Device Tree

From WikiPedia

In computing, a **devicetree** (also written **device tree**) is a data structure describing the hardware components of a particular computer so that the operating system's kernel can use and manage those components, including the CPU or CPUs, the memory, the buses and the integrated peripherals.

The device tree was derived from SPARC-based computers via the Open Firmware project. The current Devicetree specification[1] is targeted at smaller systems, but is still used with some server-class systems (for instance, those described by the Power Architecture Platform Reference).

Personal computers with the x86 architecture generally do not use device trees, relying instead on various auto configuration protocols (e.g. ACPI) to discover hardware. Systems which use device trees usually pass a static device tree (perhaps stored in ROM) to the operating system, but can also generate a device tree in the early stages of booting. As an example, Das U-Boot and kexec can pass a device tree when launching a new operating system. On systems with a boot loader that does not support device trees, a static device tree may be installed along with the operating system; the Linux kernel supports this approach.

The Devicetree specification is currently managed by a community named devicetree.org, which is associated with, among others, Linaro and Arm

Starting early in 2020 the CoreBoot team investigated various OSS alternatives for device configuration, with an eye towards replacing all of the internal proprietary solutions, e.g. DevCfg and DAL. This initiative was driven by the technical requirements of the Netrani architecture that required changes in the code being delivered to OEMs. Specifically, XBL code would no longer be delivered in source form and the OEM would no longer be able to modify/build this code. A mechanism was required to allow the OEM to customize the existing drivers, which led to the decision to embed DeviceTree support within XBL/UEFI and Q6 code. It is anticipated that other targets, e.g. Hypervisor, will embrace DeviceTree in the future.

This vtechstudy is intended to introduce you to general DeviceTree topics. To complete all of the quiz questions, please do the following preparation:

- Join qgroup: QDTE.qpm.viewer
    - Using QPM, install QDTE on your local laptop
- If using Linux workstation, install dtc from upstream OSS project
    - git clone git://git.kernel.org/pub/scm/utils/dtc/dtc.git
- If using Windows:
    - Join qgroup: dtc.qpm.viewer
    - Install DTC from QPM
- Pull down the examples for use with this vtechstudy:
    - git clone https://github.qualcomm.com/mturney/dt-vtechstudy.git

Most information you will find about DeviceTree online or in books is usually presented in the context of the Linux kernel and device drivers. You want to be aware of this as you read through the literature, especially when the discussion includes information about how the kernel binds drivers to a DTB node. Information like kernel bindings is not DeviceTree specific and may have no bearing on how QC is using DeviceTree in the XBL/UEFI/Q6 boot-stack. That said, it is almost impossible to separate DeviceTree from the Linux kernel, consider that the upstream OSS project is housed on kernel.org, as the URL above for pulling dtc indicates. Many design choices implemented by the upstream project derive directly from Linux kernel requirements. The most obvious example, because it has a direct impact on QC usage is that the Linux kernel team bends over backwards to prevent the build from breaking. The dtc compiler only fails and halts the build on syntax errors and throws warnings when a semantic error is detected. There is no command-line argument to dtc to force semantic warnings to be treated as errors by returning a non-zero return code. This is one of the two reasons QC created a wrapper-script (gendtb.py) that boot build systems use to invoke dtc. QC wants every warning to be treated as an error and we prefer not to modify the dtc source to accomplish this. The second reason for the gendtb.py wrapper is to easily mimic the way the kernel processes .dts files. The kernel first uses the C pre-processor on the .dts file and then runs the output through dtc, this allows the use of simple #define macros and #include facilities provided by the C pre-processor.

Online Reference Materials

1. go/safari
    a. Mastering Embedded Linux Programming : Chapter 3 – All About Bootloaders
    b. Linux Device Driver Development – Second Edition : Chapter 5 – Understanding and Leveraging the Device Tree
2. Devicetree Specification v0.3
3. NXP App Note: Introduction to Device Trees
4. Device Tree Mysteries
5. Device Tree Usage
6. Device Tree Source Format
7. Device Tree Dynamic Object Format Internals
8. QC Presentations
    a. Tech Talk – Device Tree in XBL (4-2-21)
9. Online Videos
    a. https://www.youtube.com/watch?v=xamjHjjyeBI (from 2013, ~50 minutes)
    b. https://www.youtube.com/watch?v=Nz6aBffv-Ek (from 2020, ~43 minutes)