Technical Note

http://www.OfItselfSo.com

http://www.OfItselfSo.com/BeagleNotes/AboutTheDeviceTree.pdf

v01.00 - Feb 2015

About the Device Tree

There is a fair amount of information about the Linux Device Tree on the Internet and those pages all contain various and useful insights. When I was looking (Autumn 2014), there did not seem to be anything intended for the beginner which brought all of the information together and presented specific novice friendly advice on the Device Tree structure and how to decompile, configure and re-compile the Device Tree. This paper has been written in an attempt to address that issue. In other words, this document contains the information I would like to have had when I was first trying to figure out the Device Tree.

The discussion below is primarily focused on the standard (as shipped) Debian Linux distribution for the Beaglebone Black running the 3.8 kernel but many of the sections are generic and should be reasonably usable for other micro-controllers and Linux versions.

Before the Device Tree

Before the Device Tree, the Linux kernel contained the all of the information about the hardware of every supported platform. This information, such as memory locations, interrupts, on chip peripherals and many, many, other things was compiled into the kernel. This approach worked fairly well when there were just a few platforms being supported.

Previously, the Linux kernel contained all of the device configuration for each supported platform.

Due to the fact that a description of every hardware platform was built into the kernel source, the boot loader could tell the kernel which platform it was running on by passing in a value (known as the machine type integer) at start up. The kernel would then internally look up the appropriate platform parameters and then use them to figure out how to utilize the hardware available to it.

The rapid proliferation of microcontroller boards necessitated a new approach to configuring each target platform.

There are problems with this *hard coded* approach. The first problem is that recent times have seen an ever proliferating number of small microcontroller boards each with their own set of hardware. The Beaglebone Black and Raspberry Pi are two common ones – but there are many others. The second, and related issue, is that the Linux kernel is centrally maintained (called the mainline) and the maintainers were having a hard time keeping up. Eventually Linus Torvalds had enough and issued one of his epic rants (warning NSW). Linus declared that henceforth no longer would each and every new device be supported in the mainline kernel and that a new solution must be found.

The Device Tree is now used to provide platform specific configuration information.

The choice of forking the Linux kernel code and implementing non-mainline configurations for each new micro-controller was really not a serious long term option and so the Device Tree concept was developed. The Device Tree enables micro-controllers to use the same mainline kernel code along with a separate, board specific, hardware configuration. Mainline Linux kernel version 3.7 and higher all support the Device Tree.

The boot loader now passes the Device Tree configuration into the kernel at boot up.

With the Device Tree method, the boot loader reads the both the kernel image and the compiled Device Tree binary into RAM memory and then passes the memory address of the Device Tree binary into the kernel as part of the launch. The kernel, once it is running, looks at the memory address, reads the device information, configures things appropriately and then gets on with the job of running the Linux system.

How do I know which Device Tree Binary my System is Using?

The Device Tree file which the kernel reads at boot time is a *binary* file.

The end of the previous section discussed how, when using the Device Tree, the boot process consists of the boot loader reading both the Linux kernel and the Device Tree Binary into memory and then passing the memory address of the Device Tree over to the kernel for processing. This brings up an important point:

> The Device Tree file the Linux Kernel reads is a BINARY file. It is not human readable.

Device Tree Binary files have a `.dtb` filename extension.

The binary Device Tree file, which is typically given a `.dtb` file name extension, actually consists of byte code (much like Java and .NET) and the kernel can process it by using a kind of internal interpreter.

A Device Tree Compiler converts Device Tree Source (`.dts`) files into Device Tree Binary files (`.dtb`).

The fact that the Device Tree is a binary file implies that, for any given device tree configuration, there is also a compiler and some source code somewhere. This is true, however, the source code for your Device Tree will probably not be present on your system. You may have to go and find it in a repository somewhere. It can be a bit difficult to figure out which source you should use as there are a lot of them to choose from. However, there is also a de-compiler which can turn a Device Tree Binary file back into source code and, as will be discussed shortly, you can use that to re-generate the source code for any Device Tree Binary file. You can have a bit of a dig on your file system for the Device Tree Source files if you wish. Device Tree Source files usually have a `.dts` file name extension and they can be viewed in any text editor as they have a hierarchical structure similar to XML. There is much more discussion of this in later sections.

On the Beaglebone Black you are probably using the `am335x-boneblack.dtb` Device Tree Binary.

So, to return to the original question, how do you know which device tree binary your system is using? In short, on the standard out-of-the-box Beaglebone Black, the file is `am335x-boneblack.dtb` and it can probably be found in the `/boot/uboot/dtbs/` directory. This may change for future revisions though or for different distributions. If you wish to confirm for yourself which Device Tree Binary you are using then you will probably need to look in two places. First look for the `uEnv.txt` file. This file, usually located at `/boot/uboot/uEnv.txt`, provides the environment variables for the u-boot boot loader. In the `uEnv.txt` file you might find the `.dtb` file specified directly (on some systems) or, more likely, you will see a line like:

```
loadfdt=load mmc ${mmcdev}:2 ${fdtaddr} /boot/uboot/dtbs/${fdtfile}
```

The Device Tree Binary file in the above example is hard coded to be in the `/boot/uboot/dtbs` directory with the name specified by the `${fdtfile}` variable. This variable is in turn provided by the u-boot boot loader from its internal configuration (it is compiled in). You can see what the `${fdtfile}` variable will be set to by finding the `u-boot.img` file and looking for the text "`fdtfile`". Since the `u-boot.img` file is a binary file the easiest way to do this is to use the `strings` command on it.

You can also look at the `uEnv.txt` and `u-boot.img` file to find out what Device Tree Binary you are using.

```
strings /boot/uboot/u-boot.img | grep fdtfile
```

… on some systems the command returns the simple...

```
fdtfile=am335x-boneblack.dtb
```

… on others you might see a complex `if-then-fi` statement which figures out the `.dtb` file from a list of various board types and you can work it out from that.

There are also other ways to do this. For example, you can boot into the u-boot command line and use the `printenv` command to see the environment variables. This will not be discussed further here – you can easily look it up if sufficiently interested.

Decompiling the Device Tree

You can de-compile a .dtb binary back to the .dts source.

Once you have figured out what Device Tree Binary your system is using you can either look it up online and download the source from a repository or you can just decompile the binary right back into the `.dts` source from whence it came.

If you do decompile the Device Tree Binary file you might find that the source code does not look exactly the same as the one you get if you downloaded it. This is because the decompilation process produces a single flat tree whereas the source you download will probably have include files which include multiple other files which eventually build the tree. Either way, the code equates to the same thing – but if you just decompile the `.dtb` file you don't have the additional problem of getting all the include files to work properly.

Getting the Device Tree Compiler/De-Compiler

The Device Tree Compiler is also the de-compiler.

The de-compiler for the Device Tree is also the compiler – each mode uses a different flag. You can check to see if you have the Device Tree Compiler by issuing the command (as root)

```
dtc -v
```

If you get an error, you need to acquire the Device Tree Compiler. On the Beaglebone Black, until it gets accepted "upstream", the Device Tree Compiler is supplied by Robert C. Nelson and can be obtained using the following sequence of commands (as root)…

```
wget -c https://raw.github.com/RobertCNelson/tools/master/pkgs/dtc.sh
chmod +x dtc.sh
./dtc.sh
```

You will probably have to download the Device Tree Compiler.

…after installation, the `dtc -v` command should be fully functional.

```
~# dtc -v
Version: DTC 1.4.0-gf345d9e4
```

Now you are ready to decompile. First, change into the `/boot/uboot/dtbs` or other directory on your system where the Device Tree Binary files are located. Note, there will probably be a lot of `.dtb` files in there.

```
cd /boot/uboot/dtbs
```

The de-compilation of a Device Tree Binary is pretty straight forward.

Earlier you found out which Device Tree Binary you are using (most likely `am335x-boneblack.dtb` on the Beaglebone Black) so we can decompile it with the command

```
dtc -I dtb -O dts am335x-boneblack.dtb > bbb.dts
```

We can view the file by looking at it in an editor.

```
nano bbb.dts
```

At this point you can recompile the `.dts` file back into a binary – but first a ***WARNING***.

WARNING: Changing the Device Tree can Render the System Unbootable.

Modifying the Device Tree incorrectly can cause your system to fail to boot. You should plan for this.

If you modify the Device Tree Source you are messing with some serious and fundamental system parameters. If you make a wrong or erroneous configuration change you can render your system unbootable. That is "*unbootable*" as in "*bricked*". NOTE: This is not just a theoretical possibility - you are sure to do this eventually and so you must prepare ahead of time for this event.

One useful way to recover from a bad Device Tree configuration is to do your experimentation on an operating system that boots off of a removable device. On the Beaglebone Black this is the microSD card. If you boot off of an O/S on a microSD card and you make a change to the Device Tree and the attempted reboot fails then all you need to do is remove the SD card and transport it to another device (a PC with an SD card slot). On the second device you can just mount the microSD card and use the command line or text editor to undo the changes. Once the changes have been reverted you can replace the card into the Beaglebone Black and boot up – fairly short and simple.

If you boot off of internal memory such as EMMC your recourse is to create a bootable SD card, boot from that, mount the EMMC memory as a directory and revert the changes there.

So here I go again – this is important enough to emphasise once more.

WARNING, WARNING, WARNING

As an amateur it is highly probable that if you modify the Device Tree often enough you will eventually make a change that prevents your Beaglebone Black from booting. If you took care to make and/or boot from a bootable SD card it is relatively easy to undo any changes. If your Beaglebone Black boots from EMMC memory then you need to have a backup image on SD card you can re-flash with or boot from.

Work from a bootable removable device or have a backup you can boot from.

ADVICE: Work from an O/S on a removable device. If you cannot do this take the time to make a bootable backup on an SD card so you can boot from it and mount the EMMC memory to fix things. In either case, practice the reboot process before doing anything to the Device Tree. You will not regret investing the time!

COMPILING THE DEVICE TREE

Before you compile the Device Tree Source into a Device Tree Binary please read the warnings in the previous section. It is possible to make changes to the Device Tree which can, if you attempt to reboot using it, cause the boot process to fail. You need to have a plan to cope with that ahead of time.

Compiling the Device Tree Source is just a different option on the Device Tree Compiler.

Compiling a flat Device Tree Source file into a Device Tree Binary is pretty straight forward. The example code below assumes that you have created a Device Tree Source file named `bbb.dts` by decompiling the `am335x-boneblack.dtb` file as discussed in the previous section. If you are attempting to compile Device Tree Sources downloaded from a repository on the Internet the process is a bit more complex and is not discussed here.

First rename our existing `am335x-boneblack.dtb` file so we can recover it if things go wrong. All of the commands are performed as root and the `.dtb` files are, in this example, in the `/boot/uboot/dtbs` directory.

Note how this section saves a copy of the original Device Tree Binary.

```
cd /boot/uboot/dtbs
cp am335x-boneblack.dtb am335x-boneblack.dtb_ORIGINAL
```

Now we recompile the source and rebuild the `.dtb` file

```
dtc -I dts -O dtb  bbb.dts > am335x-boneblack.dtb
```

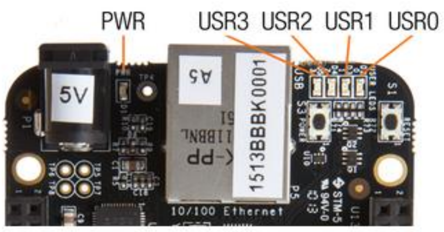
At this point we can restart by issuing the reboot command. IMPORTANT, heed the previous warnings and make sure you can cope with a boot failure here.

```
reboot
```

TESTING THE NEW DEVICE TREE

If all went well in the previous section, your Beaglebone Black has now rebooted and is running exactly the same as it was before. This is not surprising as the compilation of the Device Tree Source should create exactly the same Device Tree Binary as was previously decompiled.

Ideally, you'll want to make a change to see if you really have modified the correct Device Tree Binary file. The easiest way to make a simple, visible change is to disable the heartbeat LED. The heartbeat LED is the little green USR0 light on the Beaglebone Black that blinks at 2Hz to show you that everything is OK.



The USR LEDs on the Beaglebone Black. Credit: http://beagleboard.org/getting-started

As a test we can modify the Device Tree to stop the USR0 LED from blinking.

Here is how to modify the Device Tree Source to disable the USR0 LED. Open up the `bbb.dts` source file you previously decompiled in a text editor and have a look at the structure of the file. We will go into what everything means in a future section but for now find the section that looks like

```
led0 {
     label = "beaglebone:green:usr0";
     gpios = <0x5 0x15 0x0>;
     linux,default-trigger = "heartbeat";
     default-state = "off";
 };
```

These lines will be buried well down in the Device Tree Source file. If you are using the `nano` editor use the CTRL-W key to look for the text "`heartbeat`". Change the trigger statement so it says "`none`". For example...

```
led0 {
     label = "beaglebone:green:usr0";
     gpios = <0x5 0x15 0x0>;
     linux,default-trigger = "none";
     default-state = "off";
};
```

We always have to reboot after re-compiling the Device Tree Binary.

... then re-compile the Device Tree Binary ...

```
dtc -I dts -O dtb  bbb.dts > am335x-boneblack.dtb
```

... and reboot

```
reboot
```

After the reboot you should find that the heartbeat led no longer flashes but that all other functionality on the Beaglebone Black remains intact.

If you make a syntax error (missing semi-colon for example) the device tree compiler will not successfully compile the source so you don't have to worry about creating a bad Device Tree Binary due to a minor typo. Having said that, if you make a configuration change which is syntactically correct but which is erroneous or which conflicts with some other setting you may well find the system no longer boots or if it does boot then large swathes of functionality are missing.

You should now undo any changes you made and reboot.

Probably at this point you should re-edit the `bbb.dts` source file and turn the heartbeat LED back on. The heartbeat LED is one of the signals that indicate the Beaglebone Black has booted correctly and you'll probably want to check it after some of the future reboots.

The Structure of the Device Tree

If you open up a Device Tree Source file in a text editor you will see what appears to be a very complex, but highly ordered, layout of information. A summary of the structure will be presented here and you can find a comprehensive, but possibly not so novice friendly, discussion at the devicetree.org website using the link http://www.devicetree.org/Device_Tree_Usage

Nodes and Properties

The Device Tree source file consists of nodes and key-value pairs known as properties. Nodes can have sub-nodes as well. For example:

```
/ {
    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 {
        };
    };
};
```

The Device Tree consists of a hierarchical list of nodes, sub-nodes and properties.

Credit: http://www.devicetree.org/Device_Tree_Usage

Nodes are delimited by curly Braces "{}". Properties can have zero, one or many values.

In general, if you see a set of curly braces "{}" you know you are looking at a node. If you see an equals sign "=" then you are dealing with a key-value pair property and if there is no equals sign then the presence of the name of the property is considered sufficient to invoke the desired configuration option and no values are necessary.

Any data inside of angle brackets "<>" are space delimited 32 bit integers.

Also note that properties can have multiple values associated with them. Sometimes these multiple values are comma delimited and sometimes they placed inside a data structure group and space delimited. The data structure groups are angle brackets "<>" (which means 32 bit integer numeric data is enclosed) and, less commonly, square brackets "[]" (which enclose binary data). Strings are always enclosed in double quotes. It is possible that a property can contain multiple items of different types all on the same line – although it is uncommon to see this in practice.

You will sometimes see the int32 values enclosed within in angle brackets "<>" referred to as cells.

OK. Now that we have the basics let's look at a Device Tree Source file which was just decompiled from the `am335x-boneblack.dtb` on the Beaglebone Black standard distribution. It is too long to display here in its entirety. However the first part looks like…

```
/dts-v1/;
/ {
   #address-cells = <0x1>;
   #size-cells = <0x1>;
   compatible = "ti,am335x-bone", "ti,am33xx";
   interrupt-parent = <0x1>;
   model = "TI AM335x BeagleBone";

   chosen {
   };

   aliases {
           serial0 = "/ocp/serial@44e09000";
           serial1 = "/ocp/serial@48022000";
           serial2 = "/ocp/serial@48024000";
           serial3 = "/ocp/serial@481a6000";
           serial4 = "/ocp/serial@481a8000";
```

```
                serial5 = "/ocp/serial@481aa000";
        };

        memory {
                device_type = "memory";
                reg = <0x80000000 0x10000000>;
        };

        cpus {

                cpu@0 {
                        compatible = "arm,cortex-a8";
                        operating-points = <0xf4240 0x149970 0xc3500 ……>;
                        voltage-tolerance = <0x2>;
                        clock-latency = <0x493e0>;
                        cpu0-supply = <0x2>;
```

The full Device Tree is well over 1300 lines on the Beaglebone Black.

…. and so on for another 1300 lines.

At first glance this code seems to be quite incomprehensible. However, there is a lot we can do to reduce the complexity. First, we note that if we remove everything contained within the level 1 nodes it can be seen there is actually a relatively simple outer structure…

```
   /dts-v1/;
   / {
      chosen {...};
      aliases {...};
      memory {...};
      cpus {...};
      pmu {...};
      soc {...};
      pinmux@44e10800 {...};
      ocp {...};
      fixedregulator@0 {...};
      __symbols__ {...};
   };
```

If we collapse everything but the top level nodes we see that the structure appears to be much simpler.

Assuming you are approaching the Device Tree as user who simply wants to adjust a few settings to enable some of the Beaglebone Blacks on board peripherals (such as the GPIO's, SPI ports or PWM's etc.) then most of these level 1 nodes can be completely ignored.

Most likely you will only ever have to concern yourself with the `pinmux@44e10800` and `ocp` nodes.

In order to configure the various peripheral subsystems the only nodes whose contents are likely to be of interest are the `pinmux@44e10800{}` node and the `ocp{}` node.

So let's have a look at the `ocp{}` node and its contents. This node is very large – most of the `.dts` file in fact. However stripping things down let's look at the top and then skip a large chunk of the contents till we get down to the section that configures our old friend the USR0 LED which was discussed previously.

```
        ocp {
                compatible = "simple-bus";
                #address-cells = <0x1>;
                #size-cells = <0x1>;
                ranges;
                ti,hwmods = "l3_main";
                linux,phandle = <0x15>;
                phandle = <0x15>;

                <<<< many, many subnodes skipped in here>>>>

                gpio-leds {
                        compatible = "gpio-leds";
                        pinctrl-names = "default";
                        pinctrl-0 = <0x3>;
                        led0 {
                                label = "beaglebone:green:usr0";
                                gpios = <0x5 0x15 0x0>;
                                linux,default-trigger = "heartbeat";
                                default-state = "off";
                        };
                        <<<<  subnodes skipped in here>>>>
                };
                <<<< many more subnodes skipped in here>>>>
            };
```

The Compatible Property

At the Linux level, all access between the user and the hardware is ultimately performed by a device driver which provides a device interface. The device driver is custom written to know how to access the hardware. So one would logically expect the Device Tree to specify the device driver somewhere and, indeed, that is usually the first property we see under a node. Note in the example above under the `ocp` node there is a line which says...

```
compatible = "simple-bus"
```

… and under the `gpio-leds` sub-node there is a similar line property which says

```
compatible = "gpio-leds";
```

The `compatible` property tells the kernel which device driver to use.

The `compatible` property is telling the kernel which device drivers it can use to handle that particular node. There are a lot of device drivers available and it is possible that, for any particular node, there may be several which will work. In fact it is possible to specify a list of compatible device drivers (hence the name of the property). The kernel will read this line, choose the first one, and interrogate every device driver it knows about. If it gets a match it uses that driver, if it gets no matches it uses the next value in the property (if any) and looks for that.

If a node does not have a `compatible` property it can be assumed that the node and property information is simply being fed into the device driver of the node above it in the hierarchy.

We can look up the gpio-leds device driver specified by the `gpio-leds` sub-node if we wish – a link is...

https://www.kernel.org/doc/Documentation/devicetree/bindings/leds/leds-gpio.txt

… and in fact it is interesting to do so because we see that not only does it tell us all of the information it needs but even includes some example Device Tree configurations.

This leads us to an important point about the Device Tree. All of the information below the `gpio-leds{}` node is intended for the gpio-leds driver. This device driver consumes it and uses it to configure itself.

A new `compatible` property on a sub-node tells the kernel to use a different device driver.

```
gpio-leds {
        compatible = "gpio-leds";
        pinctrl-names = "default";
        pinctrl-0 = <0x3>;

        led0 {
            label = "beaglebone:green:usr0";
            gpios = <0x5 0x15 0x0>;
            linux,default-trigger = "heartbeat";
            default-state = "off";
        };

        led1 {
            label = "beaglebone:green:usr1";
            gpios = <0x5 0x16 0x0>;
            linux,default-trigger = "mmc0";
            default-state = "off";
        };

        led2 {
            label = "beaglebone:green:usr2";
            gpios = <0x5 0x17 0x0>;
            linux,default-trigger = "cpu0";
            default-state = "off";
        };

        led3 {
            label = "beaglebone:green:usr3";
            gpios = <0x5 0x18 0x0>;
            default-state = "off";
            linux,default-trigger = "mmc1";
        };
    };
```

Similarly the `ocp{}` node information is passed to the `simple-bus` compatible driver – at least until it encounters a sub-node (like the `gpio-leds` node) with its own `compatible` property. At that time the information below that node is passed into the new driver. Thus the Device Tree can be viewed as a strictly defined cascading hierarchy of device drivers, configuration information and sub-nodes with their own device drivers and configuration information.

The content of the nodes in the Device Tree is as variable as it is because the information is intended for different device drivers.

> The reason why each node in the Device Tree Source appears to have a different structure is because the configuration information they contain is intended for different device drivers - each with its own requirements.

Within any one device driver, as can be seen by the `led0`, `led1`, `led2` and `led3` nodes in the example above, the configuration information is remarkably consistent.

> The important point to remember here is that, unless you are a system architect, you do usually not need to know what most of the nodes in the Device Tree Source do and what their configuration information means.

Don't let the complexity worry you. You do not need to know what most of the nodes in the Device Tree do.

In other words, don't let all the complexity worry you. The structure and meaning of the nodes you need to know about is documented and most likely any changes you make will be a variation on an example you found somewhere else.

OCP AND THE PINMUX

There are many functional I/O devices included with the CPU. These are called On Chip Peripherals (OCP).

The CPUs of most modern microcontrollers (including the AM335x 1GHz ARM Cortex-A8 in the Beaglebone Black) are typically built to handle a multitude of requirements. For example, designers of a microcontroller board using that CPU may wish to have multiple GPIO's, SPI, I2C, PWM, A/D, HDMI, USB ports and many others. The architects of the CPU are aware of this and typically implement as many of these devices as they can just to make sure their chip gets selected for people's designs. These types of device are called On Chip Peripherals (hence the acronym OCP) and although they are called *peripherals* they are actually built right onto the silicon with the CPU itself.

Each OCP device will probably require one or more physical inputs or outputs on the exterior of the CPU so that the rest of the system can interface with it. The problem with this is that, given all the OCP devices which are implemented, there are many more OCP device I/O requirements than there are pads on the exterior of the CPU.

There are so many OCP devices that they have to share the I/O lines on the CPU. This means you cannot simultaneously enable two OCP devices that use shared resources.

The designers of the CPU, reasoning that most people implementing their CPU will never need to use all OCP devices at once, have implemented a rather elegant solution to the problem. They have set things up so that OCP devices share the I/O pads on the CPU. For example, the same I/O pad is alternately used by the following OCP devices: `GPIO0_5`, `I2C1 SCL`, `mmc2 sdwp`, `SPI cs0` and others. There are up to 8 possible usages of any CPU pad and the actual usage is settable at runtime. The name for the possible usages is called the "*mode*" (mode0 to mode7).

The I/O line on an OCP device is called a "pin". The physical connection on the exterior of the CPU is called a "pad" – but is also referred to as a "pin" often enough that you have to take care to figure out what is really meant.

> **Nomenclature:** the input or output of the OCP device is called a "*pin*". The actual physical I/O pin on the CPU through which the OCP pin may or may not be eventually exposed is called a "*pad*".

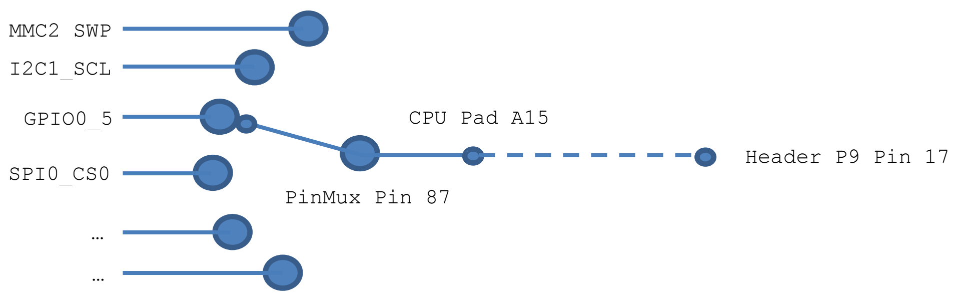
Be aware of that when you read the documentation that much of the time when you see the word "*pin*" it is an OCP device I/O line internal to the CPU which is being referenced. It must be noted that sometimes you will see the physical pad on the CPU referred to as a "*pin*" and once the CPU pad has been routed out and exposed on a header block (the Beaglebone Black P8 or P9 headers for example) it is also commonly referred to as a "*pin*". The meaning of the word "*pin*" is just one of those things you have to infer from the context and which can drive you insane if you don't realize what is going on.

The PinMux controls which of the OCP device pins are routed to the pads on the bottom of the CPU (and hence to the rest of the board).

The internal CPU component that does the switching of OCP pins onto physical pads is called the PinMux. The PinMux configuration (and hence "*pinmux mode*") for all OCP devices is set by the Device Tree at boot – although it can also be dynamically adjusted later by software running in kernel mode (usually device drivers). Note that normal user mode software (even running as root) cannot adjust the PinMux mode settings.

Actually, the PinMux is itself an OCP device – however it is so important and fundamental to the operation of the rest of the OCP devices it is usually handled separately. This is why you see individual `ocp{}` and `pinmux{}` level 1 nodes in the Device Tree.

Let's look at how the PinMux works. In concept it can be visualised as a simple switch



The PinMux can be Visualised as a Simple Rotary Switch.

The PinMux mode determines which OCP device line is active on any pin.

Of course the PinMux is not a mechanical rotary switch like that – it is all solid state on the silicon wafer of the CPU – but the effect is the same. In the example above, if you put PinMux pin 87 in mode 7 you will get `GPIO0_5` present on the CPU pad A15 and this has in turn been routed out to pin 17 on the P9 header by the designers of the Beaglebone Black. If you put the PinMux pin in mode 0 then the `SPI Port0 CS0` line will be present on pin 17 on the P9 header. It should be noted that not all CPU pads go to the P8 and P9 headers – some go straight to the USB ports, HDMI port or EMMC memory etc. depending on what the designers of the Beaglebone Black (not the CPU) wanted to do with them.

The PinMux pins each OCP device uses are hardcoded by the designers of the CPU and cannot be changed.

It is important to realize that the OCP pins which use any particular CPU pad are hard coded – really hard coded - as in burnt onto the silicon. This means that any specific OCP device must use a pre-defined set of CPU pads and this cannot be changed. If two OCP devices use the same CPU pad then you cannot enable both at the same time. You can visualize this in the above diagram by realizing that the PinMux switch cannot both point to `GPIO0_5` and `SPI0_CS0` at the same time.

The above reason is why you will sometimes see comments like "*if you wish to use GPIO_77 you must disable the HDMI video*". The reason is that both of those two OCP devices share the same PinMux pin in different modes. You simply cannot use both at once and there is no way to change this.

OCP Devices which are not properly configured in the PinMux can still sometimes be accessed but will not work.

Note: an OCP device which is not enabled in the PinMux will still sometimes "work" in the sense that you can access them via their Device Drivers. However, if the PinMux mode is incorrect, any I/O lines that device uses will not be connected to anything.

PHANDLES, PINCTRL AND PINCTRL-0

OCP Devices usually need configuration information in the PinMux node of the Device Tree as well as the OCP node.

If you have been following the discussion in the previous section it will now be clear that in order to use any OCP device we really have to configure two things in the Device Tree. We have to configure the device in the `ocp{}` node in order to launch and configure its device driver and we also have to configure the PinMux in the `pinmux@44e10800` section to make its pins visible on the CPU pads (and hence elsewhere on the board).

By inference, we can guess that the two sections are probably not isolated. The OCP section is going to need to know what the PinMux node is doing so that it can configure itself appropriately. It can also be assumed that if one OCP device needs to refer to the goings on in the PinMux then probably others will need to do so as well.

The `phandle` **property provides a label for other nodes in the Device Tree.**

Given the hierarchical nature of the Device Tree it is also logical to assume that there is probably some sort of mechanism to label a node and then reference that label if nodes elsewhere need to refer to it. This is the purpose of the `phandle` or `linux,phandle` properties.

The values of the `phandle` or `linux,phandle` properties are 32 bit integers which uniquely identify the node. You will often see both used and they mean the same thing.

```
pinmux@44e10800 {
        compatible = "pinctrl-single";
          …..
        pinmux_userled_pins {
            pinctrl-single,pins = <0x54 0x7 0x58 0x17 0x5c 0x7 0x60 0x17>;
            linux,phandle = <0x3>;
            phandle = <0x3>;
        };
    …..
```

In the above example the `pinmux_userled_pins{}` sub node of the `pinmux@44e10800{}` node has a phandle of 3 (the decimal value of `0x3`) and this can be used to refer to that node elsewhere in the Device Tree. Also note the use of the angle brackets "<>" to denote the fact that the phandle value is a 32 bit integer.

The `pinctrl-0` **property is typically used to reference a** `phandle` **in the PinMux node**

References elsewhere in the Device Tree will use various property names to refer to a phandle but one of the most common, the one that is used to refer to `pinmux@44e10800{`} node phandles, is called `pinctrl-0`. We can see this used in the `gpio-leds{}` sub node of the `ocp{}` node

```
    ocp {
        …..
        gpio-leds {
            compatible = "gpio-leds";
            pinctrl-names = "default";
            pinctrl-0 = <0x3>;

            led0 {
                label = "beaglebone:green:usr0";
                gpios = <0x5 0x15 0x0>;
                linux,default-trigger = "heartbeat";
                default-state = "off";
            };
            …..
```

Clearly the gpio-leds device driver needs to know about the state of the PinMux in order to properly configure the USR0 led and it finds the information it needs by looking up the node with the phandle of `0x3`. It knows to do this because it is told to do so by its `pinctrl-0` property.

You might be wondering why the number `0x3` is used as a unique identifier. In fact it is possible to use a text designator. We could configure the Device Tree to use a label like "`MY_LEDPINS`" if we wished. In that case the `pinmux_userled_pins` node would be defined like...

It is possible to use text labels instead of numbers to reference `phandles`.

```
MY_LEDPINS: pinmux_userled_pins {….}
```

and the reference to it would be implemented as

```
pinctrl-0 = <&MY_LEDPINS>;
```

As far as I can tell textual labels are uncommon – I haven't seen them used very often and I am not sure why. They are preserved though the compilation and de-compilation process with the Beaglebone Black Device Tree Compiler but perhaps they were not with earlier versions so that may be the reason.

The Unit Address

In the Device Tree Source you will sometimes see nodes which contain an "@" character in their name. For example:

```
ocp {
   …..
   gpio@44e07000 {...
   gpio@4804c000 {...
   gpio@481ac000 {...
   gpio@481ae000 {...
   …..
```

The text after an "@" character in a node name is called the Unit Address. It refers to an internal memory structure inside the device driver that consumes the Device Tree information.

The text after the "@" character is called a Unit Address and its meaning is relevant only to the device driver accepting the Device Tree information. Unit Addresses typically represent a location in the parent nodes address space. In the above example, the Unit Address specifies an actual physical memory address location, however, it does not have to do so. It could be possible to see a list of subnodes labelled like…

```
ocp {
   …..
   led@0 {...
   led@1 {...
   led@2 {...
   led@3 {...
   …..
```

… if the addresses 0, 1, 2, 3 were somehow meaningful to the device driver being configured in the node above.

The Device Tree Structure and the Device Files

As has been mentioned previously, the Device Tree forms a hierarchical tree structure. It is interesting to note that once the Device Tree has been processed, the information it contains is brought out and presented at the file system level as a series of directories and files.

The contents of the Device Tree are exposed in a directory structure by the kernel after boot. This provides a debug reference.

The directory `/proc/device-tree` is the root of the tree and represents the root of the Device Tree. If we look at the contents of the `/proc/device-tree` directory we see...

```
ls -l /proc/device-tree
-r--r--r--  1 root root  4 Apr 23 20:33 #address-cells
-r--r--r--  1 root root  4 Apr 23 20:33 #size-cells
dr-xr-xr-x  2 root root  0 Apr 23 20:33 __symbols__
dr-xr-xr-x  2 root root  0 Apr 23 20:33 aliases
dr-xr-xr-x  5 root root  0 Apr 23 20:33 bone_capemgr
dr-xr-xr-x  2 root root  0 Apr 23 20:33 chosen
-r--r--r--  1 root root 25 Apr 23 20:33 compatible
dr-xr-xr-x  3 root root  0 Apr 23 20:33 cpus
dr-xr-xr-x  2 root root  0 Apr 23 20:33 fixedregulator@0
-r--r--r--  1 root root  4 Apr 23 20:33 interrupt-parent
dr-xr-xr-x  2 root root  0 Apr 23 20:33 memory
-r--r--r--  1 root root 21 Apr 23 20:33 model
-r--r--r--  1 root root  1 Apr 23 20:33 name
dr-xr-xr-x 56 root root  0 Apr 23 20:23 ocp
dr-xr-xr-x 11 root root  0 Apr 23 20:33 pinmux@44e10800
dr-xr-xr-x  2 root root  0 Apr 23 20:33 pmu
dr-xr-xr-x  3 root root  0 Apr 23 20:33 soc
```

If you compare the above list with Device Tree Source file you will immediately see that every level 1 node is represented as a directory. If we change into the `ocp` directory and then in to the `gpio-leds` directory we can again see some very familiar constructs.

```
-r--r--r-- 1 root root 10 Apr 23 22:33 compatible
dr-xr-xr-x 2 root root  0 Apr 23 20:37 led0
dr-xr-xr-x 2 root root  0 Apr 23 20:37 led1
dr-xr-xr-x 2 root root  0 Apr 23 20:37 led2
dr-xr-xr-x 2 root root  0 Apr 23 20:37 led3
-r--r--r-- 1 root root 10 Apr 23 20:37 name
-r--r--r-- 1 root root  4 Apr 23 20:37 pinctrl-0
-r--r--r-- 1 root root  8 Apr 23 20:37 pinctrl-names
```

The properties are just normal files and we can view the contents if we wish. Issuing the command...

```
strings compatible
```

.. returns the text ...

The `/proc/device-tree` directory tree only shows the contents of the Device Tree at boot. It does not show later configuration changes.

```
gpio-leds
```

… and this is exactly the contents of that property in the Device Tree source.

As has been seen, the `/proc/device-tree` directory tree provides a nice file system mapping of the contents of the Device Tree Binary the kernel processed at boot. However certain configuration items can be changed at runtime by user space programs interacting with device drivers. The PinMux settings are a good example of this. None of these changes will be reflected in the `/proc/device-tree` directory tree.

If your Linux kernel has been compiled with the `CONFIG_DEBUG_FS` flag (and on the Beaglebone Black default installation this is the case) then the current state of the device configuration can be found in the /`sys/kernel/debug` directory tree. This tree exists to provide the current device configuration information to user space programs and as changes are made these files will change.

The `/sys/kernel/debug` directory tree exists to provide the current device configuration information. As changes are made these files will change.

All of the contents of the `/sys/kernel/debug` will not be discussed here – it is much too large a subject for this document and the file system structure is not nearly so clean a one-for-one mapping as is seen in the `/proc/device-tree` directory tree.

The current configuration state of the PinMux can be found in the file …

```
/sys/kernel/debug/pinctrl/44e10800.pinmux/pins
```

… if we look at the contents of this file with a `cat` command we can see there are a number of lines like…

```
…
pin 85 (44e10954) 00000037 pinctrl-single
pin 86 (44e10958) 00000062 pinctrl-single
pin 87 (44e1095c) 00000062 pinctrl-single
pin 88 (44e10960) 0000002f pinctrl-single
pin 89 (44e10964) 00000027 pinctrl-single
…
```

Note the line referencing pin 87. This is the PinMux pin discussed earlier in the *OCP and PinMux* section. Interpreting this line is the subject for another document but we can see from the `00000062` item that the PinMux mode for this pin is "2" and by referring to other documentation we can note that this means that the SCL line of the I2C2 device is currently being routed out though CPU pad A15 and hence is visible on Header 9 Pin 17. Any attempt to use the GPIO0_5 OCP device will be disappointing. It will not work – you won't get an error but any read or write will be meaningless as its I/O line simply is not connected to anything.

Various device drivers also implement their own device files and sometimes file system trees.

There are other directories and files which map onto the OCP devices in the Device Tree. One commonly used one is the `/sys/class/gpio` directory which allows user space programs to manipulate the OCP GPIOs as if they were files. This is known as the *SYSFS* file system. It is important to realize that these files are created and maintained by the device drivers and when you read and write to them you are actually interacting with the device driver itself. There are lots of these types of file systems and they have different names and different access mechanisms. For example, the file system which can interact with the SPI ports as if they were files is called *SPIDEV* and if it has been enabled in the Device Tree the files to access it can be found at `/dev/spidev*`

Note, if you manipulate the device files they still will not work if the OCP device is not enabled in the PinMux. For example, if you change `GPIO0_5` to be an output you can set its state to 0 or 1 all you wish via SYSFS but if the PinMux mode is not correct you will see no change on pin 17 of the P9 header. Manipulating an OCP device via SYSFS (or SPIDEV or whatever) has absolutely no effect on the PinMux.

DEVICE CONFIGURATION CHANGES AT RUNTIME

Recall from previous sections that the Device Tree is used by the kernel at boot time to configure itself and also that it uses the Device Tree to find and launch the appropriate device drivers for various system purposes. Much of the Device Tree configuration information is consumed by the device drivers themselves rather than by the kernel directly.

All of this configuring and launching of device drivers takes place in kernel mode and user space programs (even ones running as root) simply do not have the permissions to do this sort of thing. Having said that, there are ways user space programs can configure OCP devices (and launch device drivers) – equally there are some things that just cannot be done.

User space programs typically interact with OCP devices via a device driver.

The typical way user space programs interact with OCP devices is by talking to the relevant device driver. This is what is happening when you use SYSFS to enable a GPIO, configure it as an output and set its state to high. When you "export" the GPIO and set its state and direction in the appropriate files you are actually talking to a device driver which is intercepting these instructions and doing the real work for you. Other device drivers offer only a minimal file system type interface (SPIDEV for example) and expect you to use `ioctl()` calls to interact with the device. The way you interact with the various device drivers depends on the device driver and it must be said that the `ioctl()` method is more common than the easy interface SYSFS presents.

You still need to configure the PinMux to use an OCP device. The individual device drivers typically do not do that.

As mentioned previously, neither SYSFS nor SPIDEV or other device drivers interact with the PinMux and so you still need to properly "mux" all of the pins the device needs in order to get that OCP device to work.

The OCP devices are resident in the memory of the Beaglebone Black at various well documented locations. This memory location is called the base address for the device. The configuration state of these devices is set by manipulating various memory locations offset from the device base addresses and these are known as the device registers. Each OCP device is different and hence the registers for each device are different. You have to read the AM335x Sitara Processor Technical Reference Manual to find out what registers to set for a device and what all the individual bits mean.

The OCP devices are mapped in to the system memory. It is possible to write to them by writing to the `/dev/mem` file.

Because the OCP devices are resident on the memory bus, it is logical to assume that if a user space program can figure out a way to write to a specific memory location it could then configure the OCP devices directly. In fact there is such a method, known as "*Memory Mapped Access*", which uses the device file `/dev/mem`. Essentially the `/dev/mem` file presents the Beaglebone Black's entire addressable memory as a file and you can get and set bits in the memory using simple file reads and writes from a user space program (but you have to be root). Of course this is just accessing what is effectively a different device driver – one which presents the RAM memory as the file `/dev/mem` - and it is really this device driver which has the appropriate access permissions.

> Be aware that, while using the `/dev/mem` file is possible, it is not considered "*the Linux Way*" and hence its use is deprecated. Memory mapped access introduces device specific items (memory addresses etc.) into code which should be transportable.

You still cannot write to the PinMux in the `/dev/mem` file. This is explicitly forbidden.

The PinMux is an OCP device and it is memory mapped into `/dev/mem` but you still cannot set the pinmux mode for an OCP device by twiddling bits in its registers. This is explicitly forbidden – it doesn't error if you try it, but it doesn't work either. You can read the PinMux registers though.

CAPE MANAGERS AND THE BEAGLEBONE BLACK

As discussed in the previous section, there are not a lot of good options to set device configurations at runtime from a user space program. In particular, even though you can configure various OCP device options via its corresponding device driver, if the Device Tree did not start that device driver you cannot configure anything because that device driver is not running. For example, if the Device Tree does not specify an SPI 0 port then the corresponding SPIDEV device file will not exist. So, in theory at least, the Device Tree has to specify both the OCP device and PinMux it otherwise you have very limited options. It should be noted that you can use the Memory Mapped Access method without the device driver present but that tends to require some intricate code.

It should also be noted that editing the Device Tree is a non-trivial thing and if you noted the warnings in previous sections you will know that it can render your Beaglebone Black un-bootable if you get it wrong. Of course you will have also practiced recovering from that before messing with the Device Tree.

The Beaglebone Black implements a Cape Manager system which permits fragments of the Device Tree to be applied at runtime by user space programs.

In recognition of this, the developers of the Beaglebone Black devised a mechanism called the "*Cape Manager*". Essentially this is a very clever device driver which can accept a fragment of Device Tree Binary from a user mode program, and then adjust the systems device configuration using its kernel mode privileges. This includes starting and stopping device drivers. The ultimate effect is as if the fragment was contained within the full Device Tree binary which was processed at boot time.

The fragments of Device Tree are called Device Tree Overlays and are typically given a `.dto` filename extension when compiled. Because the Device Tree Overlays were intended to provide a runtime configuration mechanism for "*Capes*", which are the name for daughter boards (or shields etc.) in the Beaglebone world, the interface mechanism is called the Cape Manager. The Cape Manager presents a nice file system based interface intended to permit manipulation of the Device Tree at runtime in a style which is analogous to the method used by the SYSFS driver to enable manipulation of the GPIOs.

The Device Tree Overlays necessarily have some of their own syntax. However stripping off the outer wrapper it can be seen they look a lot like sections of the full Device Tree.

Device Tree Overlays typically contain two sections. These sections correspond the PinMux and OCP nodes of the full Device Tree.

```
/plugin/;
/ {
fragment@0 {
        target = <&am33xx_pinmux>;
        __overlay__ {
                foo_pins: foo_pins {
                        pinctrl-single,pins = <0x0A8 0x1f>;
                };
        };
};

fragment@1 {
        target = <&ocp>;
        __overlay__ {
                foo {
                        compatible = "corp,foo";
                        bar = <5>;
                        power-gpio = <&gpio2 18 0x0>;
                        pinctrl-names = "default";
                        pinctrl-0 = <&foo_pins>;
                };
        };
    };
};
```

Credit: http://elinux.org/BeagleBone_and_the_3.8_Kernel

Knowing, as we now do, that there are two sections of the Device Tree which need configuring for any OCP device we can pretty much see what is going on here. In fact, it is also not too difficult to see how it would be possible to just copy and paste the internal sections of each fragment into the appropriate section (PinMux and OCP) of the Device Tree Source if we wished to do so (or vice versa). Most developers of Beaglebone Black Capes provide Device Tree Overlay source files – so if you are on a system without a Cape Manager, it is useful to know how to cut and paste the code into a full Device Tree File.

The Cape Manager and Device Tree Overlay system is only available on the Beaglebone Black.

The Cape Manager/Device Tree Overlay system is pretty much a Beaglebone Black invention and is not present on most other systems. Primarily it was intended to enable the easy runtime configuration required by Beaglebone Black Capes and there is even a mechanism which enables the Cape to provide its own Device Tree Overlay to the Beaglebone Black – much like the self-configuration option of USB devices on PC's. It should be noted that this self-configuration mechanism is rarely used.

One of the major benefits (to my mind) of Device Tree Overlays is that they make no permanent change to the system. This is very useful if the Overlay does not work and the Beaglebone Black locks up. In that event, all that is required is a simple reboot.

The Cape Manager is only available on the 3.8 kernel. It is not yet available on the 3.14 kernel – although work is in progress.

One problem with Device Tree Overlays is that they are very hard to implement by the system developers. To enable support for Cape Manager/Device Tree Overlays on a Linux system requires intricate modifications to many kernel source files. Simply put, the internal workings of Device Tree Overlays are very complex. This goes a long way towards explaining why Device Tree Overlays are only available on the Beaglebone Black and even then only on the 3.8 kernel. Work is underway on a port to the 3.14 kernel but as of the time this document was written it was not available yet.

The Cape Manager system sometimes has problems with device conflicts – work is also underway to develop solutions for this.

Another problem with the Cape Manager/Device Tree Overlay mechanism on the Beaglebone Black is that it can have problems with conflicts. Recall the earlier discussion of how many devices can use the same PinMux pins – if multiple Capes require the same device (or conflicting devices on the same PinMux pin) then hard to diagnose problems arise. There is a new Device Tree Overlay system under development, called Cape-Universal, which attempts to address the conflict issues. I am not sufficiently familiar with this mechanism and so will not discuss it here.

All in all the Cape Manager/Device Tree Overlay system is a pretty beautiful thing – a work of considerable skill and invention. However, it is still a work in progress and is not yet available on the later kernels. It will be very interesting to see what develops in the future.