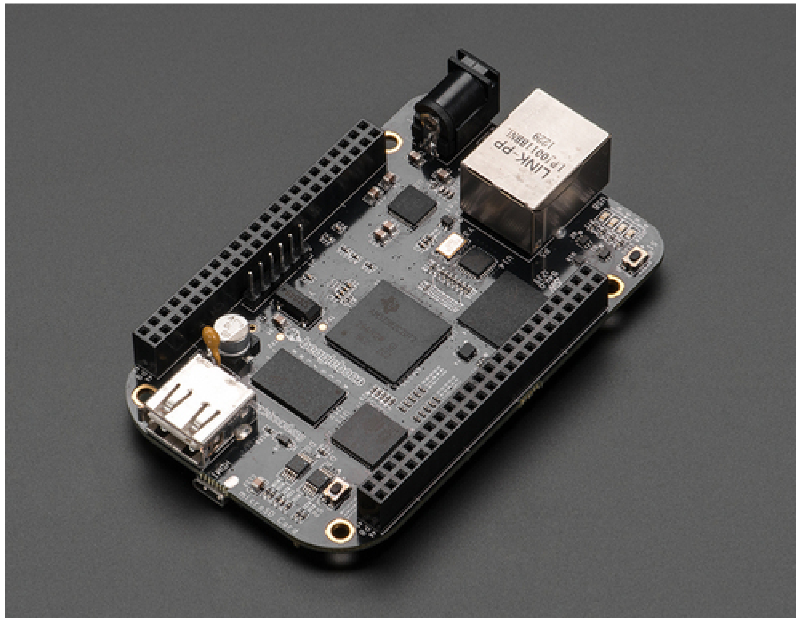
adafruit learning system

Introduction to the BeagleBone Black Device Tree

Created by Justin Cooper

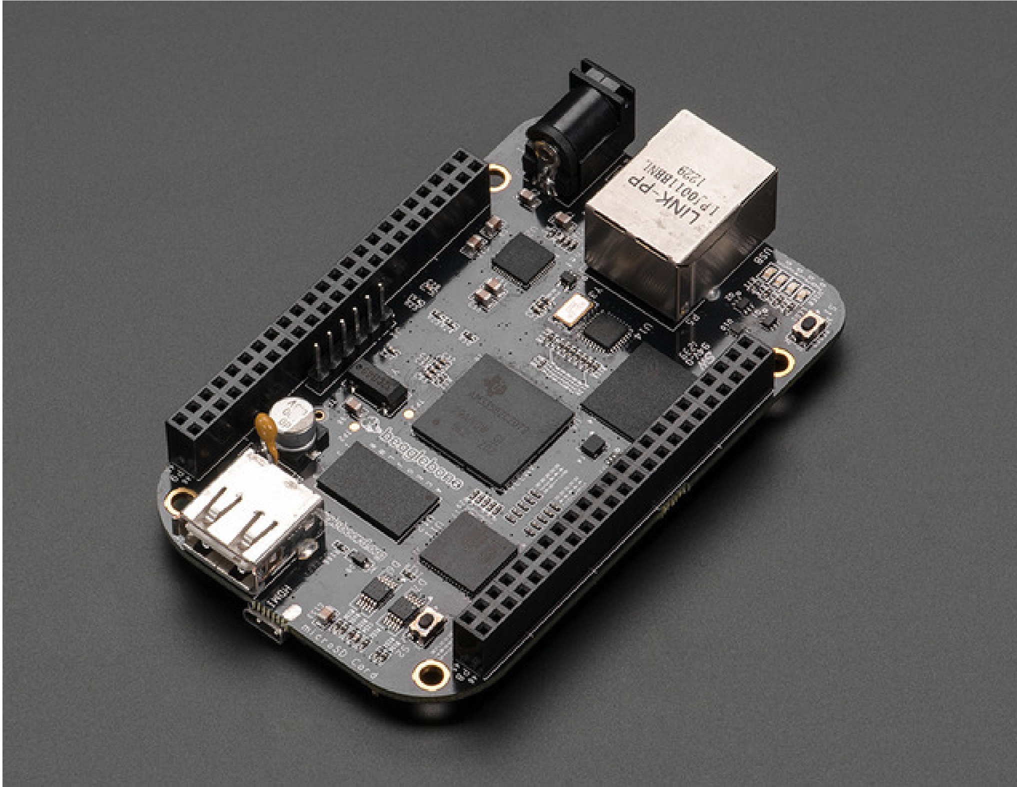


Last updated on 2018-08-22 03:36:32 PM UTC

Guide Contents

Guide Contents 2
Overview 3
Device Tree Background 4
Device Tree Overlays 5
Exporting and Unexporting an Overlay 9
Compiling an Overlay 11

Overview



The Device Tree (DT), and Device Tree Overlay are a way to describe hardware in a system. An example of this would be to describe how the UART interfaces with the system, which pins, how they should be muxed, the device to enable, and which driver to use.

The original BeagleBone didn't use the DT, but the recently released BeagleBone Black was released with the DT and is now using the 3.8 Linux Kernel.

The following pages will attempt to break down the concepts, and give examples on how and why you'd want to use the device tree in your every day development and hacking.

Device Tree Background

There is a lot of history on why the Device Tree (DT) was necessary for the BeagleBone Black. With the influx of ARM systems in the past few years, there was a lot of confusion and conflicts in the Linux kernel surrounding the ARM components, prompting Linus Torvalds to push back (https://adafru.it/clg). That push back eventually led to any new ARM boards essentially needing to use the flattened device tree, instead of ARM board files.

This sort of forced the hands of the BeagleBone Black developers, as they wouldn't be able to get a new ARM board file admitted into the mainline of the Linux kernel. They then decided to implement the DT in the latest kernel version released with the BeagleBone Black (currently using 3.8).

One issue discovered with the DT was that it wasn't designed for open embedded systems that needed to modify the system (muxing pins, enabling devices, etc) during run-time. Pantelis Antoniu implemented a solution to this issue using device tree overlays, and a cape manage (https://adafru.it/clh)r. Later, Grant Likely proposed the system (https://adafru.it/cli) that would allow for a new device tree overlay format that was a "direct extension from the existing dtb data format." allowing for modification of the DT during run-time from user-space.

More details about the history, and implementation details can be found at the following links:
https://github.com/jadonk/validation-scripts/tree/master/test-capemgr (https://adafru.it/clj)
https://docs.google.com/document/d/17P54kZkZO_-JtTjrFuVz-Cp_RMMg7GB_8W9JK9sLKfA/pub (https://adafru.it/clk)

Device Tree Overlays

Let's break down a fairly simple device tree overlay, and walk through each section in order to better understand what it's doing.

Below is the device tree overlay for the UART1 device. It tells the kernel everything it needs to know in order to properly enable UART1 on pins P9_24 and P9_26.

One thing to note, make sure you're using the latest data to figure out the register offsets, and mux modes. I've found a really good reference to be the BoneScript pin reference located in the GitHub Repository (https://adafru.it/dii).

```
/*
 * Copyright (C) 2013 CircuitCo
 *
 * Virtual cape for UART1 on connector pins P9.24 P9.26
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
/dts-v1/;
/plugin/;

/ {
    compatible = "ti,beaglebone", "ti,beaglebone-black";

        /* identification */
        part-number = "BB-UART1";
        version = "00A0";

        /* state the resources this cape uses */
        exclusive-use =
                /* the pin header uses */
                "P9.24",        /* uart1_txd */
                "P9.26",        /* uart1_rxd */
                /* the hardware ip uses */
                "uart1";

        fragment@0 {
                target = <&am33xx_pinmux>;
                __overlay__ {
                        bb_uart1_pins: pinmux_bb_uart1_pins {
                                pinctrl-single,pins = <
                                        0x184 0x20 /* P9.24 uart1_txd.uart1_txd MODE0 OUTPUT (TX) */
                                        0x180 0x20 /* P9.26 uart1_rxd.uart1_rxd MODE0 INPUT (RX) */
                                >;
                        };
                };
        };

 fragment@1 {
                target = <&uart2>; /* really uart1 */
                __overlay__ {
                        status = "okay";
                        pinctrl-names = "default";
                        pinctrl-0 = <&bb_uart1_pins>;
                };
        };
};
```

To start with, the above overlay is a "tree structure of nodes and properties (https://adafru.it/cll)".

We can break the file down to better understand it. The first section is just a comment that is optional, but usually a good idea to help folks understand what it's supposed to do, and for copyright/license information:

```
/*
 * Copyright (C) 2013 CircuitCo
 *
 * Virtual cape for UART1 on connector pins P9.24 P9.26
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 */
```

The next two lines are tokens to describe the version of the dts file, and that it's a plugin:

```
/dts-v1/;
/plugin/;
```

The next line describes the start of the root node of the DT:

```
/ {
```

The "compatible" line describes which platforms that the DT overlay is designed to work with, starting with most compatible to the least compatible. It's important to name all the platforms that you'd like to support, as it will fail to load in any platforms not mentioned.

```
compatible = "ti,beaglebone", "ti,beaglebone-black";
```

The next section, the part number and version are further guards to ensure that the proper DT overlays are loaded. In addition, they should also be used for the name of the .dts file in the form of -.dts. Also, as far as I can tell, the revision must be 00A0 on the BeagleBone Black.

```
/* identification */
part-number = "BB-UART1";
version = "00A0";
```

The exlusive-use property allows overlays to describe what resources they need, and prevents any other overlays from using those resources. In this case, we're using pins P9.24 and P9.26, as well as the uart1 device.

```
/* state the resources this cape uses */
        exclusive-use =
                /* the pin header uses */
                "P9.24",        /* uart1_txd */
                "P9.26",        /* uart1_rxd */
                /* the hardware ip uses */
                "uart1";
```

Next up are the device tree fragments. These will describe which target to overlay, and then each fragment will be customized to either mux pins, or enable devices. The below fragment looks fairly complicated, but it's not too bad. The first thing is that we're setting what the target for this fragment is going to overlay. In this case, it's the am33x_pinmux, which is compatible with the pinctrl-single driver. Examples on how to use this driver are found on the pinctrl-single documentation page (https://adafru.it/clm).

The next line is the __overlay__ node itself. The first property within that node will be used in the next fragment (bb_uart1_pins), and contains the definition for muxing the pins, which is handled by the pinctrl-single driver. You can find out how to set that by viewing the documentation page. In particular, this is the section that describes how it should be setup:

- The pin configuration nodes for pinctrl-single are specified as pinctrl register offset and value pairs using pinctrl-single,pins. Only the bits specified in pinctrl-single,function-mask are updated. For example, setting a pin for a device could be done with: pinctrl-single,pins = <0xdc 0x118>; Where 0xdc is the offset from the pinctrl register base address for the device pinctrl register, and 0x118 contains the desired value of the pinctrl register.

Within the pinctrl-single, pins value block, the two rows of values are setting the pins for the UART1 device. In this case, it's P9_24 as MODE0, which will be an OUTPUT (TX). As well as P9_26 as MODE0 an INPUT (RX). You can tell that P9_24 needed to be set to 0x184 by cross-referencing the pin reference for BoneScript (https://adafru.it/dj1) (there are more around the web, but some are out of date).

```
fragment@0 {
                target = <&am33xx_pinmux>;
                __overlay__ {
                        bb_uart1_pins: pinmux_bb_uart1_pins {
                                pinctrl-single,pins = <
                                        0x184 0x20 /* P9.24 uart1_txd.uart1_txd MODE0 OUTPUT (TX)  */
                                        0x180 0x20 /* P9.26 uart1_rxd.uart1_rxd MODE0 INPUT (RX)  */
                                >;
                        };
                };
        };
```

The last fragment enables the uart device. The target for the overlay is uart2 (which is commented as actually being uart1). It also references the previous fragments property (bb_uart1_pins) to map the pins enabled by the pinctrl driver to the uart device.

```
 fragment@1 {
                target = <&uart2>; /* really uart1 */
                __overlay__ {
                        status = "okay";
                        pinctrl-names = "default";
                        pinctrl-0 = <&bb_uart1_pins>;
                };
        };
```

This may seem quite overwhelming, and really obscure, and it kind of is. The easiest way to create a new overlay for what you want done is to start with one that is already close to what you want to do. The best place to look for that is in /lib/firmware (on Angstrom), and peruse the overlays already created for you to use.

Exporting and Unexporting an Overlay

The following examples are demonstrated using the default Angstrom distribution.

To start with, navigate into /lib/firmware to view all of the device tree overlays available by default.

```
root@beaglebone:~# cd /lib/firmware
root@beaglebone:/lib/firmware# ls -ltr
total 888
...
```

You'll see quite a few overlays available. The source (dts), and compiled (dtbo) files are both in that directory.

Open any of the source files to view their contents. The overlays themselves are fairly descriptive, and well commented.

Let's use the BB-UART1-00A0.dts that we walked through on the previous page.

We could view the .dts file, but we already know what it contains. Let's take advantage of the fact that there's already a compiled .dtbo file available to us. We'll build and compile one of our own in a future section.

Ok, let's next navigate to where we can view which overlays are enabled by the bone cape manager:

```
root@beaglebone:/lib/firmware# cd /sys/devices/bone_capemgr.*
```

Note above, the * is necessary as we don't know what that number can be. It depends on the boot order. I've seen the path as /sys/devices/bone_capemgr.8/slots, as well as /sys/devices/bone_capemgr.9/slots. You'll need to determine what your particular path is.

Next up, cat the contents of the slots file:

```
root@beaglebone:/sys/devices/bone_capemgr.8# cat slots
```

It should look something like this, assuming you haven't customized your Angstrom installation very much:

```
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:PF---
4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
```

According to the BeagleBone documentation, "the first 3 slots are assigned by EEPROM IDs on the capes". The next two are overlays loaded at boot. Number 4 is the EMMC memory built in that you're mostly likely booting your Angstrom distribution from. The 5th overlay is for enabling the HDMI component.

If you were to now export another overlay, such as our favorite UART1 overlay, you would see a new option listed as number 6. Let's try that by exporting the UART1 dtbo file:

```
root@beaglebone:/sys/devices/bone_capemgr.8# echo BB-UART1 > slots
```

We're taking the output of echo, "BB-UART1", and writing it to the slots file to enable the drivers and device for UART1 using the overlay. Now, let's check that the overlay loaded properly:

```
root@beaglebone:/sys/devices/bone_capemgr.8# cat slots
```

We should now have the UART1 device loaded up, and ready to go:

```
0: 54:PF---
1: 55:PF---
2: 56:PF---
3: 57:PF---
4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
6: ff:P-O-L Override Board Name,00A0,Override Manuf,BB-UART1
```

Now, let's say you're done with using the UART1 device, and need those pins for something else. One way to remove the overlay would be to restart your BeagleBone. The other way would be to unexport it.

You can export it by executing the following command:

```
root@beaglebone:/sys/devices/bone_capemgr.8# echo -6 > slots
```

We took the 6th listed option, and put a '-' before it, and wrote it to the slots file.

One thing to note. As of the 6-20-2013 release of Angstrom, unloading various overlays can cause a kernel panic, and cause you to lose your ssh session, along with making the capemgr unpredictable. It's recommended to just restart your system to unload overlays until that issue is resolved.

Now that we know that restarting the system will cause the overlays to unload, how do we have them loaded when the system boots up?

This is fairly simple to do. All you need to do is reference them in the uEnv.txt in the small FAT partition on your BeagleBone Black.

The following steps illustrate how to do this for the UART1 overlay:

```
mkdir /mnt/boot
mount /dev/mmcblk0p1 /mnt/boot
nano /mnt/boot/uEnv.txt
#append this to the end of the single line of uEnv.txt (not on a new line):
capemgr.enable_partno=BB-UART1
```

Compiling an Overlay

Now that we've manually enabled an overlay, let's take an overlay created and used in the Adafruit BBIO Python library for enabling the SPI0 device and bus. You can find the latest copy of this overlay at the Adafruit Github repository for Adafruit_BBIO (https://adafru.it/clo).

Below is a copy of the overlay for SPI0 that we'll use to compile, and enable:

```
/dts-v1/;
/plugin/;

/ {
    compatible = "ti,beaglebone", "ti,beaglebone-black";

    /* identification */
    part-number = "ADAFRUIT-SPI0";

    /* version */
    version = "00A0";

    fragment@0 {
        target = <&am33xx_pinmux>;
        __overlay__ {
            spi0_pins_s0: spi0_pins_s0 {
                pinctrl-single,pins = <
                  0x150 0x30 /* spi0_sclk, INPUT_PULLUP | MODE0 */
                  0x154 0x30 /* spi0_d0, INPUT_PULLUP | MODE0 */
                  0x158 0x10 /* spi0_d1, OUTPUT_PULLUP | MODE0 */
                  0x15c 0x10 /* spi0_cs0, OUTPUT_PULLUP | MODE0 */
                >;
            };
        };
    };

    fragment@1 {
        target = <&spi0>;
        __overlay__ {
             #address-cells = <1>;
             #size-cells = <0>;

             status = "okay";
             pinctrl-names = "default";
             pinctrl-0 = <&spi0_pins_s0>;

             spidev@0 {
                 spi-max-frequency = <24000000>;
                 reg = <0>;
                 compatible = "spidev";
             };
             spidev@1 {
                 spi-max-frequency = <24000000>;
                 reg = <1>;
                 compatible = "spidev";
             };
        };
    };
};
```

Quickly looking at the above overlay, we can see it follows the same pattern as the overlay for UART1. there are two fragments, with the first one utilizing the pinctrl-single driver to mux the pins. We're using 4 of the SPI0 pins, setting everything to Mode 0. Then, fragment@1 is targeting the spi0 device with the proper pins, and setting various parameters specific to using the spi device, such as the 'spi-max-frequency'. You can see the available options in the spi-bus documentation (https://adafru.it/clp).

Navigate to your home directory, and open nano to copy and paste the new file. You'll need to save it exactly as named below as well:

```
root@beaglebone:/tmp# cd ~
root@beaglebone:~# nano ADAFRUIT-SPI0-00A0.dts
```

Next, we'll execute the command to compile this file into the device tree overlay compiled format (.dtbo):

```
dtc -O dtb -o ADAFRUIT-SPI0-00A0.dtbo -b 0 -@ ADAFRUIT-SPI0-00A0.dts
```

The compilation should be nearly instant, and you should end up with the newly compiled file:

```
root@beaglebone:~# ls -ltr
...
-rw-r--r-- 1 root root 1255 Jul 29 14:33 ADAFRUIT-SPI0-00A0.dts
-rw-r--r-- 1 root root 1042 Jul 29 14:35 ADAFRUIT-SPI0-00A0.dtbo
```

Let's break down the options used to compile the overlay.

To start with we're using the device tree compiler (dtc). Everything required to compile DT overlays are included with the latest Angstrom distribution.

-O dtb is the output format. We're outputting device tree binaries.
-o is the output filename.
-b 0 is setting the physical boot CPU. (a zero)
-@ generates a symbols node as part of the dynamic DT loading of the overlay

You'll know if you don't have a new enough version of dtc if the compiler complains about the missing -@ flag. You can attempt an upgrade of dtc by executing the following (this may need to be done on Ubuntu for now):

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