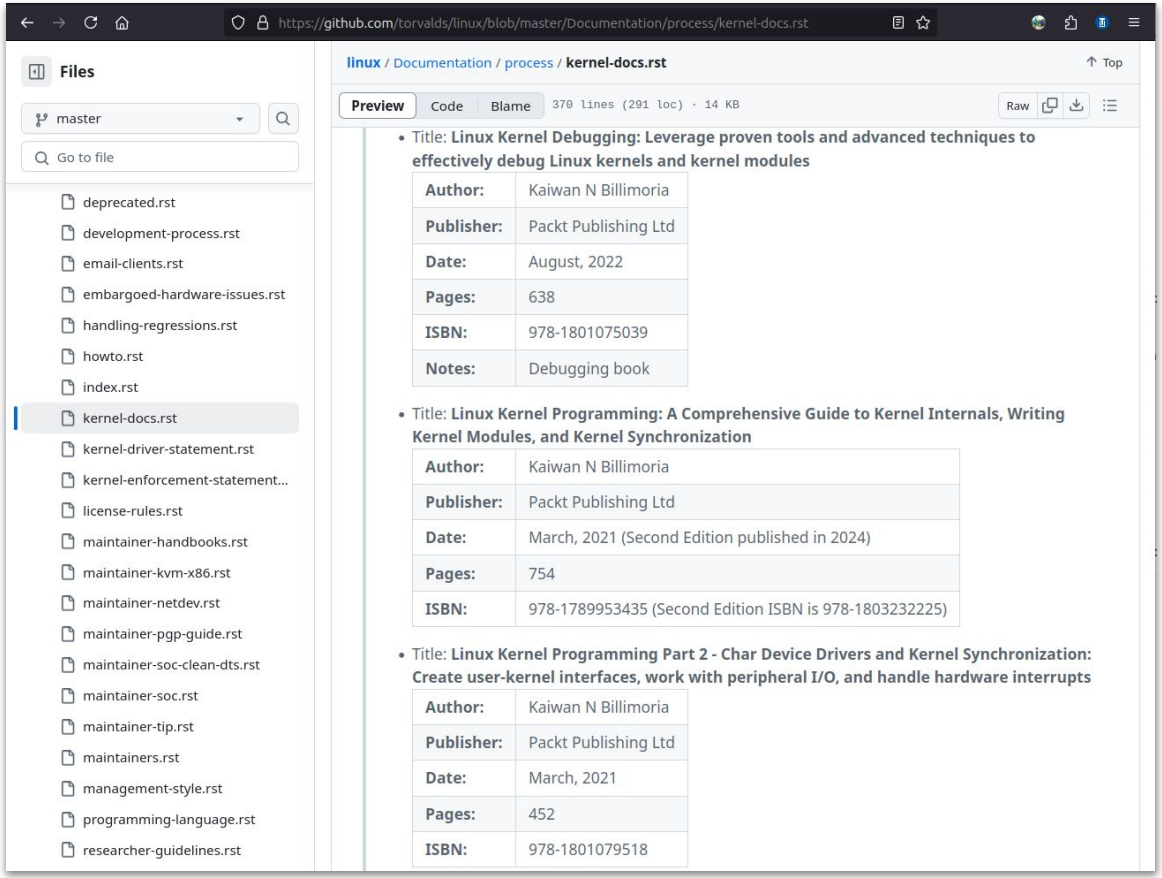
Writing your first Device Tree (DT) Overlay – it's really quite easy!

Kaiwan N Billimoria

whoami

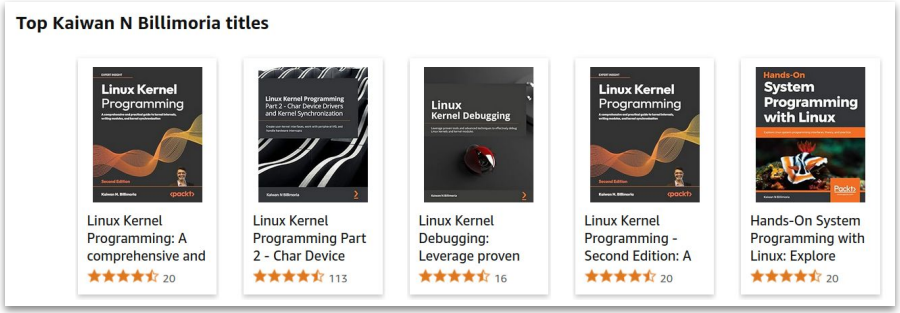
I’m the author of a few of books on Linux (see next). Am happy that a few are listed in the *Published Books* section of the official kernel docs: [linux/Documentation/process/kernel-docs.rst at master]



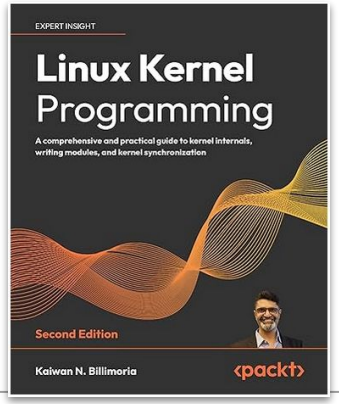
whoami

- Kaiwan N Billimoria: profile all-in-one : https://bit.ly/m/kaiwan

My Books!



★ LinkedIn public profile
★ My Amazon Author page
★ Corporate Training on Linux
  - Brief
  - More detailed
★ My GitHub repos
★ My tech blog



An offer to all:
25% off on the popular *Linux Kernel Programming, 2nd Ed* book!
https://packt.link/CvDQO
23 Nov to 31 Dec 2024

Device Tree - What

- Wikipedia has a one-liner for “Device Tree”: *(it’s a) mechanism for passing hardware information*

- A meta-language or grammar specified by the Open Firmware (OF) project

- Systems that use the Device Tree (DT)
  - originated on PowerPC (PPC)
  - heavily used in ARM (AArch32) and ARM64 (AArch64)
  - the typical PC (x86[_64]) doesn’t use it (as of now); they instead traditionally rely on auto-config via UEFI/BIOS, self-enumerating buses (PCI[e]) and ACPI. (There are exceptions: some x86 platforms using coreboot use a DT)
  - the Linux kernel can parse DTs for the following arch’s:
    ARC, ARM, C6x, H8/300, MicroBlaze, MIPS, NDS32, Nios II, OpenRISC, PowerPC, RISC-V, SuperH, and Xtensa
  - All ARM-based Android devices use the DT

Device Tree - Why?

- ARM has always supported an enormous number of boards (or platforms); IOW, many boards use ARM core(s)
- Traditionally this requires board-specific source code particular to each board type, thus resulting in a board-specific kernel
  - this is the case even when the variations between some boards are extremely minor
  - as well, it's not just the kernel image that can be board-specific; the bootloader (typically U-Boot) too can be
- Think of large Android OEMs; they may well have dozens of models of an Android device with very minor variations, which ultimately become different commercial models. Maintaining a separate bootloader and kernel for each of them will quickly become a nightmare!

Device Tree - Why?

- Think of large Android OEMs; they may well have dozens of models of an Android device with very minor variations, which ultimately become different commercial models. Maintaining a separate bootloader and kernel for each of them will quickly become a nightmare!

- *The DT to the rescue!*
- With the DT, hardware or board-specific details - all the minor variances - are moved into the DT for each model, which is quite easily maintainable
- (As we shall see, the *DT Overlay* concept further simplifies this)

Device Tree - How?

Q. Where's the DT source?
A. in Device Tree Source (DTS) files

- The DT isn't code, it's a 'hardware description' grammar (somewhat analogous to VHDL/XML/...) defined by the Open Firmware (OF) spec

- Where exactly are the DT source files?
  - in the kernel source tree
    - `arch/<arch>/boot/dts` (where <arch> is the CPU family)
  - from 6.5, even AArch32 - just like AArch64 from earlier - has vendor-specific directories underneath to make it more manageable/readable
  - F.e. for arch 'arm' (AArch32) on 6.12:

```
$ ls arch/arm/boot/dts/
actions/      armv7-m.dtsi   cros-adc-thermistors.dtsi  Makefile    nvidia/     sigmastar/     tps65217.dtsi
airoha/       aspeed/        cros-ec-keyboard.dtsi      marvell/    nxp/        socionext/     tps65910.dtsi
allwinner/    axis/          cros-ec-sbs.dtsi           mediatek/   qcom/       st/            unisoc/
alphascale/   broadcom/      gemini/                    microchip/  realtek/    sunplus/       vt8500/
amazon/       calxeda/       hisilicon/                 moxa/       renesas/    synaptics/     xen/
amlogic/      cirrus/        hpe/                       nspire/     rockchip/   ti/            xilinx/
arm/          cnxt/          intel/                     nuvoton/    samsung/    tps6507x.dtsi
$
```

Device Tree - How?

The source : in Device Tree Source (DTS) files

- The DTS files get compiled into DTB (or DTBO) binary blobs; these are what's passed to the bootloader and/or kernel, and what's parsed in by the kernel at boot (via of*() routines; **OF = Open Firmware**)
- The Device Tree Compiler - a tool named **'dtc'** - compiles the DT source to a DT binary 'blob'
- The full DTS can have multiple source files (typically due to the `#include` directive, just as in C)
- Example: the DTS of the TI Beagle Bone Black (BBB) platform begins this way:

```
/dts-v1/;
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
#include "am335x-boneblack-common.dtsi"
#include "am335x-boneblack-hdmi.dtsi"

/ {
      model = "TI AM335x BeagleBone Black";
      compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
}; [ … ]
```

- It can thus quickly get quite complex to read in at a glance…

Device Tree - How?

The source : in Device Tree Source (DTS) files

- It can thus get complex to read in at a glance...
- Can leverage the fact that the board device tree is 'exposed' via sysfs here: `/sys/firmware/devicetree/` (or via the soft link: `/proc/device-tree -> /sys/firmware/devicetree/base/`)
- So, a trick/tip (though it has its limitations): on the board, do this:

```
dtc -I fs -@ /proc/device-tree -O dts -o myboard.dts 2>/dev/null
```

```
-I, --in-format <arg>
Input formats are:
       dts - device tree source text
       dtb - device tree blob
       fs  - /proc/device-tree style directory

-@, --symbols          Enable generation of symbols

-o, --out <arg>        Output file
-O, --out-format <arg>
Output formats are:
       dts - device tree source text
       dtb - device tree blob
       yaml - device tree encoded as YAML
       asm - assembler source
```

Device Tree - How

- Important to realize that the device tree isn't code and will NOT drive devices; for that, we still very much require the device driver!

- The DT, though, plays a key role in binding the correct driver to a specified device (via the '`compatible`' property)

- For more detailed coverage on the DT, please refer to the following open source PDF doc:

  'ABOUT THE DEVICE TREE' (credit: ofitselfso.com)

  (it's provided within the participant courseware as well).

Device Tree Blob Overlay (DTBO) - Why

- DT overlays are a means to add to or edit the existing board DT without directly modifying and re-compiling it

- Advantages this brings:
  - Makes it significantly easier as the work is typically on a small portion of the large DTS(s)
  - Flexibility
  - Great for:
    - products with many models with minor variations between them - can use a single base DT and several DT overlays to specify differences / tweaks
    - maintaining several different configurations (or 'presets') of a product; end-user can use a particular one as required
    - performing quick mods (f.e. changing the `status` property of some node)
    - dev / evaluation chips that connect to the board via GPIO header pins
    - add-on boards/capes/HATs
  - *(More from Toradex)*

Device Tree Blob Overlay (DTBO) - How

- 'Official' kernel documentation is very important: https://elixir.bootlin.com/linux/v6.11.2/source/Documentation/devicetree/overlay-notes.rst

- Here's a very simple yet complete DTO source file (works on the TI BBB - BeagleBone Black):

```
/dts-v1/;
/plugin/;
&{/chosen} {
     overlays {
               BB-TEST-OVERLAY.kernel = "Tue Oct 14 22:31:00 2000";
     };
};
&{/ocp} {
     /* our (pseudo) demo on-chip peripheral (ocp) named 'my_device' */
     my_device {
          compatible = "mycorp,mydev";
          status = "okay";
          label = "Test";
          my_value = <12>;
     };
};
```

Worry not, the next few slides will explain it

Device Tree Blob Overlay (DTBO) - How

- The DTBO header:

```
/dts-v1/;

/plugin/;
```

  - Specifies the DTS version (as v1)
  - `/plugin/;` this specifies that it's a DT**O** source file and not a full DTS
  - Can specify the DT overlay source file as **foo.dto** (as opposed to foo.dts although both work)

Device Tree Blob Overlay (DTBO) - How

- The **chosen** node:
  - Typically represents the kernel command line - params to pass at boot to the kernel (via the bootloader)
  - Here, though, the `chosen` node acts as a helper enabling one **to see loaded overlays under the DT**, like this:

    `ls /proc/device-tree/chosen/overlays/`

```
&{/chosen} {
        overlays {
                BB-TEST-OVERLAY.kernel = "Tue Oct 14 22:31:00 2000";
                /* This should be '__TIMESTAMP__' but that's defined in kernel src only..
                 * So have simply hardcoded it here to some timestamp */
        };
};
```

- This is useful, enabling one to check whether the new device/chip/whatever has actually been added to the board DT at runtime

Device Tree Blob Overlay (DTBO) - How

- Two broad ways in which one can specify the peripheral device/chip:
  - One, the traditional (older) approach where its specified as a **fragment** and a 'target path' *to the node to overlay*
  - Two, where the overlay target location is explicitly specified by label (modern, preferred)
- First approach: traditional/older example DT overlay snippet:

```
/ {
        compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
        fragment@0 {
                target-path = "/";
                __overlay__ {
                        my_device {
                                compatible = "corp,bar";
                                /* … various properties and child nodes go here … */
                        };
                };
        };
};
```

Eg. https://github.com/beagleboard/bb.org-overlays/blob/master/src/arm/BB-I2C1-BME280.dts

Device Tree Blob Overlay (DTBO) - How

- Two broad ways in which one can specify the peripheral device/chip:
  - One, the traditional (older) approach where its specified as a **fragment** and a 'target path' *to the node to overlay*
  - Two, where the overlay target location is explicitly specified by label (modern, preferred)

- Second approach, modern - target location by label - example DT overlay snippet:

```
…
&{/ocp} {
        /* our (pseudo) demo on-chip peripheral (ocp) named 'my_device' */
        my_device {
             compatible = "corp,bar";
             /* … various properties and child nodes go here … */
        };
};
…
```

Device Tree Blob Overlay (DTBO) - How

Second approach - same effect, less complicated! Preferred as “overlay can be applied to any base DT containing the label, no matter where the label occurs in the DT.”

```
&{/ocp} {
  /* our (pseudo) demo on-chip peripheral (ocp) named 'my_device' */
        my_device {
                compatible = "mycorp,mydev";
                status = "okay";
                label = "Test";
                my_value = <12>;
        };
};
```

- The ‘`&{/ocp}`’ specifies the overlay target location by explicit path
  - In this example, the TI BBB board has a DT node named **‘ocp’** - short for ‘**on-chip peripherals**’
  - So here we’re specifying this node as the ‘target’ for our new device

Device Tree Blob Overlay (DTBO) - How

Modern approach - same effect, less complicated!

```
&{/ocp} {
      /* our (pseudo) demo on-chip peripheral (ocp) named 'my_device' */
      my_device {
            compatible = "mycorp,mydev";
            status = "okay";
            label = "Test";
            my_value = <12>;
      };
};
```

- We specify a new node for our new device / peripheral / whatever
- here we've named it `my_device`
- the `compatible` property of course is critical, allowing the kernel (bus driver) to bind the new device to a (client) driver - which will use the very same `compatible` property string - to match and thus drive it
- Above, we specify a few sample properties for our device

Device Tree Blob Overlay (DTBO) - How

Trying out our simple DTB Overlay on the TI BeagleBone Black !

- *<The examples below are of course wrt the TI BBB board only>*
- Generate the DTBO - the DT Blob Overlay - file, from source (.dto) using the dtc compiler:

```
dtc -@ -I dts -O dtb -o BBB-TESTOVERLAY.dtbo bbb_testoverlay.dto
```

- Copy across the DTBO file to the `/boot/dtbs/$(uname -r)/overlays` directory
- Enable it by inserting these lines in **`/boot/uEnv.txt`** :

```
enable_uboot_overlays=1
dtb_overlay=/boot/dtbs/5.10.168-ti-r71/overlays/BBB-TESTOVERLAY.dtbo
```

Device Tree Blob Overlay (DTBO) - How

TIP: To specify (more than one) custom DTBO's, you can make use of the **`/boot/uEnv.txt`** file's following lines:

```
dtb_overlay=/boot/dtbs/$(uname -r)/BBB-TESTOVERLAY.dtbo
[ … ]
###Additional custom capes
#uboot_overlay_addr4=</path/to/my2.dtbo>
[ … ]
#uboot_overlay_addr7=</path/to/my5.dtbo>
[ … ]
###Custom Cape #dtb_overlay=...
#uboot_overlay_addr0=<file0>.dtbo
[ … ]
#uboot_overlay_addr3=<file0>.dtbo
```

1 DT overlay here

Up to 4 DT overlays here

Up to 4 DT overlays here; overrides the 'auto detect' overlays though

So we can specify up to nine DTBO's in effect!

Of course, you must ensure the DT overlays do not conflict in any manner.. (pin muxing, etc)

Device Tree Blob Overlay (DTBO) - How

- A very simple demo is available here: https://github.com/kaiwan/drv_johannes/tree/main/20_dt_probe
  - (It's forked from here: Johannes4Linux/Linux_Driver_Tutorial:main)

- You can try this very simple DT overlay on either the TI BBB or the Raspberry Pi family boards

- *Do try it out!*

DT Overlay : sample run

Sample run on a TI BBB (32-bit)

- Copy across the DTBO file to
  `/boot/dtbs/$(uname -r)/overlays`
- `/boot/uEnv.txt`: the line `enable_uboot_overlays=1` must be there
- Enable it by inserting a line in `/boot/uEnv.txt` (so that U-Boot recognizes it as a DTBO to load at boot):
  `dtb_overlay=/boot/dtbs/5.10.168-ti-r71/overlays/BBB-TESTOVERLAY.dtbo`
- Until around 2017, the 'Cape Manager' driver was the way to add/remove DTBOs; nowadays it's deprecated and the 'way' is to simply edit uEnv.txt
- Also: "In recognition of the fact that some people may wish to change the PinMux Mode dynamically (via the command line) at runtime, the **config-pin** utility was developed. The config-pin command enables the root user to change the PinMux mode - but it will not cause the relevant Device Drivers for the OCP Devices to be loaded. Changes made with the config-pin utility are not permanent and will need to be re-applied after each reboot. "

```
bbb $ uname -r
5.10.168-ti-r71
bbb $ ls
bbb_testoverlay.dto  bkp/  dt_probe.c  Makefile  README  rpi_testoverlay.dto  run_on_rpi*
bbb $
bbb $ make
FNAME_C=dt_probe

--- Building : KDIR=/lib/modules/5.10.168-ti-r71/build ARCH= CROSS_COMPILE= ccflags-y="-UDEBUG -Wall -
DULE" MYDEBUG=n DBG_STRIP=y ---
gcc (Debian 10.2.1-6) 10.2.1 20210110

make -C /lib/modules/5.10.168-ti-r71/build M=/home/debian/kaiwanTECH/drv_johannes/20_dt_probe modules
make[1]: Entering directory '/usr/src/linux-headers-5.10.168-ti-r71'
FNAME_C=dt_probe
  CC [M]  /home/debian/kaiwanTECH/drv_johannes/20_dt_probe/dt_probe.o
FNAME_C=dt_probe
  MODPOST /home/debian/kaiwanTECH/drv_johannes/20_dt_probe/Module.symvers
  CC [M]  /home/debian/kaiwanTECH/drv_johannes/20_dt_probe/dt_probe.mod.o
  LD [M]  /home/debian/kaiwanTECH/drv_johannes/20_dt_probe/dt_probe.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.10.168-ti-r71'
if [ "y" = "y" ]; then \
   strip --strip-debug dt_probe.ko ; \
fi
--- compiles the Device Tree Blob (DTB) [Overlay] from the DTS (ARM/PPC/etc) ---
dtc -@ -I dts -O dtb -o BBB-TESTOVERLAY.dtbo bbb_testoverlay.dto
dtc -@ -I dts -O dtb -o rpi_testoverlay.dtbo rpi_testoverlay.dto
# DTBO - Device Tree Blob Overlay
bbb $
bbb $ sudo cp BBB-TESTOVERLAY.dtbo /boot/dtbs/5.10.168-ti-r71/overlays/
bbb $ # now edit /boot/uEnv.txt adding this DTB overlay ...
bbb $ grep "^dtb_overlay=" /boot/uEnv.txt
dtb_overlay=/boot/dtbs/5.10.168-ti-r71/overlays/BBB-TESTOVERLAY.dtbo
bbb $
bbb $ sudo dmesg -C; sudo insmod ./dt_probe.ko ; sudo dmesg
[  232.561043] Loading the driver...
[  232.561342] my_device_driver ocp:my_device: dt_probe:dt_probe(): in the probe function!
[  232.561361] my_device_driver ocp:my_device: dt_probe:dt_probe(): label: demo device
[  232.561376] my_device_driver ocp:my_device: dt_probe:dt_probe(): my_value: 12
bbb $
```

DT Overlay : sample run

Sample run on a Raspberry Pi 4 Model B (64-bit)

Via the run_on_rpi helper script

On the R Pi, the script invokes `sudo dtoverlay rpi_testoverlay.dtbo` *to trigger it*

```
20_dt_probe $ lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 11 (bullseye)
Release:        11
Codename:       bullseye
20_dt_probe $ id
uid=1000(pi) gid=1000(pi) groups=1000(pi),4(adm),20(dialout),24(cdrom),27(sudo),29(audio),44(video),46(plugdev),60(games),
100(users),104(input),106(render),108(netdev),997(gpio),998(i2c),999(spi)
20_dt_probe $ uname -r
6.1.21-v8+
20_dt_probe $
20_dt_probe $ ls
bbb_testoverlay.dts  dt_probe.c  Makefile  rpi_testoverlay.dts  run_on_rpi*
20_dt_probe $
20_dt_probe $ ./run_on_rpi
make -C /lib/modules/6.1.21-v8+/build M=/home/pi/kaiwanTECH/drv_johannes/20_dt_probe modules
make[1]: Entering directory '/usr/src/linux-headers-6.1.21-v8+'
  CC [M]  /home/pi/kaiwanTECH/drv_johannes/20_dt_probe/dt_probe.o
  MODPOST /home/pi/kaiwanTECH/drv_johannes/20_dt_probe/Module.symvers
  CC [M]  /home/pi/kaiwanTECH/drv_johannes/20_dt_probe/dt_probe.mod.o
  LD [M]  /home/pi/kaiwanTECH/drv_johannes/20_dt_probe/dt_probe.ko
make[1]: Leaving directory '/usr/src/linux-headers-6.1.21-v8+'
dtc -@ -I dts -O dtb -o rpi_testoverlay.dtbo rpi_testoverlay.dts
dtc -@ -I dts -O dtb -o bbb_testoverlay.dtbo bbb_testoverlay.dts
Built Device Tree Overlay and kernel module

sudo dmesg
[10528.378867] Loading the driver...
[10528.379153] my_device_driver my_device: dt_probe:dt_probe(): in the probe function!
[10528.379165] my_device_driver my_device: dt_probe:dt_probe(): dt_probe - label: Test
[10528.379173] my_device_driver my_device: dt_probe:dt_probe(): my_value: 12
```

DT Overlay for the DHT2x temperature + humidity sensor chip (on the TI BBB)

The DT Overlay .dto file for the DHT22 sensor chip on the TI BBB

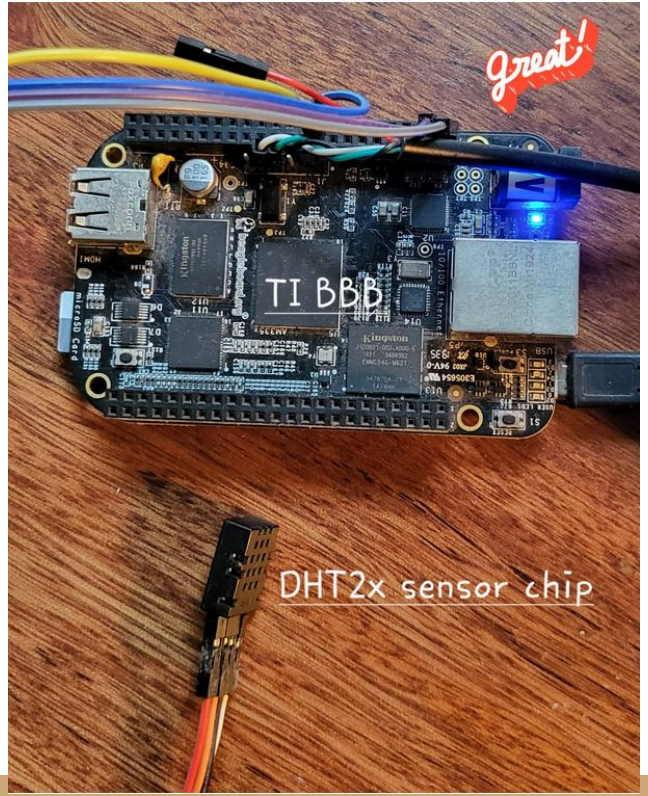
```
/dts-v1/;
/plugin/;

&{/chosen} {
        overlays {
            BB-I2C2-DHT2X.kernel = "Tue Oct 14 22:31:00 2000";
        };

};

&i2c2 {
        status = "okay";
        clock-frequency = <100000>;
        #address-cells = <1>;
        #size-cells = <0>;

        dht22@38 {
                compatible = "asair,dht2x_kdrv";
                reg = <0x38>;
                status = "okay";
        };
};
```



Code

DT Overlay for the DHT2x temperature + humidity sensor chip

Sample run on a TI BBB (32-bit)

- Copy across the DTBO file to `/boot/dtbs/$(uname -r)/overlays`
- `/boot/uEnv.txt`: the line `enable_uboot_overlays=1` must be there
- Enable our DT overlay by inserting a line this into `/boot/uEnv.txt`: `dtb_overlay=/boot/dtbs/5.10.168-ti-r71/overlays/BBB-I2C2-DHT2X.dtbo`
- Code

```
bbb $ grep I2C2-DHT2X /boot/uEnv.txt
uboot_overlay_addr4=/boot/dtbs/5.10.168-ti-r71/overlays/BBB-I2C2-DHT2X.dtbo
bbb $ ls /proc/device-tree/chosen/overlays/
BB-ADC-00A0.kernel          BB-I2C2-DHT2X.kernel    name
BB-HDMI-TDA998x-00A0.kernel  BB-TEST-OVERLAY.kernel
bbb $ # ah it's loaded up!
bbb $
bbb $ show-pins | grep -E "P9.19|P9.20"
P9.20 / cape i²c sda              94 fast rx  up  3 i²c 2 sda        ocp/P9_20_pinmux (pinmux_P9_20_default_pin)
P9.19 / cape i²c scl              95 fast rx  up  3 i²c 2 scl        ocp/P9_19_pinmux (pinmux_P9_19_default_pin)
bbb $
bbb $ grep "compatible" bbb_dts/bbb_i2c2_dht2x.dts
                    compatible = "asair,dht2x_kdrv";
bbb $ grep "compatible.*dht" dht2x_kdrv.c
        {.compatible = "asair,dht2x_kdrv"},
bbb $
bbb $ # they match
bbb $
bbb $ sudo insmod ./dht2x_kdrv.ko
bbb $ dmesg |tail -n5
[ 1181.364980] dht2x_kdrv 2-0038: hey, in probe! name=dht2x_kdrv addr=0x38
[ 1181.470979] dht2x_kdrv 2-0038: chip found
bbb $ ./disp_temp_humd.sh
./disp_temp_humd.sh: line 12: warning: command substitution: ignored null byte in input
+++ Detected we're running on the TI AM335x BeagleBone Black
temperature(milliC),rel_humidity(milli%)
24545,84332
24545,84290
24551,84280
^C
bbb $
```

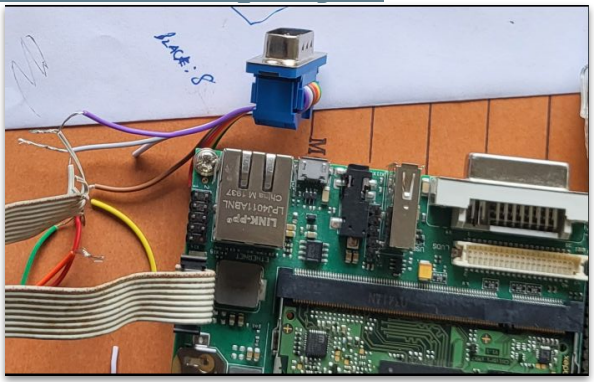
Other sample DT Overlays

- I have a Colibri IMX7 SoM 1 GB RAM, eMMC (SoC: i.MX7 from Freescale) with an IRIS Rev2 Carrier Board from **Toradex**
- They provide a Yocto BSP (and others as well)

- They make several DT overlays available:

```
git clone  git://git.toradex.com/device-tree-overlays.git
```

- A couple of DT overlays that they provide is shown...



Other sample DT Overlays

- From **Toradex** ; example 1 of 2:

```
$ cat overlays/colibri-imx7_ad7879_overlay.dtsi
// SPDX-License-Identifier: GPL-2.0-or-later OR MIT
/*
 * Copyright 2020-2022 Toradex
 */

// Resistive AD7879 touch controller on the Colibri iMX7 for
the 7"
// display orderable at Toradex.

&ad7879_ts {
    status = "okay";
};
```

Other sample DT Overlays

From **Toradex** ; example 2 of 2 (structured as a DTS include file, a .dtsi):

```
$ cat overlays/display-vga-640x480_overlay.dtsi
[ … ]
// VGA Signal 640x480@60Hz Industry standard timing
&panel_dpi {
	compatible = "panel-dpi";
	status = "okay";

	/* for 0.3mm pixels */
	width-mm = <192>;
	height-mm = <144>;

	panel-timing {
		clock-frequency = <25175000>;
		hactive = <640>;
		vactive = <480>; hsync-len = <96>;
		hfront-porch = <16>;
		hback-porch = <48>;
		vsync-len = <2>;
		vfront-porch = <10>;
		vback-porch = <33>;
		hsync-active = <0>;
		vsync-active = <0>;
		pixelclk-active = <0>;
	};
};
```

Device Tree Blob Overlay (DTBO)

Useful resources

- The Beaglebone Black and Device Tree Overlays
- Device Tree Overlays Technical Overview, Toradex
- Using Device Tree Overlays, example on BeagleBone Cape add-on boards, M Opdenacker, Feb 2022
- Existing DT overlays source for the TI BBB
- https://elinux.org/Beagleboard:BeagleBone_cape_interface_spec#LEDs
- Delving further.. see the "*Virtual cape for LED on P8_03*" overlay here: https://git.beagleboard.org/beagleboard/BeagleBoard-DeviceTrees/blob/v4.19.x-ti-overlays/src/arm/overlays/BONE-LED_P8_03.dts

Related - a brief on Pin muxing

Useful resources
About the Beaglebone Black PinMux Modes

- Modern SoCs have so many on-chip / OCP devices or IP blocks (the ‘cold’ ones, they can’t be detected at boot & hence they’re platform devices in the DT)
- Eg. multiple GPIO's, SPI, I2C, PWM, A/D, HDMI, USB, etc etc
- Problem: there simply aren’t enough physical CPU pads to the GPIO/pin headers to account for all devices/IP blocks
- Solution: SoC designers have learned to multiplex - ‘mux’ - them into several different ‘modes’ (f.e. the TI BBB has 8 modes mode0 to mode7)
- What is a **‘pin’**?
    - “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. “

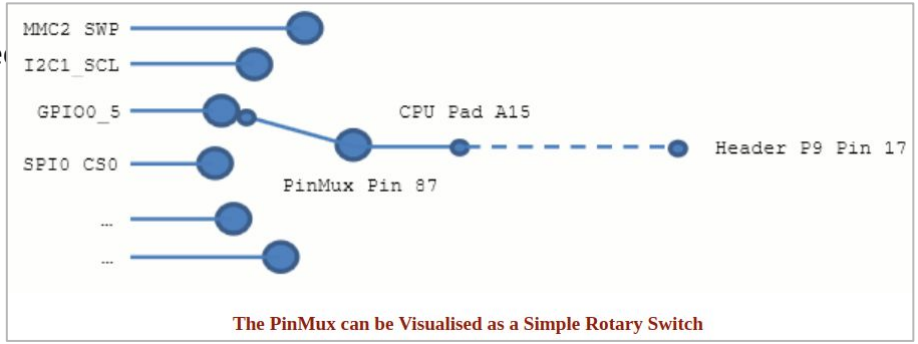
Related - Pin muxing

From: *About the Beaglebone Black PinMux Modes*

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) or with Device Tree Overlays. Note that normal user mode software (even running as root) cannot adjust the PinMux mode settings.

Let's look at how the PinMux works. In concept it can be visualise
as a simple switch (right):

- PDF of TI BBB P8 Header PinMux modes
- PDF of TI BBB P9 Header PinMux modes



The PinMux can be Visualised as a Simple Rotary Switch

Thank you!

*Done **

FYI, this PDF is available here

https://bit.ly/3V3csBB

* No, we're never really 'done'.

Q. What's the biggest room in the world?

A. The room for improvement.