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].

[1] (*1*,*2*) https://www.devicetree.org/specifications/

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. 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.

2. Data Model

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

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.

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.

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.

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]

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.