part 1

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 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.dtb
```

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.

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

The structure of a device tree

A device tree for Linux running on Zynq typically has the following form.

```
/dts-v1/;

/ {

  #address-cells = <1>;

  #size-cells = <1>;

  compatible = "xlnx,zynq-zed";

  interrupt-parent = <&gic>;

  model = "Xillinux for Zedboard";

  aliases {
    serial0 = &ps7_uart_1;
  } ;
  chosen {
    bootargs = "consoleblank=0 root=/dev/mmcblk0p2 rw
rootwait earlyprintk";
    linux,stdout-path = "/axi@0/uart@E0001000";
  };

  cpus {

      [ ... CPU definitions ... ]

   } ;
  ps7_ddr_0: memory@0 {
    device_type = "memory";
    reg = < 0x0 0x20000000 >;
  } ;
  ps7_axi_interconnect_0: axi@0 {
    #address-cells = <1>;
    #size-cells = <1>;
```

```
    compatible = "xlnx,ps7-axi-interconnect-1.00.a",
"simple-bus";
    ranges ;

      [ ... Peripheral definitions ... ]

  } ;
} ;
```

This is the device tree used with Xillinux, with two parts cut out: The one describing the CPUs (because it isn't interesting) and the one defining the peripherals (because it's long, and we'll get down to its details later on).

The source of the device tree used by default is available as e.g. /boot/devicetree-3.3.0-xillinux-1.0.dts in Xillinux' file system.

After the version declaration, the device tree starts with a slash, saying "this is the tree's root", and then there are assignments within curly brackets. From the DTS compiler's point of view, these curly brackets enclose deeper hierarchies in the tree (think directories in a file system). It will be the kernel code's job to walk down this tree, and grab the desired information from certain paths (like reading from a file after reaching the desired path in a file system, if you like).

It's important to remember that the tree's structure is built just the way the kernel expects to find it. The assignments have no specific meaning to the compiler. In fact, many assignments are ignored by the kernel, just like a file in a file system is ignored if no program cares to open it.

Accessing the data from user space

The comparison with a file system isn't only natural, it's also implemented in the kernel's /proc/device-tree: Each curly bracket is represented as a directory having the name of the string coming just before it.

So this is session on a system using the device tree listed above:

```
# hexdump -C '/proc/device-tree/#size-cells'

00000000  00 00 00 01
|....|

00000004

# hexdump -C '/proc/device-tree/axi@0/compatible'

00000000  78 6c 6e 78 2c 70 73 37  2d 61 78 69 2d 69 6e 74
|xlnx,ps7-axi-int|

00000010  65 72 63 6f 6e 6e 65 63  74 2d 31 2e 30 30 2e 61
|erconnect-1.00.a|

00000020  00 73 69 6d 70 6c 65 2d  62 75 73 00
|.simple-bus.|

0000002c
```

or simply

```
# cat '/proc/device-tree/axi@0/compatible'

xlnx,ps7-axi-interconnect-1.00.asimple-bus
```

Note the axi@0 element's definition in the device tree listing above. It says "ps7_axi_interconnect_0: axi@0". The string before the colon is the label, which is possibly referred to within the DTS file, but doesn't appear in the DTB. As just said, It's the string close to the curly brackets that defines the name of the hierarchy (and hence also the directory).

As this example demonstrates, the assignments turn into plain files in the /proc filesystem. If there is no assignment (e.g. "ranges" under "axi@0"), an empty file is created.

What the examples above show, is that the device tree can be used conveniently to convey information to user space programs as well as code within the Linux kernel, as the

/proc/device-tree pseudo-filesystem makes this information accessible. Needless to say, there is a dedicated API in the kernel for accessing the device tree's structure and data.

And you may have noted that the integer is represented in Big Endian. The Zynq's processor runs Little Endian. Just a small thing to keep in mind.

Boot parameters in the device tree

There are three sources for the kernel boot command line in general:

- Those given as CONFIG_CMDLINE in the kernel configuration
- Those passed on by the boot loader (typically U-boot on ARM processors, LILO or GRUB on x86)
- Those included in the device tree, under chosen/bootargs (see listing above)

Which one is used depends on kernel configuration parameters. In Xillinux, the device tree's chosen/bootargs is used.

The chosen UART for kernel boot messages is hardcoded in the initialization routine. As a matter of fact, boot messages will appear on the UART even if the ps7_uart_1: serial@e0001000 entry in the device tree is deleted altogether (but the UART won't be available as /dev/ttyPS0).

The somewhat misleading "aliases" and "linux,stdout-path" assignments are leftovers from other architectures, and have no significance at this time.

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

Defining peripherals

It's likely that you're reading this because you want to write a Linux driver for your own peripheral. The recommended book for learning the basics is the famous Linux Device Drivers. But before jumping into writing a device driver of your own, allow me to share rule number one for writing drivers for Linux: **Never write a device driver for Linux**. Rather, find a well-maintained driver for some other hardware with similar functionality, and hack it. This is not just easier, but you're likely to avoid problems you're not even aware of. Copying snippets of code from other drivers will make your own understandable to others, portable, and with a better chance to be accepted into the kernel tree.

So the key is understanding what another driver is doing, and then adjust the parts that are related. In case of doubt, do what everyone else is doing. Creativity and personal style are not helpful.

Now back to the device tree. Let's look at the segment that was omitted an in part II:

```
 ps7_axi_interconnect_0: axi@0 {
   #address-cells = <1>;
   #size-cells = <1>;
   compatible = "xlnx,ps7-axi-interconnect-1.00.a",
"simple-bus";
   ranges ;
   gic: interrupt-controller@f8f01000 {
     #interrupt-cells = < 3 >;
     compatible = "arm,cortex-a9-gic";
     interrupt-controller ;
     reg = < 0xf8f01000 0x1000  >,< 0xf8f00100 0x100  >;
   } ;
```

```
  pl310: pl310-controller@f8f02000 {
    arm,data-latency = < 3 2 2 >;
    arm,tag-latency = < 2 2 2 >;
    cache-level = < 2 >;
    cache-unified ;
    compatible = "arm,pl310-cache";
    interrupts = < 0 34 4 >;
    reg = < 0xf8f02000 0x1000 >;
  } ;

    [ ... more items ... ]

  xillybus_0: xillybus@50000000 {
    compatible = "xlnx,xillybus-1.00.a";
    reg = < 0x50000000 0x1000 >;
    interrupts = < 0 59 1 >;
    interrupt-parent = <&gic>;
    xlnx,max-burst-len = <0x10>;
    xlnx,native-data-width = <0x20>;
    xlnx,slv-awidth = <0x20>;
    xlnx,slv-dwidth = <0x20>;
    xlnx,use-wstrb = <0x1>;
  } ;
} ;
```

Only the first two devices from the original file are shown, and also the last one, which we'll focus on. As mentioned earlier, this is an excerpt from the full DTS file which is available as /boot/devicetree-3.3.0-xillinux-1.0.dts in Xillinux' file system.

Let's pay attention to the first entry in the list: It's the Zynq processor's interrupt controller. The existence of this entry makes sure that the interrupt controller's driver is loaded. Note that its label is "gic". This label will be referenced in every device that uses interrupts.

We are now finally in position to talk about the interesting stuff: How all this works together with the Linux code.

Relation to the kernel driver

Four things must happen to have our device driver alive and kicking:

- The driver loaded by the kernel when the hardware is present (i.e. declared in the device tree)
- The driver needs to know the physical addresses allocated to the device
- The driver needs to know which interrupt(s) the device will trigger, so it can register interrupt handlers
- Application-specific information needs to be retrieved

The kernel has an API for accessing the device tree directly, but it's much easier to use the dedicated interface for device drivers, which is highly influenced by the API used for PCI/PCIe drivers. Let's consider the xillybus_0 entry, which is rather typical for custom logic attached to the AXI bus.

The label and node name

First, the label ("xillybus") and entry's name ("xillybus@50000000"). The label could have been omitted altogether, and the entry's node name should stick to this format (some-name@address), so that a consistent entry is generated in /sys (/sys/devices/axi.0/50000000.xillybus/ in this case). The data in this device tree entry will appear in /proc/device-tree/axi@0/xillybus@50000000/, but that's definitely not the way to access it from within the kernel.

Making the driver autoload

The first assignment in the node, compatible = “xlnx,xillybus-1.00.a” is the most crucial one: It’s the link between the hardware and its driver. When the kernel scans the entries in the bus for devices (i.e. nodes in the device tree under a bus node) it retrieves the “compatible” properties and compares those strings with a list of strings it “knows about”. This happens automatically on two occasions during the boot process:

- When the kernel starts up, it kicks off compiled-in drivers that match “compatible” entries it finds in the device tree.
- At a later stage (when /lib/modules is available), all kernel modules that match “compatible” entries in the device tree are loaded.

The connection between a kernel driver and the “compatible” entries it should be attached to, is made by a code segment as follows in the driver’s source code:

```
static struct of_device_id xillybus_of_match[]
__devinitdata = {

  { .compatible = "xlnx,xillybus-1.00.a", },

  {}

};



MODULE_DEVICE_TABLE(of, xillybus_of_match);
```

This hocus-pocus code declares that the current driver matches a certain “compatible” entry, and leaves the rest to the kernel infrastructure. Note that there’s a NULL struct entry in the list: It’s possible to define multiple “compatible” strings, all of which will cause the current driver to load, so this list must be terminated with a NULL (“{}” in the list, as shown above).

Also, near the bottom of the code, something like this is necessary:

```
static struct platform_driver xillybus_platform_driver =
{
  .probe = xilly_drv_probe,
  .remove = xilly_drv_remove,
  .driver = {
    .name = "xillybus",
    .owner = THIS_MODULE,
    .of_match_table = xillybus_of_match,
  },
};
```

And then platform_driver_register(&xillybus_platform_driver) must be called in the module's initialization function. This informs the kernel about the function to be called if hardware matching xillybus_of_match has been found, which is xilly_drv_probe() in this case.

To the kernel, the "compatible" string is just a string that needs to be equal (as in strcmp) to what some driver has declared. The "xlnx," prefix is just a way to keep clear of name clashes.

By the way, a peripheral entry in the device tree may declare several "compatible" strings. Also, it's possible that more than one driver will be eligible for a certain peripheral entry, in which case they are all probed until one of them returns success on the probing function. Which one is given the opportunity first is not defined.

It's also possible to require a match with the hardware's name and type, but this is not used often.

An important thing to note when writing kernel modules, is that the automatic loading mechanism (modprobe, actually) depends on an entry for the "compatible" string in /lib/modules/{kernel version}/modules.ofmap and other definition files in the same directory. The correct way to make this happen for your own module, is to copy the *.ko file to somewhere under the relevant /lib/modules/{kernel version}/kernel/drivers/ directory and go

```
# depmod -a
```

on the target platform, with that certain kernel version loaded (or define which kernel version to depmod).

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

Getting the resources

Having the kernel module driver loaded, it's time to get control of the hardware's resources. That is, being able to read and write to the registers, and receiving its interrupts.

Still on the same entry in the device tree,

```
    xillybus_0: xillybus@50000000 {

      compatible = "xlnx,xillybus-1.00.a";

      reg = < 0x50000000 0x1000 >;
      interrupts = < 0 59 1 >;
      interrupt-parent = <&gic>;

      xlnx,max-burst-len = <0x10>;
      xlnx,native-data-width = <0x20>;
```

```
    xlnx,slv-awidth = <0x20>;
    xlnx,slv-dwidth = <0x20>;
    xlnx,use-wstrb = <0x1>;
  } ;
```

we now focus on the part marked in bold.

The driver typically takes ownership of the hardware's memory segment in the probing function (which is the one that is pointed to in the "probe" entry of the platform_driver structure, declared for the driver, e.g. xilly_drv_probe() for Xillybus' driver).

We'll look at the skeleton of a typical probing function (please don't copy from this code, but rather from a real, working driver):

```
static int __devinit xilly_drv_probe(struct
platform_device *op)

{

  const struct of_device_id *match;



  match = of_match_device(xillybus_of_match, &op->dev);



  if (!match)

    return -EINVAL;
```

The first operation is a sanity check, verifying that the probe was called on a device that is relevant. This is probably not really necessary, but this check appears in many drivers.

Accessing registers

Next, the memory segment is allocated and mapped into virtual memory:

```
  int rc = 0;
  struct resource res;
  void *registers;

  rc = of_address_to_resource(&op->dev.of_node, 0, &res);
  if (rc) {
    /* Fail */
  }

  if  (!request_mem_region(res.start, resource_size(&res),
"xillybus")) {
    /* Fail */
  }

  registers = of_iomap(op->dev.of_node, 0);

  if (!registers) {
    /* Fail */
  }
```

of_address_to_resource() populates the “res” structure with the memory segment given by the first “reg” assignment (hence the second argument = 0) in the peripheral’s device

tree entry. In our example it's "reg = < 0x50000000 0x1000 >", meaning that the allocated chunk starts at physical address 0x50000000 and has the size of 0x1000 bytes. of_address_to_resource() will therefore set res.start = 0x50000000 and res.end = 0x50000fff.

In the absence of an automatic tool for Zynq, the address and size should be copied manually from the address map defined in XPS (click the "Addresses" tab XPS' main window).

Then request_mem_region() is called in order to register the specific memory segment, like any device driver. The purpose is just to avoid clashes between two drivers accessing the same register space (which should never happen anyhow). The resource_size() inline function returns the size of the segment, as one would expect (0x1000 in this case).

The of_iomap() function is a combination of of_address_to_resource() and ioremap(), and is essentially equivalent to ioremap(res.start, resource_size(&res)). It makes sure that the physical memory segment has a virtual memory mapping, and returns the virtual address of the beginning of that segment.

Needless to say, these operations need to be reverted before the module is removed from the kernel or if an error occurs later on.

It may be tempting to use the "register" pointer just like any pointer, or "better" still, a pointer to volatile. The rule of thumb in Linux kernel programming is that if you feel tempted to use the "volatile" keyword, you're doing something wrong: The correct way to access hardware registers is with iowrite32(), ioread32() and other io-something functions and macros. All device drivers demonstrate this.

Note that even though nothing will crash when attempting to access the hardware registers by using the "register" variable as a plain pointer, this is likely to lead to cache coherency problems, in particular on an ARM platform like the Zynq EPP.

Attaching the interrupt handler

The driver's side to this is quite simple. Something like this:

```
irq = irq_of_parse_and_map(op->dev.of_node, 0);
```

```
  rc = request_irq(irq, xillybus_isr, 0, "xillybus",
op->dev);
```

The irq_of_parse_and_map() call merely looks up the interrupt's specification in the device tree (more about this below) and returns its identifying number, as request_irq() expects to have it ("irq" matches the enumeration in /proc/interrupts as well). The second argument, zero, says that the first interrupt given in the device tree should be taken.

And then request_irq() registers the interrupt handler. This function is explained in the LDD3 book.

The device tree declaration goes something like (copied from above):

```
     interrupts = < 0 59 1 >;

     interrupt-parent = <&gic>;
```

So what are these three numbers assigned to "interrupt"?

The first number (zero) is a flag indicating if the interrupt is an SPI (shared peripheral interrupt). A nonzero value means it is an SPI. The truth is that these interrupts **are** SPIs according to Zynq's Technical Reference Manual (the TRM), and still the common convention is to write zero in this field, saying that they aren't. Since this misdeclaration is so common, it's recommended to stick to it, in particular since declaring the interrupt as an SPI will cause some confusion regarding the interrupt number. This is discussed in detail here.

The second number is related to the interrupt number. To make a long story short, click the "GIC" box in XPS' main window's "Zynq" tab, look up the number assigned to the interrupt (91 for xillybus in Xillinux) and subtract it by 32 (91 - 32 = 59).

The third number is the type of interrupt. Three values are possible:

- 0 — Leave it as it was (power-up default or what the bootloader set it to, if it did)
- 1 — Rising edge
- 4 — Level sensitive, active high

Other values are not allowed. That is, falling edge and active low are not supported, as the hardware doesn't support those modes. If you need these, put a NOT gate in the logic.

It's notable that the third number is often zero in "official" device trees, so the Linux kernel leaves the interrupt mode to whatever it was already set to. This usually means active high level triggering, and still, this makes the Linux driver depend on that the boot loader didn't mess up.

Finally, the interrupt-parent assignment. It should always point to the interrupt controller, which is referenced by &gic. On device trees that were reverse compiled from a DTB file, a number will appear instead of this reference, typically 0x1.

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

Application-specific data

As mentioned earlier, the device tree is commonly used to carry specific information, so that a single driver can manage similar pieces of hardware. For example, if the hardware is an LCD display driver, the information about its pixel dimensions and maybe even physical dimensions may appear in the device tree. Serial port interface hardware (i.e. RS-232, UART) is likely to inform the driver about what clock frequency is driving the logic, so that the driver can set up the clock division registers correctly. And so on.

In the simplest, and most common form, this information is conveyed by a simple assignment in the peripheral's entry, e.g.

```
xlnx,slv-awidth = <0x20>;
```

The "xlnx," prefix merely protects against name collisions. The name string is arbitrary, but some kind of prefix is recommended at least for the sake of clarity. The "xlnx," prefix is common, because it's used by the automatic tool that generates device trees for the Microblaze soft processor.

To grab this information, the kernel code says something like

```
void *ptr;

  ptr = of_get_property(op->dev.of_node, "xlnx,slv-
awidth", NULL);



  if (!ptr) {

    /* Couldn't find the entry */

  }
```

The third argument in the call, NULL, is possibly a pointer to an int variable, to which the length of the data is written.

To access the actual data, assuming that it's a number, go

```
int value;

  value = be32_to_cpup(ptr);
```

The be32_to_cpup reads the data from the address given by "ptr", converts from Big Endian to the processor's native Little Endian, so "value" will contain the integer one expects it to contain.

There are plenty of other API functions for reading arrays etc. See drivers/of/base.c in the Linux sources.

Summary

All in all, setting up a device tree entry for a custom IP peripheral is quite simple:

- Pick a "magic string" for the "compatible" assignment, possibly the name+version format employed by the automatic tools.

- Look up the address allocation on the bus in XPS, and write the “reg=” assignment accordingly
- Add the “interrupt-parent = <&gic>” line (if interrupts are used)
- Look up the interrupt allocation in XPS, and write the “interrupt=” assignment (if applicable)
- Add any custom parameters as needed

And that’s it. Good luck!