number of arguments is encoded by a property in the phandle-target node. For example, a gpios property would contain a list of GPIO specifies consisting of a phandle and 1 or more arguments. The number of arguments are determined by the #gpio-cells property in the node pointed to by the phandle.

int **of_add_property**(struct device_node *np, struct property *prop)

Add a property to a node

Parameters

`struct device_node *np` Caller's Device Node

`struct property *prop` Property to add

int **`of_remove_property`**(struct device_node *np, struct property *prop)
Remove a property from a node.

Parameters

`struct device_node *np` Caller's Device Node

`struct property *prop` Property to remove

Description

Note that we don't actually remove it, since we have given out who-knows-how-many pointers to the data using get-property. Instead we just move the property to the "dead properties" list, so it won't be found any more.

int **`of_alias_get_id`**(struct device_node *np, const char *stem)
Get alias id for the given device_node

Parameters

`struct device_node *np` Pointer to the given device_node

`const char *stem` Alias stem of the given device_node

Description

The function travels the lookup table to get the alias id for the given device_node and alias stem.

Return

The alias id if found.

int **`of_alias_get_highest_id`**(const char *stem)
Get highest alias id for the given stem

Parameters

`const char *stem` Alias stem to be examined

Description

The function travels the lookup table to get the highest alias id for the given alias stem. It returns the alias id if found.

bool **`of_console_check`**(struct device_node *dn, char *name, int index)
Test and setup console for DT setup

Parameters

`struct device_node *dn` Pointer to device node

`char *name` Name to use for preferred console without index. ex. "ttyS"

`int index` Index to use for preferred console.

Description

Check if the given device node matches the stdout-path property in the /chosen node. If it does then register it as the preferred console.

Return

TRUE if console successfully setup. Otherwise return FALSE.

int **of_map_id**(struct device_node *np, u32 id, const char *map_name, const char *map_mask_name, struct device_node **target, u32 *id_out)

Translate an ID through a downstream mapping.

Parameters

struct device_node *np root complex device node.

u32 id device ID to map.

const char *map_name property name of the map to use.

const char *map_mask_name optional property name of the mask to use.

struct device_node **target optional pointer to a target device node.

u32 *id_out optional pointer to receive the translated ID.

Description

Given a device ID, look up the appropriate implementation-defined platform ID and/or the target device which receives transactions on that ID, as per the "iommu-map" and "msi-map" bindings. Either of **target** or **id_out** may be NULL if only the other is required. If **target** points to a non-NULL device node pointer, only entries targeting that node will be matched; if it points to a NULL value, it will receive the device node of the first matching target phandle, with a reference held.

Return

0 on success or a standard error code on failure.

struct device_node ***of_parse_phandle**(const struct device_node *np, const char *phandle_name, int index)

Resolve a phandle property to a device_node pointer

Parameters

const struct device_node *np Pointer to device node holding phandle property

const char *phandle_name Name of property holding a phandle value

int index For properties holding a table of phandles, this is the index into the table

Return

The device_node pointer with refcount incremented. Use *of_node_put()* on it when done.

int **of_parse_phandle_with_args**(const struct device_node *np, const char *list_name, const char *cells_name, int index, struct of_phandle_args *out_args)

Find a node pointed by phandle in a list

Parameters

const struct device_node *np pointer to a device tree node containing a list

const char *list_name property name that contains a list

const char *cells_name property name that specifies phandles' arguments count

int index index of a phandle to parse out

struct of_phandle_args *out_args optional pointer to output arguments structure (will be filled)

Description

This function is useful to parse lists of phandles and their arguments. Returns 0 on success and fills out_args, on error returns appropriate errno value.

Caller is responsible to call *of_node_put()* on the returned out_args->np pointer.

Example:

```
phandle1: node1 {
    #list-cells = <2>;
};

phandle2: node2 {
    #list-cells = <1>;
};

node3 {
    list = <&phandle1 1 2 &phandle2 3>;
};
```

To get a device_node of the `node2` node you may call this: of_parse_phandle_with_args(node3, "list", "#list-cells", 1, `args`);

int **of_parse_phandle_with_fixed_args**(const struct device_node *np, const char *list_name, int cell_count, int index, struct of_phandle_args *out_args)

Find a node pointed by phandle in a list

Parameters

const struct device_node *np pointer to a device tree node containing a list

const char *list_name property name that contains a list

int cell_count number of argument cells following the phandle

int index index of a phandle to parse out

struct of_phandle_args *out_args optional pointer to output arguments structure (will be filled)

Description

This function is useful to parse lists of phandles and their arguments. Returns 0 on success and fills out_args, on error returns appropriate errno value.

Caller is responsible to call *of_node_put()* on the returned out_args->np pointer.

Example:

```
phandle1: node1 {
};
```

```
phandle2: node2 {
};

node3 {
    list = <&phandle1 0 2 &phandle2 2 3>;
};
```

To get a device_node of the `node2` node you may call this: of_parse_phandle_with_fixed_args(node3, "list", 2, 1, `args`);

int **of_property_count_u8_elems**(const struct device_node *np, const char *propname)
Count the number of u8 elements in a property

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

Description

Search for a property in a device node and count the number of u8 elements in it.

Return

The number of elements on sucess, -EINVAL if the property does not exist or its length does not match a multiple of u8 and -ENODATA if the property does not have a value.

int **of_property_count_u16_elems**(const struct device_node *np, const char *propname)
Count the number of u16 elements in a property

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

Description

Search for a property in a device node and count the number of u16 elements in it.

Return

The number of elements on sucess, -EINVAL if the property does not exist or its length does not match a multiple of u16 and -ENODATA if the property does not have a value.

int **of_property_count_u32_elems**(const struct device_node *np, const char *propname)
Count the number of u32 elements in a property

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

Description

Search for a property in a device node and count the number of u32 elements in it.

Return

The number of elements on sucess, -EINVAL if the property does not exist or its length does not match a multiple of u32 and -ENODATA if the property does not have a value.

int **of_property_count_u64_elems**(const struct device_node *np, const char *propname)
Count the number of u64 elements in a property

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

Description

Search for a property in a device node and count the number of u64 elements in it.

Return

The number of elements on sucess, -EINVAL if the property does not exist or its length does not match a multiple of u64 and -ENODATA if the property does not have a value.

int **of_property_read_string_array**(const struct device_node *np, const char *propname, const char **out_strs, size_t sz)
Read an array of strings from a multiple strings property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

const char **out_strs output array of string pointers.

size_t sz number of array elements to read.

Description

Search for a property in a device tree node and retrieve a list of terminated string values (pointer to data, not a copy) in that property.

Return

If **out_strs** is NULL, the number of strings in the property is returned.

int **of_property_count_strings**(const struct device_node *np, const char *propname)
Find and return the number of strings from a multiple strings property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

Description

Search for a property in a device tree node and retrieve the number of null terminated string contain in it.

Return

The number of strings on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EILSEQ if the string is not null-terminated within the length of the property data.

int **of_property_read_string_index**(const struct device_node *np, const char *propname, int index, const char **output)
Find and read a string from a multiple strings property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

int index index of the string in the list of strings

const char **output pointer to null terminated return string, modified only if return value is 0.

Description

Search for a property in a device tree node and retrieve a null terminated string value (pointer to data, not a copy) in the list of strings contained in that property.

The out_string pointer is modified only if a valid string can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EILSEQ if the string is not null-terminated within the length of the property data.

bool **of_property_read_bool**(const struct device_node *np, const char *propname)
Find a property

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

Description

Search for a property in a device node.

Return

true if the property exists false otherwise.

int **of_property_read_u8_array**(const struct device_node *np, const char *propname, u8 *out_values, size_t sz)
Find and read an array of u8 from a property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u8 *out_values pointer to return value, modified only if return value is 0.

size_t sz number of array elements to read

Description

Search for a property in a device node and read 8-bit value(s) from it.

dts entry of array should be like: `property = /bits/ 8 <0x50 0x60 0x70>;`

The out_values is modified only if a valid u8 value can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data isn't large enough.

int **of_property_read_u16_array**(const struct device_node *np, const char *propname, u16 *out_values, size_t sz)
Find and read an array of u16 from a property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u16 *out_values pointer to return value, modified only if return value is 0.

size_t sz number of array elements to read

Description

Search for a property in a device node and read 16-bit value(s) from it.

dts entry of array should be like: `property = /bits/ 16 <0x5000 0x6000 0x7000>;`

The out_values is modified only if a valid u16 value can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data isn't large enough.

int **of_property_read_u32_array**(const struct device_node *np, const char *propname, u32 *out_values, size_t sz)
Find and read an array of 32 bit integers from a property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u32 *out_values pointer to return value, modified only if return value is 0.

size_t sz number of array elements to read

Description

Search for a property in a device node and read 32-bit value(s) from it.

The out_values is modified only if a valid u32 value can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data isn't large enough.

int **of_property_read_u64_array**(const struct device_node *np, const char *propname, u64 *out_values, size_t sz)
Find and read an array of 64 bit integers from a property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u64 *out_values pointer to return value, modified only if return value is 0.

size_t sz number of array elements to read

Description

Search for a property in a device node and read 64-bit value(s) from it.

The out_values is modified only if a valid u64 value can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data isn't large enough.

struct **of_changeset_entry**
Holds a changeset entry

Definition

```
struct of_changeset_entry {
  struct list_head node;
  unsigned long action;
  struct device_node *np;
  struct property *prop;
  struct property *old_prop;
};
```

Members

node list_head for the log list

action notifier action

np pointer to the device node affected

prop pointer to the property affected

old_prop hold a pointer to the original property

Description

Every modification of the device tree during a changeset is held in a list of of_changeset_entry structures. That way we can recover from a partial application, or we can revert the changeset

struct **of_changeset**
changeset tracker structure

Definition

```
struct of_changeset {
  struct list_head entries;
};
```

Members

entries list_head for the changeset entries

Description

changesets are a convenient way to apply bulk changes to the live tree. In case of an error, changes are rolled-back. changesets live on after initial application, and if not destroyed after use, they can be reverted in one single call.

bool **of_device_is_system_power_controller**(const struct device_node *np)
Tells if system-power-controller is found for device_node

Parameters

const struct device_node *np Pointer to the given device_node

Return

true if present false otherwise

bool **of_graph_is_present**(const struct device_node *node)
check graph's presence

Parameters

const struct device_node *node pointer to device_node containing graph port

Return

True if **node** has a port or ports (with a port) sub-node, false otherwise.

int **of_property_count_elems_of_size**(const struct device_node *np, const char *propname, int elem_size)
Count the number of elements in a property

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

int elem_size size of the individual element

Description

Search for a property in a device node and count the number of elements of size elem_size in it.

Return

The number of elements on sucess, -EINVAL if the property does not exist or its length does not match a multiple of elem_size and -ENODATA if the property does not have a value.

int **of_property_read_u32_index**(const struct device_node *np, const char *propname, u32 index, u32 *out_value)
Find and read a u32 from a multi-value property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u32 index index of the u32 in the list of values

u32 *out_value pointer to return value, modified only if no error.

Description

Search for a property in a device node and read nth 32-bit value from it.

The out_value is modified only if a valid u32 value can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data isn't large enough.

int **of_property_read_u64_index**(const struct device_node *np, const char *propname, u32 index, u64 *out_value)

Find and read a u64 from a multi-value property.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u32 index index of the u64 in the list of values

u64 *out_value pointer to return value, modified only if no error.

Description

Search for a property in a device node and read nth 64-bit value from it.

The out_value is modified only if a valid u64 value can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data isn't large enough.

int **of_property_read_variable_u8_array**(const struct device_node *np, const char *propname, u8 *out_values, size_t sz_min, size_t sz_max)

Find and read an array of u8 from a property, with bounds on the minimum and maximum array size.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u8 *out_values pointer to found values.

size_t sz_min minimum number of array elements to read

size_t sz_max maximum number of array elements to read, if zero there is no upper limit on the number of elements in the dts entry but only sz_min will be read.

Description

Search for a property in a device node and read 8-bit value(s) from it.

dts entry of array should be like: `property = /bits/ 8 <0x50 0x60 0x70>;`

The out_values is modified only if a valid u8 value can be decoded.

Return

The number of elements read on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data is smaller than sz_min or longer than sz_max.

int **of_property_read_variable_u16_array**(const struct device_node *np, const char *propname, u16 *out_values, size_t sz_min, size_t sz_max)

Find and read an array of u16 from a property, with bounds on the minimum and maximum array size.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u16 *out_values pointer to found values.

size_t sz_min minimum number of array elements to read

size_t sz_max maximum number of array elements to read, if zero there is no upper limit on the number of elements in the dts entry but only sz_min will be read.

Description

Search for a property in a device node and read 16-bit value(s) from it.

dts entry of array should be like: `property = /bits/ 16 <0x5000 0x6000 0x7000>;`

The out_values is modified only if a valid u16 value can be decoded.

Return

The number of elements read on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data is smaller than sz_min or longer than sz_max.

int **of_property_read_variable_u32_array**(const struct device_node *np, const char *propname, u32 *out_values, size_t sz_min, size_t sz_max)

Find and read an array of 32 bit integers from a property, with bounds on the minimum and maximum array size.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u32 *out_values pointer to return found values.

size_t sz_min minimum number of array elements to read

size_t sz_max maximum number of array elements to read, if zero there is no upper limit on the number of elements in the dts entry but only sz_min will be read.

Description

Search for a property in a device node and read 32-bit value(s) from it.

The out_values is modified only if a valid u32 value can be decoded.

Return

The number of elements read on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data is smaller than sz_min or longer than sz_max.

int **of_property_read_u64**(const struct device_node *np, const char *propname, u64 *out_value)

Find and read a 64 bit integer from a property

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u64 *out_value pointer to return value, modified only if return value is 0.

Description

Search for a property in a device node and read a 64-bit value from it.

The out_value is modified only if a valid u64 value can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data isn't large enough.

int **of_property_read_variable_u64_array**(const struct device_node *np, const char *propname, u64 *out_values, size_t sz_min, size_t sz_max)

Find and read an array of 64 bit integers from a property, with bounds on the minimum and maximum array size.

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

u64 *out_values pointer to found values.

size_t sz_min minimum number of array elements to read

size_t sz_max maximum number of array elements to read, if zero there is no upper limit on the number of elements in the dts entry but only sz_min will be read.

Description

Search for a property in a device node and read 64-bit value(s) from it.

The out_values is modified only if a valid u64 value can be decoded.

Return

The number of elements read on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EOVERFLOW if the property data is smaller than sz_min or longer than sz_max.

int **of_property_read_string**(const struct device_node *np, const char *propname, const char **out_string)

Find and read a string from a property

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

const char **out_string pointer to null terminated return string, modified only if return value is 0.

Description

Search for a property in a device tree node and retrieve a null terminated string value (pointer to data, not a copy).

Note that the empty string “” has length of 1, thus -ENODATA cannot be interpreted as an empty string.

The out_string pointer is modified only if a valid string can be decoded.

Return

0 on success, -EINVAL if the property does not exist, -ENODATA if property does not have a value, and -EILSEQ if the string is not null-terminated within the length of the property data.

int **of_property_match_string**(const struct device_node *np, const char *propname, const char *string)

Find string in a list and return index

Parameters

const struct device_node *np pointer to node containing string list property

const char *propname string list property name

const char *string pointer to string to search for in string list

Description

This function searches a string list property and returns the index of a specific string value.

int **of_property_read_string_helper**(const struct device_node *np, const char *propname, const char **out_strs, size_t sz, int skip)

Utility helper for parsing string properties

Parameters

const struct device_node *np device node from which the property value is to be read.

const char *propname name of the property to be searched.

const char **out_strs output array of string pointers.

size_t sz number of array elements to read.

int skip Number of strings to skip over at beginning of list.

Description

Don’t call this function directly. It is a utility helper for the of_property_read_string*() family of functions.

int **of_graph_parse_endpoint**(const struct device_node *node, struct *of_endpoint* *endpoint)

parse common endpoint node properties

Parameters

const struct device_node *node pointer to endpoint device_node

struct of_endpoint *endpoint pointer to the OF endpoint data structure

Description

The caller should hold a reference to **node**.

struct device_node ***of_graph_get_port_by_id**(struct device_node *parent, u32 id)
get the port matching a given id

Parameters

struct device_node *parent pointer to the parent device node

u32 id id of the port

Return

A 'port' node pointer with refcount incremented. The caller has to use *of_node_put()* on it when done.

struct device_node ***of_graph_get_next_endpoint**(const struct device_node *parent, struct device_node *prev)
get next endpoint node

Parameters

const struct device_node *parent pointer to the parent device node

struct device_node *prev previous endpoint node, or NULL to get first

Return

An 'endpoint' node pointer with refcount incremented. Refcount of the passed **prev** node is decremented.

struct device_node ***of_graph_get_endpoint_by_regs**(const struct device_node *parent, int port_reg, int reg)
get endpoint node of specific identifiers

Parameters

const struct device_node *parent pointer to the parent device node

int port_reg identifier (value of reg property) of the parent port node

int reg identifier (value of reg property) of the endpoint node

Return

An 'endpoint' node pointer which is identified by reg and at the same is the child of a port node identified by port_reg. reg and port_reg are ignored when they are -1. Use *of_node_put()* on the pointer when done.

struct device_node ***of_graph_get_remote_endpoint**(const struct device_node *node)
get remote endpoint node

Parameters

const struct device_node *node pointer to a local endpoint device_node

Return

Remote endpoint node associated with remote endpoint node linked to **node**. Use *of_node_put()* on it when done.

struct device_node *`of_graph_get_port_parent`(struct device_node *node)
get port's parent node

Parameters

`struct device_node *node` pointer to a local endpoint device_node

Return

device node associated with endpoint node linked to **node**. Use *`of_node_put()`* on it when done.

struct device_node *`of_graph_get_remote_port_parent`(const struct device_node *node)
get remote port's parent node

Parameters

`const struct device_node *node` pointer to a local endpoint device_node

Return

Remote device node associated with remote endpoint node linked to **node**. Use *`of_node_put()`* on it when done.

struct device_node *`of_graph_get_remote_port`(const struct device_node *node)
get remote port node

Parameters

`const struct device_node *node` pointer to a local endpoint device_node

Return

Remote port node associated with remote endpoint node linked to **node**. Use *`of_node_put()`* on it when done.

struct device_node *`of_graph_get_remote_node`(const struct device_node *node, u32 port, u32 endpoint)
get remote parent device_node for given port/endpoint

Parameters

`const struct device_node *node` pointer to parent device_node containing graph port/endpoint

`u32 port` identifier (value of reg property) of the parent port node

`u32 endpoint` identifier (value of reg property) of the endpoint node

Return

Remote device node associated with remote endpoint node linked to **node**. Use *`of_node_put()`* on it when done.

struct `of_endpoint`
the OF graph endpoint data structure

Definition

```
struct of_endpoint {
  unsigned int port;
  unsigned int id;
```

```
  const struct device_node *local_node;
};
```

Members

port identifier (value of reg property) of a port this endpoint belongs to

id identifier (value of reg property) of this endpoint

local_node pointer to device_node of this endpoint

for_each_endpoint_of_node

`for_each_endpoint_of_node (parent, child)`

iterate over every endpoint in a device node

Parameters

parent parent device node containing ports and endpoints

child loop variable pointing to the current endpoint node

Description

When breaking out of the loop, of_node_put(child) has to be called manually.

int **of_address_to_resource**(struct device_node *dev, int index, struct resource *r)
Translate device tree address and return as resource

Parameters

struct device_node *dev Caller's Device Node

int index Index into the array

struct resource *r Pointer to resource array

Description

Note that if your address is a PIO address, the conversion will fail if the physical address can't be internally converted to an IO token with pci_address_to_pio(), that is because it's either called too early or it can't be matched to any host bridge IO space

void __iomem ***of_iomap**(struct device_node *np, int index)
Maps the memory mapped IO for a given device_node

Parameters

struct device_node *np the device whose io range will be mapped

int index index of the io range

Description

Returns a pointer to the mapped memory

bool **of_dma_is_coherent**(struct device_node *np)
Check if device is coherent

Parameters

struct device_node *np device node

Description

It returns true if "dma-coherent" property was found for this device in the DT, or if DMA is coherent by default for OF devices on the current platform.

unsigned int **irq_of_parse_and_map**(struct device_node *dev, int index)
Parse and map an interrupt into linux virq space

Parameters

struct device_node *dev Device node of the device whose interrupt is to be mapped

int index Index of the interrupt to map

Description

This function is a wrapper that chains *of_irq_parse_one()* and irq_create_of_mapping() to make things easier to callers

struct device_node ***of_irq_find_parent**(struct device_node *child)
Given a device node, find its interrupt parent node

Parameters

struct device_node *child pointer to device node

Return

A pointer to the interrupt parent node, or NULL if the interrupt parent could not be determined.

int **of_irq_parse_raw**(const __be32 *addr, struct of_phandle_args *out_irq)
Low level interrupt tree parsing

Parameters

const __be32 *addr address specifier (start of "reg" property of the device) in be32 format

struct of_phandle_args *out_irq structure of_phandle_args updated by this function

Description

This function is a low-level interrupt tree walking function. It can be used to do a partial walk with synthetized reg and interrupts properties, for example when resolving PCI interrupts when no device node exist for the parent. It takes an interrupt specifier structure as input, walks the tree looking for any interrupt-map properties, translates the specifier for each map, and then returns the translated map.

Return

0 on success and a negative number on error

int **of_irq_parse_one**(struct device_node *device, int index, struct of_phandle_args *out_irq)
Resolve an interrupt for a device

Parameters

struct device_node *device the device whose interrupt is to be resolved

int index index of the interrupt to resolve

struct of_phandle_args *out_irq structure of_phandle_args filled by this function

Description

This function resolves an interrupt for a node by walking the interrupt tree, finding which interrupt controller node it is attached to, and returning the interrupt specifier that can be used to retrieve a Linux IRQ number.

int **of_irq_to_resource**(struct device_node *dev, int index, struct resource *r)
Decode a node's IRQ and return it as a resource

Parameters

struct device_node *dev pointer to device tree node

int index zero-based index of the irq

struct resource *r pointer to resource structure to return result into.

int **of_irq_get**(struct device_node *dev, int index)
Decode a node's IRQ and return it as a Linux IRQ number

Parameters

struct device_node *dev pointer to device tree node

int index zero-based index of the IRQ

Return

Linux IRQ number on success, or 0 on the IRQ mapping failure, or -EPROBE_DEFER if the IRQ domain is not yet created, or error code in case of any other failure.

int **of_irq_get_byname**(struct device_node *dev, const char *name)
Decode a node's IRQ and return it as a Linux IRQ number

Parameters

struct device_node *dev pointer to device tree node

const char *name IRQ name

Return

Linux IRQ number on success, or 0 on the IRQ mapping failure, or -EPROBE_DEFER if the IRQ domain is not yet created, or error code in case of any other failure.

int **of_irq_to_resource_table**(struct device_node *dev, struct resource *res, int nr_irqs)
Fill in resource table with node's IRQ info

Parameters

struct device_node *dev pointer to device tree node

struct resource *res array of resources to fill in

int nr_irqs the number of IRQs (and upper bound for num of **res** elements)

Return

The size of the filled in table (up to **nr_irqs**).

void **of_msi_configure**(struct device *dev, struct device_node *np)
Set the msi_domain field of a device

Parameters

struct device *dev device structure to associate with an MSI irq domain

struct device_node *np device node for that device

void *`of_fdt_unflatten_tree`(const unsigned long *blob, struct device_node *dad, struct device_node **mynodes)
create tree of device_nodes from flat blob

Parameters

const unsigned long *blob Flat device tree blob

struct device_node *dad Parent device node

struct device_node **mynodes The device tree created by the call

Description

unflattens the device-tree passed by the firmware, creating the tree of struct device_node. It also fills the "name" and "type" pointers of the nodes so the normal device-tree walking functions can be used.

Return

NULL on failure or the memory chunk containing the unflattened device tree on success.

1.3.2 Driver model functions

int `of_driver_match_device`(struct device *dev, const struct device_driver *drv)
Tell if a driver's of_match_table matches a device.

Parameters

struct device *dev the device structure to match against

const struct device_driver *drv the device_driver structure to test

const struct of_device_id *`of_match_device`(const struct of_device_id *matches, const struct device *dev)
Tell if a struct device matches an of_device_id list

Parameters

const struct of_device_id *matches array of of device match structures to search in

const struct device *dev the of device structure to match against

Description

Used by a driver to check whether an platform_device present in the system is in its list of supported devices.

int `of_dma_configure_id`(struct device *dev, struct device_node *np, bool force_dma, const u32 *id)
Setup DMA configuration

Parameters

struct device *dev Device to apply DMA configuration

struct device_node *np Pointer to OF node having DMA configuration

bool force_dma Whether device is to be set up by of_dma_configure() even if DMA capability is not explicitly described by firmware.

const u32 *id Optional const pointer value input id

Description

Try to get devices's DMA configuration from DT and update it accordingly.

If platform code needs to use its own special DMA configuration, it can use a platform bus notifier and handle BUS_NOTIFY_ADD_DEVICE events to fix up DMA configuration.

ssize_t **of_device_modalias**(struct device *dev, char *str, ssize_t len)
Fill buffer with newline terminated modalias string

Parameters

struct device *dev Calling device

char *str Modalias string

ssize_t len Size of **str**

struct **of_dev_auxdata**
lookup table entry for device names & platform_data

Definition

```
struct of_dev_auxdata {
  char *compatible;
  resource_size_t phys_addr;
  char *name;
  void *platform_data;
};
```

Members

compatible compatible value of node to match against node

phys_addr Start address of registers to match against node

name Name to assign for matching nodes

platform_data platform_data to assign for matching nodes

Description

This lookup table allows the caller of *of_platform_populate()* to override the names of devices when creating devices from the device tree. The table should be terminated with an empty entry. It also allows the platform_data pointer to be set.

The reason for this functionality is that some Linux infrastructure uses the device name to look up a specific device, but the Linux-specific names are not encoded into the device tree, so the kernel needs to provide specific values.

Note

Using an auxdata lookup table should be considered a last resort when converting a platform to use the DT. Normally the automatically generated device name will not matter, and drivers should obtain data from the device node instead of from an anonymous platform_data pointer.

struct platform_device ***of_find_device_by_node**(struct device_node *np)
Find the platform_device associated with a node

Parameters

struct device_node *np Pointer to device tree node

Description

Takes a reference to the embedded struct device which needs to be dropped after use.

Return

platform_device pointer, or NULL if not found

struct platform_device ***of_device_alloc**(struct device_node *np, const char *bus_id, struct device *parent)
Allocate and initialize an of_device

Parameters

struct device_node *np device node to assign to device

const char *bus_id Name to assign to the device. May be null to use default name.

struct device *parent Parent device.

struct platform_device ***of_platform_device_create**(struct device_node *np, const char *bus_id, struct device *parent)
Alloc, initialize and register an of_device

Parameters

struct device_node *np pointer to node to create device for

const char *bus_id name to assign device

struct device *parent Linux device model parent device.

Return

Pointer to created platform device, or NULL if a device was not registered. Unavailable devices will not get registered.

int **of_platform_bus_probe**(struct device_node *root, const struct of_device_id *matches, struct device *parent)
Probe the device-tree for platform buses

Parameters

struct device_node *root parent of the first level to probe or NULL for the root of the tree

const struct of_device_id *matches match table for bus nodes

struct device *parent parent to hook devices from, NULL for toplevel

Description

Note that children of the provided root are not instantiated as devices unless the specified root itself matches the bus list and is not NULL.

int **of_platform_populate**(struct device_node *root, const struct of_device_id *matches, const struct *of_dev_auxdata* *lookup, struct device *parent)
Populate platform_devices from device tree data

Parameters

struct device_node *root parent of the first level to probe or NULL for the root of the tree

const struct of_device_id *matches match table, NULL to use the default

const struct of_dev_auxdata *lookup auxdata table for matching id and platform_data with device nodes

struct device *parent parent to hook devices from, NULL for toplevel

Description

Similar to *of_platform_bus_probe()*, this function walks the device tree and creates devices from nodes. It differs in that it follows the modern convention of requiring all device nodes to have a 'compatible' property, and it is suitable for creating devices which are children of the root node (of_platform_bus_probe will only create children of the root which are selected by the **matches** argument).

New board support should be using this function instead of *of_platform_bus_probe()*.

Return

0 on success, < 0 on failure.

void **of_platform_depopulate**(struct device *parent)
Remove devices populated from device tree

Parameters

struct device *parent device which children will be removed

Description

Complementary to *of_platform_populate()*, this function removes children of the given device (and, recurrently, their children) that have been created from their respective device tree nodes (and only those, leaving others - eg. manually created - unharmed).

int **devm_of_platform_populate**(struct device *dev)
Populate platform_devices from device tree data

Parameters

struct device *dev device that requested to populate from device tree data

Description

Similar to *of_platform_populate()*, but will automatically call *of_platform_depopulate()* when the device is unbound from the bus.

Return

0 on success, < 0 on failure.

void **devm_of_platform_depopulate**(struct device *dev)
Remove devices populated from device tree

Parameters

struct device *dev device that requested to depopulate from device tree data

Description

Complementary to *devm_of_platform_populate()*, this function removes children of the given device (and, recurrently, their children) that have been created from their respective device tree nodes (and only those, leaving others - eg. manually created - unharmed).

1.3.3 Overlay and Dynamic DT functions

int **of_resolve_phandles**(struct device_node *overlay)
Relocate and resolve overlay against live tree

Parameters

struct device_node *overlay Pointer to devicetree overlay to relocate and resolve

Description

Modify (relocate) values of local phandles in **overlay** to a range that does not conflict with the live expanded devicetree. Update references to the local phandles in **overlay**. Update (resolve) phandle references in **overlay** that refer to the live expanded devicetree.

Phandle values in the live tree are in the range of 1 .. live_tree_max_phandle(). The range of phandle values in the overlay also begin with at 1. Adjust the phandle values in the overlay to begin at live_tree_max_phandle() + 1. Update references to the phandles to the adjusted phandle values.

The name of each property in the "__fixups__" node in the overlay matches the name of a symbol (a label) in the live tree. The values of each property in the "__fixups__" node is a list of the property values in the overlay that need to be updated to contain the phandle reference corresponding to that symbol in the live tree. Update the references in the overlay with the phandle values in the live tree.

overlay must be detached.

Resolving and applying **overlay** to the live expanded devicetree must be protected by a mechanism to ensure that multiple overlays are processed in a single threaded manner so that multiple overlays will not relocate phandles to overlapping ranges. The mechanism to enforce this is not yet implemented.

Return

`0` on success or a negative error value on error.

struct device_node ***of_node_get**(struct device_node *node)
Increment refcount of a node

Parameters

struct device_node *node Node to inc refcount, NULL is supported to simplify writing of callers

Return

The node with refcount incremented.

void **of_node_put**(struct device_node *node)
Decrement refcount of a node

Parameters

struct device_node *node Node to dec refcount, NULL is supported to simplify writing of callers

int **of_detach_node**(struct device_node *np)
"Unplug" a node from the device tree.

Parameters

struct device_node *np Pointer to the caller's Device Node

void **of_changeset_init**(struct *of_changeset* *ocs)
Initialize a changeset for use

Parameters

struct of_changeset *ocs changeset pointer

Description

Initialize a changeset structure

void **of_changeset_destroy**(struct *of_changeset* *ocs)
Destroy a changeset

Parameters

struct of_changeset *ocs changeset pointer

Description

Destroys a changeset. Note that if a changeset is applied, its changes to the tree cannot be reverted.

int **of_changeset_apply**(struct *of_changeset* *ocs)
Applies a changeset

Parameters

struct of_changeset *ocs changeset pointer

Description

Applies a changeset to the live tree. Any side-effects of live tree state changes are applied here on success, like creation/destruction of devices and side-effects like creation of sysfs properties and directories.

Return

0 on success, a negative error value in case of an error. On error the partially applied effects are reverted.

int **of_changeset_revert**(struct *of_changeset* *ocs)
Reverts an applied changeset

Parameters

struct of_changeset *ocs changeset pointer

Description

Reverts a changeset returning the state of the tree to what it was before the application. Any side-effects like creation/destruction of devices and removal of sysfs properties and directories are applied.

Return

0 on success, a negative error value in case of an error.

int **of_changeset_action**(struct *of_changeset* *ocs, unsigned long action, struct device_node *np, struct property *prop)
Add an action to the tail of the changeset list

Parameters

struct of_changeset *ocs changeset pointer

unsigned long action action to perform

struct device_node *np Pointer to device node

struct property *prop Pointer to property

Description

On action being one of: + OF_RECONFIG_ATTACH_NODE + OF_RECONFIG_DETACH_NODE, + OF_RECONFIG_ADD_PROPERTY + OF_RECONFIG_REMOVE_PROPERTY, + OF_RECONFIG_UPDATE_PROPERTY

Return

0 on success, a negative error value in case of an error.

int **of_overlay_notifier_register**(struct notifier_block *nb)
Register notifier for overlay operations

Parameters

struct notifier_block *nb Notifier block to register

Description

Register for notification on overlay operations on device tree nodes. The reported actions definied by **of_reconfig_change**. The notifier callback furthermore receives a pointer to the affected device tree node.

Note that a notifier callback is not supposed to store pointers to a device tree node or its content beyond **OF_OVERLAY_POST_REMOVE** corresponding to the respective node it received.

int **of_overlay_notifier_unregister**(struct notifier_block *nb)
Unregister notifier for overlay operations

Parameters

struct notifier_block *nb Notifier block to unregister

int **of_overlay_remove**(int *ovcs_id)
Revert and free an overlay changeset

Parameters

int *ovcs_id Pointer to overlay changeset id

Description

Removes an overlay if it is permissible. **ovcs_id** was previously returned by of_overlay_fdt_apply().

If an error occurred while attempting to revert the overlay changeset, then an attempt is made to re-apply any changeset entry that was reverted. If an error occurs on re-apply then the state of the device tree can not be determined, and any following attempt to apply or remove an overlay changeset will be refused.

A non-zero return value will not revert the changeset if error is from:

- parameter checks
- overlay changeset pre-remove notifier
- overlay changeset entry revert

If an error is returned by an overlay changeset pre-remove notifier then no further overlay changeset pre-remove notifier will be called.

If more than one notifier returns an error, then the last notifier error to occur is returned.

A non-zero return value will revert the changeset if error is from:

- overlay changeset entry notifier
- overlay changeset post-remove notifier

If an error is returned by an overlay changeset post-remove notifier then no further overlay changeset post-remove notifier will be called.

Return

0 on success, or a negative error number. ***ovcs_id** is set to zero after reverting the changeset, even if a subsequent error occurs.

int **`of_overlay_remove_all`**(void)
Reverts and frees all overlay changesets

Parameters

`void` no arguments

Description

Removes all overlays from the system in the correct order.

Return

0 on success, or a negative error number

CHAPTER

TWO

DEVICETREE OVERLAYS

2.1 Devicetree Changesets

A Devicetree changeset is a method which allows one to apply changes in the live tree in such a way that either the full set of changes will be applied, or none of them will be. If an error occurs partway through applying the changeset, then the tree will be rolled back to the previous state. A changeset can also be removed after it has been applied.

When a changeset is applied, all of the changes get applied to the tree at once before emitting OF_RECONFIG notifiers. This is so that the receiver sees a complete and consistent state of the tree when it receives the notifier.

The sequence of a changeset is as follows.

1. *of_changeset_init()* - initializes a changeset
2. A number of DT tree change calls, of_changeset_attach_node(), of_changeset_detach_node(), of_changeset_add_property(), of_changeset_remove_property, of_changeset_update_property() to prepare a set of changes. No changes to the active tree are made at this point. All the change operations are recorded in the of_changeset 'entries' list.
3. *of_changeset_apply()* - Apply the changes to the tree. Either the entire changeset will get applied, or if there is an error the tree will be restored to the previous state. The core ensures proper serialization through locking. An unlocked version __of_changeset_apply is available, if needed.

If a successfully applied changeset needs to be removed, it can be done with *of_changeset_revert()*.

2.2 Devicetree Dynamic Resolver Notes

This document describes the implementation of the in-kernel DeviceTree resolver, residing in drivers/of/resolver.c

2.2.1 How the resolver works

The resolver is given as an input an arbitrary tree compiled with the proper dtc option and having a /plugin/ tag. This generates the appropriate __fixups__ & __local_fixups__ nodes.

In sequence the resolver works by the following steps:

1. Get the maximum device tree phandle value from the live tree + 1.
2. Adjust all the local phandles of the tree to resolve by that amount.
3. Using the __local__fixups__ node information adjust all local references by the same amount.
4. For each property in the __fixups__ node locate the node it references in the live tree. This is the label used to tag the node.
5. Retrieve the phandle of the target of the fixup.
6. For each fixup in the property locate the node:property:offset location and replace it with the phandle value.

2.3 Devicetree Overlay Notes

This document describes the implementation of the in-kernel device tree overlay functionality residing in drivers/of/overlay.c and is a companion document to *Devicetree Dynamic Resolver Notes*[1]

2.3.1 How overlays work

A Devicetree's overlay purpose is to modify the kernel's live tree, and have the modification affecting the state of the kernel in a way that is reflecting the changes. Since the kernel mainly deals with devices, any new device node that result in an active device should have it created while if the device node is either disabled or removed all together, the affected device should be deregistered.

Lets take an example where we have a foo board with the following base tree:

```
---- foo.dts ---------------------------------------------------------------
    /* FOO platform */
    /dts-v1/;
    / {
            compatible = "corp,foo";

            /* shared resources */
            res: res {
            };

            /* On chip peripherals */
            ocp: ocp {
                    /* peripherals that are always instantiated */
                    peripheral1 { ... };
            };
```

```
    };
---- foo.dts --------------------------------------------------------------
```

The overlay bar.dts,

```
---- bar.dts - overlay target location by label ---------------------------
    /dts-v1/;
    /plugin/;
    &ocp {
            /* bar peripheral */
            bar {
                    compatible = "corp,bar";
                    ... /* various properties and child nodes */
            };
    };
---- bar.dts --------------------------------------------------------------
```

when loaded (and resolved as described in [1]) should result in foo+bar.dts:

```
---- foo+bar.dts ----------------------------------------------------------
    /* FOO platform + bar peripheral */
    / {
            compatible = "corp,foo";

            /* shared resources */
            res: res {
            };

            /* On chip peripherals */
            ocp: ocp {
                    /* peripherals that are always instantiated */
                    peripheral1 { ... };

                    /* bar peripheral */
                    bar {
                            compatible = "corp,bar";
                            ... /* various properties and child nodes */
                    };
            };
    };
---- foo+bar.dts ----------------------------------------------------------
```

As a result of the overlay, a new device node (bar) has been created so a bar platform device will be registered and if a matching device driver is loaded the device will be created as expected.

If the base DT was not compiled with the -@ option then the “&ocp” label will not be available to resolve the overlay node(s) to the proper location in the base DT. In this case, the target path can be provided. The target location by label syntax is preferred because the overlay can be applied to any base DT containing the label, no matter where the label occurs in the DT.

The above bar.dts example modified to use target path syntax is:

```
---- bar.dts - overlay target location by explicit path --------------------
    /dts-v1/;
    /plugin/;
    &{/ocp} {
            /* bar peripheral */
            bar {
                    compatible = "corp,bar";
                    ... /* various properties and child nodes */
            }
    };
---- bar.dts ---------------------------------------------------------------
```

2.3.2 Overlay in-kernel API

The API is quite easy to use.

1) Call of_overlay_fdt_apply() to create and apply an overlay changeset. The return value is an error or a cookie identifying this overlay.
2) Call *of_overlay_remove()* to remove and cleanup the overlay changeset previously created via the call to of_overlay_fdt_apply(). Removal of an overlay changeset that is stacked by another will not be permitted.

Finally, if you need to remove all overlays in one-go, just call *of_overlay_remove_all()* which will remove every single one in the correct order.

There is the option to register notifiers that get called on overlay operations. See of_overlay_notifier_register/unregister and enum of_overlay_notify_action for details.

A notifier callback for OF_OVERLAY_PRE_APPLY, OF_OVERLAY_POST_APPLY, or OF_OVERLAY_PRE_REMOVE may store pointers to a device tree node in the overlay or its content but these pointers must not persist past the notifier callback for OF_OVERLAY_POST_REMOVE. The memory containing the overlay will be kfree()ed after OF_OVERLAY_POST_REMOVE notifiers are called. Note that the memory will be kfree()ed even if the notifier for OF_OVERLAY_POST_REMOVE returns an error.

The changeset notifiers in drivers/of/dynamic.c are a second type of notifier that could be triggered by applying or removing an overlay. These notifiers are not allowed to store pointers to a device tree node in the overlay or its content. The overlay code does not protect against such pointers remaining active when the memory containing the overlay is freed as a result of removing the overlay.

Any other code that retains a pointer to the overlay nodes or data is considered to be a bug because after removing the overlay the pointer will refer to freed memory.

Users of overlays must be especially aware of the overall operations that occur on the system to ensure that other kernel code does not retain any pointers to the overlay nodes or data. Any example of an inadvertent use of such pointers is if a driver or subsystem module is loaded after an overlay has been applied, and the driver or subsystem scans the entire devicetree or a large portion of it, including the overlay nodes.

CHAPTER

THREE

DEVICETREE BINDINGS

3.1 Devicetree (DT) ABI

I. Regarding stable bindings/ABI, we quote from the 2013 ARM mini-summit summary document:

> "That still leaves the question of, what does a stable binding look like? Certainly a stable binding means that a newer kernel will not break on an older device tree, but that doesn't mean the binding is frozen for all time. Grant said there are ways to change bindings that don't result in breakage. For instance, if a new property is added, then default to the previous behaviour if it is missing. If a binding truly needs an incompatible change, then change the compatible string at the same time. The driver can bind against both the old and the new. These guidelines aren't new, but they desperately need to be documented."

II. General binding rules

1) Maintainers, don't let perfect be the enemy of good. Don't hold up a binding because it isn't perfect.
2) Use specific compatible strings so that if we need to add a feature (DMA) in the future, we can create a new compatible string. See I.
3) Bindings can be augmented, but the driver shouldn't break when given the old binding. ie. add additional properties, but don't change the meaning of an existing property. For drivers, default to the original behaviour when a newly added property is missing.
4) Don't submit bindings for staging or unstable. That will be decided by the devicetree maintainers *after* discussion on the mailinglist.

III. Notes

1) This document is intended as a general familiarization with the process as decided at the 2013 Kernel Summit. When in doubt, the current word of the devicetree maintainers overrules this document. In that situation, a patch updating this document would be appreciated.

3.2 DOs and DON'Ts for designing and writing Devicetree bindings

This is a list of common review feedback items focused on binding design. With every rule, there are exceptions and bindings have many gray areas.

For guidelines related to patches, see *Submitting Devicetree (DT) binding patches*

3.2.1 Overall design

- DO attempt to make bindings complete even if a driver doesn't support some features. For example, if a device has an interrupt, then include the 'interrupts' property even if the driver is only polled mode.
- DON'T refer to Linux or "device driver" in bindings. Bindings should be based on what the hardware has, not what an OS and driver currently support.
- DO use node names matching the class of the device. Many standard names are defined in the DT Spec. If there isn't one, consider adding it.
- DO check that the example matches the documentation especially after making review changes.
- DON'T create nodes just for the sake of instantiating drivers. Multi-function devices only need child nodes when the child nodes have their own DT resources. A single node can be multiple providers (e.g. clocks and resets).
- DON'T use 'syscon' alone without a specific compatible string. A 'syscon' hardware block should have a compatible string unique enough to infer the register layout of the entire block (at a minimum).

3.2.2 Properties

- DO make 'compatible' properties specific. DON'T use wildcards in compatible strings. DO use fallback compatibles when devices are the same as or a subset of prior implementations. DO add new compatibles in case there are new features or bugs.
- DO use a vendor prefix on device-specific property names. Consider if properties could be common among devices of the same class. Check other existing bindings for similar devices.
- DON'T redefine common properties. Just reference the definition and define constraints specific to the device.
- DO use common property unit suffixes for properties with scientific units. Recommended suffixes are listed at https://github.com/devicetree-org/dt-schema/blob/master/schemas/property-units.yaml
- DO define properties in terms of constraints. How many entries? What are possible values? What is the order?

3.2.3 Typical cases and caveats

- Phandle entries, like clocks/dmas/interrupts/resets, should always be explicitly ordered. Include the {clock,dma,interrupt,reset}-names if there is more than one phandle. When used, both of these fields need the same constraints (e.g. list of items).
- For names used in {clock,dma,interrupt,reset}-names, do not add any suffix, e.g.: "tx" instead of "txirq" (for interrupt).
- Properties without schema types (e.g. without standard suffix or not defined by schema) need the type, even if this is an enum.
- If schema includes other schema (e.g. /schemas/i2c/i2c-controller.yaml) use "unevaluatedProperties:false". In other cases, usually use "additionalProperties:false".
- For sub-blocks/components of bigger device (e.g. SoC blocks) use rather device-based compatible (e.g. SoC-based compatible), instead of custom versioning of that component. For example use "vendor,soc1234-i2c" instead of "vendor,i2c-v2".
- "syscon" is not a generic property. Use vendor and type, e.g. "vendor,power-manager-syscon".

3.2.4 Board/SoC .dts Files

- DO put all MMIO devices under a bus node and not at the top-level.
- DO use non-empty 'ranges' to limit the size of child buses/devices. 64-bit platforms don't need all devices to have 64-bit address and size.

3.3 Writing Devicetree Bindings in json-schema

Devicetree bindings are written using json-schema vocabulary. Schema files are written in a JSON-compatible subset of YAML. YAML is used instead of JSON as it is considered more human readable and has some advantages such as allowing comments (Prefixed with '#').

Also see *Annotated Example Schema*.

3.3.1 Schema Contents

Each schema doc is a structured json-schema which is defined by a set of top-level properties. Generally, there is one binding defined per file. The top-level json-schema properties used are:

$id A json-schema unique identifier string. The string must be a valid URI typically containing the binding's filename and path. For DT schema, it must begin with "http://devicetree.org/schemas/". The URL is used in constructing references to other files specified in schema "$ref" properties. A $ref value with a leading '/' will have the hostname prepended. A $ref value with only a relative path or filename will be prepended with the hostname and path components of the current schema file's '$id' value. A URL is used even for local files, but there may not actually be files present at those locations.

$schema Indicates the meta-schema the schema file adheres to.

title A one-line description on the contents of the binding schema.

maintainers A DT specific property. Contains a list of email address(es) for maintainers of this binding.

description Optional. A multi-line text block containing any detailed information about this binding. It should contain things such as what the block or device does, standards the device conforms to, and links to datasheets for more information.

select Optional. A json-schema used to match nodes for applying the schema. By default, without 'select', nodes are matched against their possible compatible-string values or node name. Most bindings should not need select.

allOf Optional. A list of other schemas to include. This is used to include other schemas the binding conforms to. This may be schemas for a particular class of devices such as I2C or SPI controllers.

properties A set of sub-schema defining all the DT properties for the binding. The exact schema syntax depends on whether properties are known, common properties (e.g. 'interrupts') or are binding/vendor-specific properties.

A property can also define a child DT node with child properties defined under it.

For more details on properties sections, see 'Property Schema' section.

patternProperties Optional. Similar to 'properties', but names are regex.

required A list of DT properties from the 'properties' section that must always be present.

examples Optional. A list of one or more DTS hunks implementing the binding. Note: YAML doesn't allow leading tabs, so spaces must be used instead.

Unless noted otherwise, all properties are required.

3.3.2 Property Schema

The 'properties' section of the schema contains all the DT properties for a binding. Each property contains a set of constraints using json-schema vocabulary for that property. The properties schemas are what are used for validation of DT files.

For common properties, only additional constraints not covered by the common, binding schema need to be defined such as how many values are valid or what possible values are valid.

Vendor-specific properties will typically need more detailed schema. With the exception of boolean properties, they should have a reference to a type in schemas/types.yaml. A "description" property is always required.

The Devicetree schemas don't exactly match the YAML-encoded DT data produced by dtc. They are simplified to make them more compact and avoid a bunch of boilerplate. The tools process the schema files to produce the final schema for validation. There are currently 2 transformations the tools perform.

The default for arrays in json-schema is they are variable-sized and allow more entries than explicitly defined. This can be restricted by defining 'minItems', 'maxItems', and 'additionalItems'. However, for DeviceTree Schemas, a fixed size is desired in most cases, so these properties are added based on the number of entries in an 'items' list.

The YAML Devicetree format also makes all string values an array and scalar values a matrix (in order to define groupings) even when only a single value is present. Single entries in schemas are fixed up to match this encoding.

3.3.3 Coding style

Use YAML coding style (two-space indentation). For DTS examples in the schema, preferred is four-space indentation.

3.3.4 Testing

Dependencies

The DT schema project must be installed in order to validate the DT schema binding documents and validate DTS files using the DT schema. The DT schema project can be installed with pip:

```
pip3 install dtschema
```

Note that 'dtschema' installation requires 'swig' and Python development files installed first. On Debian/Ubuntu systems:

```
apt install swig python3-dev
```

Several executables (dt-doc-validate, dt-mk-schema, dt-validate) will be installed. Ensure they are in your PATH (~/.local/bin by default).

Recommended is also to install yamllint (used by dtschema when present).

Running checks

The DT schema binding documents must be validated using the meta-schema (the schema for the schema) to ensure they are both valid json-schema and valid binding schema. All of the DT binding documents can be validated using the `dt_binding_check` target:

```
make dt_binding_check
```

In order to perform validation of DT source files, use the `dtbs_check` target:

```
make dtbs_check
```

Note that `dtbs_check` will skip any binding schema files with errors. It is necessary to use `dt_binding_check` to get all the validation errors in the binding schema files.

It is possible to run both in a single command:

```
make dt_binding_check dtbs_check
```

It is also possible to run checks with a subset of matching schema files by setting the `DT_SCHEMA_FILES` variable to a specific schema file or pattern.

```
make dt_binding_check DT_SCHEMA_FILES=trivial-devices.yaml
make dt_binding_check DT_SCHEMA_FILES=/gpio/
make dtbs_check DT_SCHEMA_FILES=trivial-devices.yaml
```

3.3.5 json-schema Resources

JSON-Schema Specifications

Using JSON Schema Book

3.3.6 Annotated Example Schema

Also available as a separate file: `example-schema.yaml`

```
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
# Copyright 2018 Linaro Ltd.
%YAML 1.2
---
# All the top-level keys are standard json-schema keywords except for
# 'maintainers' and 'select'

# $id is a unique identifier based on the filename. There may or may not be a
# file present at the URL.
$id: http://devicetree.org/schemas/example-schema.yaml#
# $schema is the meta-schema this schema should be validated with.
$schema: http://devicetree.org/meta-schemas/core.yaml#

title: An example schema annotated with jsonschema details

maintainers:
  - Rob Herring <robh@kernel.org>

description: |
  A more detailed multi-line description of the binding.

  Details about the hardware device and any links to datasheets can go here.

  Literal blocks are marked with the '|' at the beginning. The end is marked by
  indentation less than the first line of the literal block. Lines also cannot
  begin with a tab character.

select: false
  # 'select' is a schema applied to a DT node to determine if this binding
  # schema should be applied to the node. It is optional and by default the
  # possible compatible strings are extracted and used to match.

  # In this case, a 'false' schema will never match.

properties:
  # A dictionary of DT properties for this binding schema
  compatible:
    # More complicated schema can use oneOf (XOR), anyOf (OR), or allOf (AND)
    # to handle different conditions.
    # In this case, it's needed to handle a variable number of values as there
    # isn't another way to express a constraint of the last string value.
```

```
    # The boolean schema must be a list of schemas.
    oneOf:
      - items:
          # items is a list of possible values for the property. The number of
          # values is determined by the number of elements in the list.
          # Order in lists is significant, order in dicts is not
          # Must be one of the 1st enums followed by the 2nd enum
          #
          # Each element in items should be 'enum' or 'const'
          - enum:
              - vendor,soc4-ip
              - vendor,soc3-ip
              - vendor,soc2-ip
          - enum:
              - vendor,soc1-ip
        # additionalItems being false is implied
        # minItems/maxItems equal to 2 is implied
      - items:
          # 'const' is just a special case of an enum with a single possible␣
↪value
          - const: vendor,soc1-ip

  reg:
    # The core schema already checks that reg values are numbers, so device
    # specific schema don't need to do those checks.
    # The description of each element defines the order and implicitly defines
    # the number of reg entries.
    items:
      - description: core registers
      - description: aux registers
    # minItems/maxItems equal to 2 is implied

  reg-names:
    # The core schema enforces this (*-names) is a string array
    items:
      - const: core
      - const: aux

  clocks:
    # Cases that have only a single entry just need to express that with␣
↪maxItems
    maxItems: 1
    description: bus clock. A description is only needed for a single item if
      there's something unique to add.
      The items should have a fixed order, so pattern matching names are
      discouraged.

  clock-names:
    items:
      - const: bus
```

```
interrupts:
  # Either 1 or 2 interrupts can be present
  minItems: 1
  items:
    - description: tx or combined interrupt
    - description: rx interrupt
  description:
    A variable number of interrupts warrants a description of what conditions
    affect the number of interrupts. Otherwise, descriptions on standard
    properties are not necessary.
    The items should have a fixed order, so pattern matching names are
    discouraged.

interrupt-names:
  # minItems must be specified here because the default would be 2
  minItems: 1
  items:
    - const: tx irq
    - const: rx irq

# Property names starting with '#' must be quoted
'#interrupt-cells':
  # A simple case where the value must always be '2'.
  # The core schema handles that this must be a single integer.
  const: 2

interrupt-controller: true
  # The core checks this is a boolean, so just have to list it here to be
  # valid for this binding.

clock-frequency:
  # The type is set in the core schema. Per-device schema only need to set
  # constraints on the possible values.
  minimum: 100
  maximum: 400000
  # The value that should be used if the property is not present
  default: 200

foo-gpios:
  maxItems: 1
  description: A connection of the 'foo' gpio line.

# *-supply is always a single phandle, so nothing more to define.
foo-supply: true

# Vendor-specific properties
#
# Vendor-specific properties have slightly different schema requirements than
# common properties. They must have at least a type definition and
```

```
  # 'description'.
  vendor,int-property:
    description: Vendor-specific properties must have a description
    $ref: /schemas/types.yaml#/definitions/uint32
    enum: [2, 4, 6, 8, 10]

  vendor,bool-property:
    description: Vendor-specific properties must have a description. Boolean
      properties are one case where the json-schema 'type' keyword can be used
      directly.
    type: boolean

  vendor,string-array-property:
    description: Vendor-specific properties should reference a type in the
      core schema.
    $ref: /schemas/types.yaml#/definitions/string-array
    items:
      - enum: [foo, bar]
      - enum: [baz, boo]

  vendor,property-in-standard-units-microvolt:
    description: Vendor-specific properties having a standard unit suffix
      don't need a type.
    enum: [ 100, 200, 300 ]

  vendor,int-array-variable-length-and-constrained-values:
    description: Array might define what type of elements might be used (e.g.
      their range).
    $ref: /schemas/types.yaml#/definitions/uint32-array
    minItems: 2
    maxItems: 3
    items:
      minimum: 0
      maximum: 8

  child-node:
    description: Child nodes are just another property from a json-schema
      perspective.
    type: object  # DT nodes are json objects
    properties:
      vendor,a-child-node-property:
        description: Child node properties have all the same schema
          requirements.
        type: boolean

    required:
      - vendor,a-child-node-property

# Describe the relationship between different properties
dependencies:
```

```
  # 'vendor,bool-property' is only allowed when 'vendor,string-array-property'
  # is present
  vendor,bool-property: [ 'vendor,string-array-property' ]
  # Expressing 2 properties in both orders means all of the set of properties
  # must be present or none of them.
  vendor,string-array-property: [ 'vendor,bool-property' ]

required:
  - compatible
  - reg
  - interrupts
  - interrupt-controller

# if/then schema can be used to handle conditions on a property affecting
# another property. A typical case is a specific 'compatible' value changes the
# constraints on other properties.
#
# For multiple 'if' schema, group them under an 'allOf'.
#
# If the conditionals become too unweldy, then it may be better to just split
# the binding into separate schema documents.
allOf:
  - if:
      properties:
        compatible:
          contains:
            const: vendor,soc2-ip
    then:
      required:
        - foo-supply
    else:
      # If otherwise the property is not allowed:
      properties:
        foo-supply: false
  # Altering schema depending on presence of properties is usually done by
  # dependencies (see above), however some adjustments might require if:
  - if:
      required:
        - vendor,bool-property
    then:
      properties:
        vendor,int-property:
          enum: [2, 4, 6]

# Ideally, the schema should have this line otherwise any other properties
# present are allowed. There's a few common properties such as 'status' and
# 'pinctrl-*' which are added automatically by the tooling.
#
# This can't be used in cases where another schema is referenced
# (i.e. allOf: [{$ref: ...}]).
```

```
# If and only if another schema is referenced and arbitrary children nodes can
# appear, "unevaluatedProperties: false" could be used.  A typical example is
# an I2C controller where no name pattern matching for children can be added.
additionalProperties: false

examples:
  # Examples are now compiled with dtc and validated against the schemas
  #
  # Examples have a default #address-cells and #size-cells value of 1. This can
  # be overridden or an appropriate parent bus node should be shown (such as on
  # i2c buses).
  #
  # Any includes used have to be explicitly included. Use 4-space indentation.
  - |
    node@1000 {
        compatible = "vendor,soc4-ip", "vendor,soc1-ip";
        reg = <0x1000 0x80>,
              <0x3000 0x80>;
        reg-names = "core", "aux";
        interrupts = <10>;
        interrupt-controller;
    };
```

3.4 Submitting Devicetree (DT) binding patches

3.4.1 I. For patch submitters

0) Normal patch submission rules from Documentation/process/submitting-patches.rst applies.

1) The Documentation/ and include/dt-bindings/ portion of the patch should be a separate patch. The preferred subject prefix for binding patches is:

   ```
   "dt-bindings: <binding dir>: ..."
   ```

   The 80 characters of the subject are precious. It is recommended to not use "Documentation" or "doc" because that is implied. All bindings are docs. Repeating "binding" again should also be avoided.

2) DT binding files are written in DT schema format using json-schema vocabulary and YAML file format. The DT binding files must pass validation by running:

   ```
   make dt_binding_check
   ```

   See *Writing Devicetree Bindings in json-schema* for more details about schema and tools setup.

3) DT binding files should be dual licensed. The preferred license tag is (GPL-2.0-only OR BSD-2-Clause).

4) Submit the entire series to the devicetree mailinglist at

devicetree@vger.kernel.org

and Cc: the DT maintainers. Use scripts/get_maintainer.pl to identify all of the DT maintainers.

5) The Documentation/ portion of the patch should come in the series before the code implementing the binding.

6) Any compatible strings used in a chip or board DTS file must be previously documented in the corresponding DT binding text file in Documentation/devicetree/bindings. This rule applies even if the Linux device driver does not yet match on the compatible string. [checkpatch will emit warnings if this step is not followed as of commit bff5da4335256513497cc8c79f9a9d1665e09864 ("checkpatch: add DT compatible string documentation checks").]

7) The wildcard "<chip>" may be used in compatible strings, as in the following example:

   - compatible: Must contain '"nvidia,<chip>-pcie", "nvidia,tegra20-pcie"' where <chip> is tegra30, tegra132, …

   As in the above example, the known values of "<chip>" should be documented if it is used.

8) If a documented compatible string is not yet matched by the driver, the documentation should also include a compatible string that is matched by the driver (as in the "nvidia,tegra20-pcie" example above).

9) Bindings are actively used by multiple projects other than the Linux Kernel, extra care and consideration may need to be taken when making changes to existing bindings.

3.4.2 II. For kernel maintainers

1) If you aren't comfortable reviewing a given binding, reply to it and ask the devicetree maintainers for guidance. This will help them prioritize which ones to review and which ones are ok to let go.

2) For driver (not subsystem) bindings: If you are comfortable with the binding, and it hasn't received an Acked-by from the devicetree maintainers after a few weeks, go ahead and take it.

   For subsystem bindings (anything affecting more than a single device), getting a devicetree maintainer to review it is required.

3) For a series going though multiple trees, the binding patch should be kept with the driver using the binding.

3.4.3 III. Notes

0) Please see *Devicetree (DT) ABI* for details regarding devicetree ABI.

1) This document is intended as a general familiarization with the process as decided at the 2013 Kernel Summit. When in doubt, the current word of the devicetree maintainers overrules this document. In that situation, a patch updating this document would be appreciated.

INDEX