A Tutorial on the Device Tree (Zynq) -- Part I

Who is this tutorial for?

This tutorial was written with Xilinx' Zynq-7000 EPP device in mind (an ARM Cortex-A9 combined with FPGA), but the general concepts apply for any Linux kernel using the device tree. The examples assume that the Xillinux distribution for the Zedboard is used.

What's the device tree good for?

Picture this: The bootloader has just copied the Linux kernel into the processor's SDRAM. It then jumps to the kernel's entry point. The kernel is now just like any bare-metal application running on a processor. It needs to configure the processor. It needs to set up virtual memory. It needs to print something to the console. But how? All these operations are carried out by writing to registers, but how does the Linux kernel know their addresses? How does it know how many cores it can run on? How much memory it can access?

The straightforward solution is platform-specific boot routines in the kernel's sources, which are enabled by kernel configuration parameters. This is fine for everything that is usually fixed, such as the internal registers on an x86 processor, or the access of the BIOS on a PC. But when it comes to things that tend to change, for example the PCI/PCIe peripherals on a PC computer, it's desirable to let the kernel learn about them in run-time.

The ARM architecture has become a major headache in the Linux community: Even though the processors share the same compiler and many functionalities, each embodiment (i.e. chip) has its own addresses for the registers, and a slightly different configuration. On top of that, each board has its own set of external components. The result is a wild forest of header files, patches and special configuration parameters in the kernel tree, each combination matching a specific board with a specific chip containing an ARM processor. In short, it has turned out to be an ugly and unmaintainable pile of hacks which nobody is really fond of.

On top of that, each kernel binary is compiled for a specific chip on a specific board, which is more or less like compiling the kernel for each PC motherboard on the market. So there was a wish to compile the kernel for all ARM processors, and let the kernel somehow detect its hardware and apply the right drivers as needed. Exactly as it does with a PC.

But how? On a PC, the initial registers are hardcoded, and the rest of the information is supplied by the BIOS. So it's easy to auto-detect your hardware when another piece of software tells you what you have. ARM processors don't have a BIOS. The Linux kernel has only itself to trust.

So the chosen solution was a **device tree**, also referred to as **Open Firmware** (abbreviated **OF**) or **Flattened Device Tree** (**FDT**). This is essentially a data structure in byte code format (that is, not human-readable) which contains information that is helpful to the kernel when booting up. The boot loader copies that chunk of data into a known address in the RAM before jumping to the kernel's entry point.

I defined the device tree somewhat vaguely, but it's exactly how things are: Even though there are strict conventions (which isn't always followed completely), there is no rigid rule for what can go into the device tree and where it must be put. Any routine in the kernel may look up any parameter in any path in the device tree. It's the choice of the programmer what is parametrized, and where the parameter is best placed in the tree.

Adopting the standard tree structure allows using a convenient API for fetching specific data. For example, there is a clear and cut convention for how to define peripherals on the bus, and an API for getting the essential information the driver needs: Addresses, interrupts and custom variables. More about that later.

To most of us, the device tree is where we inform the kernel about a specific piece of hardware (i.e. PL logic) we've added or removed, so that the kernel can kick off the right driver to handle it (or

Related pages

- A Tutorial on the Device Tree (Zynq) -- Part II
- A Tutorial on the Device Tree (Zynq) -- Part III
- A Tutorial on the Device Tree (Zynq) -- Part IV
- A Tutorial on the Device Tree (Zynq) -- Part V
- Setting up a device tree entry on Altera's SoC FPGAs

refrain from doing so, if the hardware was removed). This is also where specific information about the hardware is conveyed.

Compiling the device tree

The device tree comes in three forms:

- A text file (*.dts) — "source"
- A binary blob (*.dtb) — "object code"
- A file system in a running Linux' /proc/device-tree directory — "debug and reverse engineering information"

In a normal flow, the DTS file is edited and compiled into a DTB file using a special compiler which comes with the Linux kernel sources. On a Xillinux distribution, it's available at /usr/src/kernels/3.3.0-xillinux-1.0+/scripts/dtc/ (or similar).

The device tree compiler can be downloaded and built separately with

```
$ git clone git://www.jdl.com/software/dtc.git dtc
$ cd dtc
$ make
```

but I'll assume below that the kernel source's dtc is used.

The syntax of the device tree's language is described here. Note that this language doesn't execute anything, but like XML, it's just a syntax to organize data. Some architectures have an automatic tool for generating a device tree from an XPS project (e.g. Microblaze), but currently there is no such tool available for the Zynq EPP platform.

The compilation from DTS to DTB is done by changing directory to the Linux kernel source tree's root. On Xillinux 1.0 running on the Zedboard it's

```
$ cd /usr/src/kernels/3.3.0-xillinux-1.0+/
```

and going

```
$ scripts/dtc/dtc -I dts -O dtb -o /path/to/my-tree.dtb /path/to/my-tree.dts
```

which creates the blob file my-tree.dtb. The dtc compiler is a binary application, which is compiled to run on the host's platform (i.e. it's not cross compiled). If the kernel hasn't been compiled on the host, there's a need to at least compile the DTS compiler: First set up a configuration for the kernel. It doesn't matter much anyhow, so copy any related configuration file to .config in the kernel tree's root directory. Or, if this happens to work:

```
$ make ARCH=arm digilent_zed_defconfig
```

And then generate the DTS compiler:

```
$ make ARCH=arm scripts
```

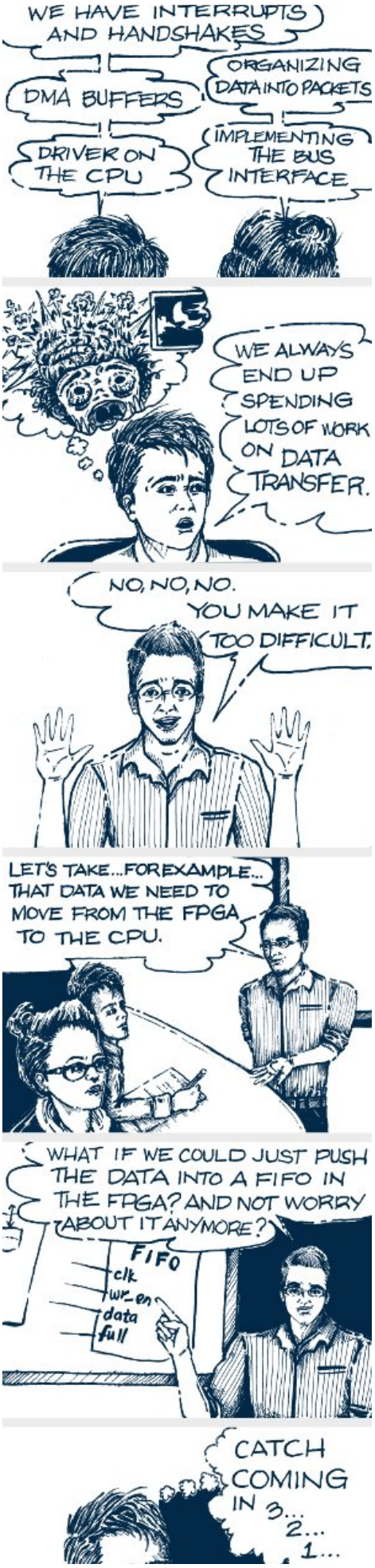
If the path to the cross compiler hasn't been set, this will end with an error. This doesn't matter if the dtc compiler was generated before this error, which is usually the case. If it said "HOSTLD scripts/dtc/dtc" somewhere after the "make" command, it's good enough. Or just try to run dtc as shown above.

Reverse compilation is also possible, either from a DTB file or a /proc/device-tree file system. To obtain a text file from a DTB blob, go something like

```
$ scripts/dtc/dtc -I dtb -O dts -o /path/to/fromdtb.dts /path/to/booted_with_this.dt
```

The DTS file is fine for compilation back to a DTB, but it's better to work with original DTS files, since references made by labels in the original DTS appear as numbers in the reverse-compiled DTS.

The device tree in effect for a running kernel can be obtained in DTS format with



```
# cd /usr/src/kernels/3.3.0-xillinux-1.0+/
# scripts/dtc/dtc -I fs -O dts -o ~/effective.dts /proc/device-tree/
```

This should be done on Xillinux running on Zedboard (or any other distribution that supplies the kernel headers). The output file goes to the home directory.

Continue to part II, which explains the device tree's structure

Further reading

- A Tutorial on the Device Tree (Zynq) -- Part II
- A Tutorial on the Device Tree (Zynq) -- Part III
- A Tutorial on the Device Tree (Zynq) -- Part IV
- A Tutorial on the Device Tree (Zynq) -- Part V
- Setting up a device tree entry on Altera's SoC FPGAs