Linux Devicetree Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

LINUX AND THE DEVICE TREE

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

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] https://elinux.org/Device_Tree_Usage

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.

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.

1.2 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….

1.3 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.

1.4 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.

1.5 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.

1.6 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";
```

(continues on next page)

(continued from previous page)

```
                #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]

1.7 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.

CHAPTER

TWO

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 '#').

2.1 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 $ref value a relative path or filename only will be prepended with the hostname and path components of the current schema file's '$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.

2.2 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 is 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.

2.3 Testing

2.3.1 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 git+https://github.com/devicetree-org/dt-schema.
↪git@master
```

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

dtc must also be built with YAML output support enabled. This requires that libyaml and its headers be installed on the host system. For some distributions that involves installing the development package, such as:

Debian:

```
apt-get install libyaml-dev
```

Fedora:

```
dnf -y install libyaml-devel
```

2.3.2 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 single schema file by setting the DT_SCHEMA_FILES variable to a specific schema file.

```
make dt_binding_check DT_SCHEMA_FILES=Documentation/devicetree/
↪bindings/trivial-devices.yaml
make dtbs_check DT_SCHEMA_FILES=Documentation/devicetree/bindings/
↪trivial-devices.yaml
```

2.4 json-schema Resources

JSON-Schema Specifications

Using JSON Schema Book

CHAPTER THREE

DT CHANGESETS

A DT 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().

CHAPTER

FOUR

DEVICE TREE DYNAMIC RESOLVER NOTES

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

4.1 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.

CHAPTER

FIVE

OPEN FIRMWARE DEVICE TREE UNITTEST

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

5.1 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 Device Tree*

(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.

5.2 2. 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.

5.2.1 2.1. 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
```

(continues on next page)

(continued from previous page)

```
                           |
                          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
```

(continues on next page)

(continued from previous page)

```
                |          |           |            |
                |         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().

5.2.2 2.2. 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 :)

DEVICE TREE 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 *Device Tree Dynamic Resolver Notes*[1]

6.1 How overlays work

A Device Tree' 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 --------------------
↪--------
```

(continues on next page)

(continued from previous page)

```
    /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 -------------------------------------------------------
↪--------
```

6.2 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.

In addition, 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.

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.

CHAPTER

SEVEN

DEVICE TREE

7.1 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.

7.2 Submitting devicetree (DT) binding patches

7.2.1 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-schema.rst 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).

7.2.2 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.

   Subsystem bindings (anything affecting more than a single device) then 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.

7.2.3 III. Notes

0) Please see …bindings/ABI.txt 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.

7.3 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*

7.3.1 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).

7.3.2 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. See property-units.txt.
- DO define properties in terms of constraints. How many entries? What are possible values? What is the order?

7.3.3 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.