Device Trees

A Database Approach to Describing Hardware

Class ESC-xxx
Tuesday, April 1, 3:30 pm

Doug Abbott
Silver City, NM
doug@intellimetrix.us

Intellimetrix
COMPUTING FOR SCIENCE AND INDUSTRY

www.intellimetrix.us

Abstract

When moving an operating system to a new platform, one of the major headaches is describing the hardware. Where are the peripheral registers located? What interrupts are the devices connected to? How are devices interconnected? Historically, these issues have been addressed by modifying the device driver code. The result is a unique kernel image for each platform.

Suppose we could describe the hardware in a database that could be passed to the kernel at boot time? The kernel and its driver code are now generic and can be easily adapted to new platforms. Device trees are a database mechanism for describing a system's hardware independent of the operating system and its device drivers. Hardware is described in plain text and translated to a binary "blob" by the Device Tree Compiler.

Introduction – the Problem

One of the biggest problems with porting an operating system such as Linux to a new platform is describing the hardware. That's because the hardware description is scattered over a dozen or so device drivers, the kernel, and the boot loader just to name a few. The ultimate result is that these various pieces of software become unique to each platform and the number of configuration options grows.

There have been a number of approaches to addressing this problem. The notion of a "board support package" or BSP attempts to gather all of the hardware dependent code in a few files in one place. It could be argued that the entire arch/ subtree of the Linux kernel source tree is a gigantic board support package.

Take a look at the arch/arm/ subtree. In there you'll find a large number of directories of the form mach-* and plat-*, presumably short for "machine" and "platform" respectively. Most of the files in these directories provide configuration information for a specific implementation of the ARM architecture. And of course, each implementation describes its configuration differently.

Wouldn't it be nice to have a single language that could be used to unambiguously describe the hardware of a computer system? That's the premise, and promise, of device trees.

Background

The device tree concept evolved in the Open Firmware project that became IEEE 1275. The IEEE subsequently withdrew support for the standard and the project ultimately merged into UEFI, the Unified Extensible Firmware Interface. Nevertheless, Open Firmware became the de facto method of booting Power PCs, including Apple MacIntoshes.

When the 32-bit and 64-bit Power PC arch/ trees in the Linux kernel were merged, a decision was made to require device trees as the only mechanism for describing hardware. The idea has since spread to several other architectures supported by Linux including ARM.

Platform devices vs. "discoverable" devices

The peripheral devices in a system can be characterized along a number of dimensions. There are, for example, character vs. block devices. There are memory mapped devices and those that connect through an external bus such as I2C or USB. Then there are *platform* devices and *discoverable* devices.

Discoverable devices are those that live on external busses such as PCI and USB that can tell the system what they are and how they are configured. That is, they can be "discovered" by the kernel. Having identified a device, it's a fairly simple matter to load the corresponding driver, which then interrogates the device to determine its precise configuration.

Platform devices, on the other hand, lack any mechanism to identify themselves. SoC (System on Chip) implementations are rife with these platform devices—system clocks, interrupt controllers, GPIO, serial ports, to name a few. The device tree mechanism is particularly useful for managing platform devices.

Basics

A device tree is a hierarchical data structure that describes the collection of devices and interconnecting busses of a computer system. It is organized as nodes that begin at a root represented by "/". Every node has a name and consists of "properties" that are name-value pairs. It may also contain "child" nodes.

Figure 1 is a sample device tree taken from the devicetree.org website. It doesn't do anything beyond illustrating the structure. Here we have two nodes named node1 and node2. node1 has two child nodes and node2 has one child. Properties are represented by name=value. Values can be strings, lists of strings, one or more numeric values enclosed by square brackets, or one or more "cells" enclosed in angle brackets. The value can also be empty if the property conveys a Boolean value by its presence or absence.

```
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
};
```

Figure 1: Sample Device Tree

Figure 2 shows how elements of a real machine are represented in the device tree. Let's start with the compatible property, which every node is required to have. The value of compatible is a list of strings. The first string exactly identifies the device by both manufacturer and model name. Any subsequent strings identify what the node might be "compatible" with. For example, many serial ports implement the National Semiconductor register interface and could be specified as compatible with the "ns16550".

#address-cells and #size-cells say how many 32-bit cells are required to specify a node's starting address and range respectively. interrupt_parent points to a node that identifies the interrupt controller on this system.

This brings us to the first child node named cpus. In here we find two child nodes, each describing a CPU core. Note the naming. Most nodes have a name of the form <name>@<unit_address> where <unit_address> is specified by the reg property and represents the beginning of the node's addressable range, whatever that is. CPUs don't have a range, so #size_cells is set to zero and we simply identify the CPUs by an index number. In this case, the system has two Cortex-A9 cores.

```
/ {
        compatible = ”acme, my_board”
        #address-cells = <1>;
        #size-cells = <1>;
        interrupt_parent = <&intc>;

        cpus {
                #size-cells = <0>;
                cpu@0 {
                        compatible = “arm, cortex-a9;
                        reg = <0>;
                };
                cpu@1 {
                        compatible = “arm, cortex-a9;
                        reg = <1>;
                };
        };
        serial@101f0000 {                       //memory mapped device
                compatible = "arm, pl011";
                reg = <0x101f0000 0x1000 >;
                interrupts = <2 0>
         };
        intc: interrupt_controller@10140000 {
                compatible = “arm, pl190”;
                reg = <0x10140000 0x1000 >;
                interrupt-controller;
                #interrupt-cells = <2>;
        };
        external_bus {
                #address_cells = <2>
                #size_cells = <1>

                i2c@1,0 {
                        compatible = “acme, a1234-i2c-bus”;
                        #address_cells = <1>;
                        #size_cells = <0>;
                        reg = <1 0 0x1000>;
                        interrupts = <6 2>;
                        rtc@58 {
                                compatible = “maxim, ds1338”;
                                reg = <58>;
                        };
                };
        };
};
```

Figure 2: A “real” device tree

Next up is the description of a serial port at address 0x101f0000. The reg property tells us it uses 0x1000 bytes. The reg property requires at least #address_cells plus #size_cells cells. Multiple ranges are specified by adding more cells.

This node also indicates that this device uses an interrupt at <2 0>. We don't necessarily know what that means until we encounter the description of the interrupt controller and see that the definition of an interrupt requires two cells.

The compatible property is what makes the connection between an entry in the device tree and a device driver that can manage the device. Most contemporary device drivers announce what they are capable of by registering some sort of "device table". For the serial device in particular, the driver would have code like this:

```
struct of_device_id my_of_match[] {
        {.compatible = "arm,pl011"}
}

MODULE_DEVICE_TABLE (of, my_of_match);
```

The MODULE_DEVICE_TABLE() macro makes the id table externally visible so the kernel can scan it to learn that this driver handles the "arm,pl011" device.

The interrupt controller node has a label that allows us to reference it from elsewhere in the tree. It has a Boolean property, interrupt-controller, that says this device receives interrupts. Note that it takes two cells to specify an interrupt. The interrupts property of other nodes then specifies which interrupt the device is attached to.

Some devices aren't memory mapped but are instead connected to some external "bus". In this example it requires two cells to specify the address, in this case a chip select and an offset from the base of the chip select. The size is still one cell.

One of the devices attached to the bus is an I2C controller. The children of this node are I2C devices, which don't have a size, only a device number.

Special Nodes

There are a couple of "special" nodes that perform functions other than describing hardware. The *aliases* node provides a way to represent the full path to a node by a shorter string as shown here:

```
aliases {
        rtc = "/external_bus/i2c@1,0/rtc@58";
};
```

The *chosen* node provides a way to pass environment data to the kernel. It strikes me as an odd name, but there it is. Here's an example:

```
chosen {
        bootargs = "rootfs=/dev/nfs rw nfsroot=192.168.1.50 console=ttySAC0,115200";
};
```

The Flattened Device Tree and the Device Tree Compiler

The original Open Firmware specification uses the text-based device tree directly. That's fine in desktops and servers, but could become problematic in more resource-constrained embedded systems. That led to the development of the *flattened device tree*, a more compact, binary representation. This is what Linux operates on.

The Device Tree Compiler, or DTC, converts the textual representation to the flattened device tree, called a device tree "blob". The DTC is part of the kernel source tree and may be invoked by an appropriate make command. For example, the command:

```
make <board_name>.dtb
```

will find the source file <board_name>.dts in arch/<arch>/boot/dts and run it through the DTC to create <board_name>.dtb. Note that this only works if you've built the kernel with device tree support. To enable device tree support (in ARM kernels at least), select **Flattened Device Tree support** under **Boot options** in the kernel configuration menu.

Device Trees and the boot loader

Having created a .dtb file, we need to somehow pass it to the boot loader, which in turn passes it to the kernel. How that happens is a function of the specific boot loader. Let's use u-boot, arguably the most popular boot loader for embedded systems, as an example. u-boot has a configuration option, CONFIG_OF_LIBFDT, that adds support for flattened device tables including an fdt command. The fdt command lets you do things like list the device tree, make new nodes, set properties, and remove nodes or properties.

So where does the .dtb file get stored? Depends on how your system is organized. Many u-boot-based systems store their boot loader, boot loader environment, OS image, and root file system in NAND flash. Each of these elements is stored in an arbitrarily sized *partition* in NAND flash. It would be easy enough to create another partition and store the .dtb there.

Another thing that turning on CONFIG_OF_LIBFDT does is to modify the bootm command to accept up to three addresses:

```
bootm <kernel_addr> <initrd_addr> <dtb_addr>
```

The strategy then is to copy the kernel image, initrd image[1], and device tree blob from their respective NAND partitions into RAM and then invoke the bootm command. Each architecture has its own convention for how the address of the device tree blob is passed to the kernel. For ARM, the convention is to pass the DTB address in R2.

Using the Device Table

Once the DTB has been read by the kernel, there is a rich set of kernel APIs for accessing nodes and properties. These APIs are declared in of.h and are used by device drivers to discover their respective devices. The corresponding source code is found in linux/drivers/of.

References

devicetree.org – home page for the device tree project. Among other things, this site documents bindings that aren't covered by the Linux kernel or the ePAPR (see next reference). There's also a good tutorial on device tree usage.

Standard for Embedded Power Architecture™ Platform Requirements ePAPR – available at www.power.org. Probably the most complete definition of the device tree.

Device Trees Everywhere, David Gibson, Benjamin Herrenschmidt. ozlabs.org/~dgibson/papers/dtc-paper.pdf

A Symphony of Flavours: Using the device tree to describe embedded hardware, Grant Likely, Josh Boyer. https://www.kernel.org/doc/ols/2008/ols2008v2-pages-27-38.pdf

xillybus.com/tutorials/device-tree-zynq-1 – Interesting 5-part tutorial on device trees.

[1] In my experience I've never encountered an initial RAM disk image in an embedded system and so this argument is passed as "-".