Using DT overlays to support the C.H.I.P.'s capes
Antoine Ténart

- antoine.tenart@bootlin.com
- Embedded Linux engineer at Bootlin.
  - Embedded Linux specialists.
  - Development, consulting and training (materials freely available under a Creative Commons license).
  - http://bootlin.com
- Contributions
  - Kernel support for the Marvell Berlin ARM SoCs.
  - Kernel support for the Annapurna Labs ARM64 Alpine v2 platform.
- Living in Toulouse, south west of France.
What is this talk about?

- Giving an overview of how to handle capes in the kernel, and describing the requirements.
- Describing the solution we went for.
- Digging into the different parts of our solution.
Context
Context: the CHIP, the capes and us

- The **CHIP**: a 9$ board by **NextThing Co.** built around the Allwinner R8 SoC (Cortex-A8).
- Funded thanks to a Kickstarter campaign in 2015.
- **Bootlin** working on the CHIP kernel support.
- Was designed from the beginning to have adapters:
  - VGA adapter.
  - HDMI adapter.
  - Pocket CHIP.
The CHIP expander

C.H.I.P. (v1.0) PINOUT

U13

GND 1 2 CHG-IN
VCC-5V 3 4 GND
VCC-3V3 5 6 TS
VCC-1V8 7 8 BAT
TWI-SDA 9 10 PWRON
TWI-SCK 11 12 GND
X1 13 14 X2
Y1 15 16 Y2
LCD-D2 17 18 PWM0
LCD-D4 19 20 LCD-D3
LCD-D6 21 22 LCD-D5
LCD-D10 23 24 LCD-D7
LCD-D12 25 26 LCD-D11
LCD-D14 27 28 LCD-D13
LCD-D18 29 30 LCD-D15
LCD-D20 31 32 LCD-D19
LCD-D22 33 34 LCD-D21
LCD-CLK 35 36 LCD-D23
LCD-VSYNC 37 38 LCD-HSYNC
GND 39 40 LCD-DE

Use square pad
New Pin Location vs. v0.21

U14

GND 1 2 VCC-5V
UART1-TX 3 4 HPL
UART1-RX 5 6 HPCOM
FEL 7 8 HPR
VCC-3V3 9 10 MICM
LRADC 11 12 MICIN1
XIO-P0 13 14 XIO-P1
XIO-P2 15 16 XIO-P3
XIO-P4 17 18 XIO-P5
XIO-P6 19 20 XIO-P7
GND 21 22 GND
AP-EINT1 23 24 AP-EINT3
TWI2-SDA 25 26 TWI2-SCK
CSIPCK 27 28 CSICK
CSI1SYNC 29 30 CSI1SYNC
CSID0 31 32 CSID1
CSID2 33 34 CSID3
CSID4 35 36 CSID5
CSID6 37 38 CSID7
GND 39 40 GND

was LRADC>
was GND>
Cape definition & benefits

- An adapter to extend board functionalities.
- Some I/Os are muxable: different capes for different usages!
- Prototype development made easy.
- DIY projects.
- Everyone can design and sell his own capes.
Requirements

▶ Capes can be changed.
▶ Not a finite set of capes.
   ▶ The capes need to be auto-detected at boot time.
▶ Capes can be stacked.
   ▶ The auto-detection mechanism should be able to enumerate the capes.
▶ This should work **without** the user intervention!
Overview
The header

- Used to organize the cape’s description.
- Needs a magic value to differentiate it from random data.
- Capes can have different versions or revisions.
- Allows each cape to store specific data.
- This header is stored in an onboard EEPROM.
  - Easy to read from / write to.
  - Cheap.
The header format

```c
struct cape_chip_header {
    u32 magic;    /* must be 0x43484950 "CHIP" */
    u8  version;  /* spec version */
    u32 vendor_id;
    u16 product_id;
    u8  product_version;
    char vendor_name[32];
    char product_name[32];
    u8  rsvd[36];   /* rsvd for future versions */
    u8  data[16];   /* per-cape specific */
} __packed;
```
Cape identification

- Each pin used to communicate to the EEPROM cannot be reused:
  - We wanted a bus with the lowest number of lines.
- We did not need a high speed bus:
  - Only used to read the cape’s header.
- The bus must support enumeration, to connect more than one cape.
- We chose the 1-wire bus.
The CHIP is based on an ARM Cortex-A8.

The hardware description is now done with Device Trees in the upstream kernel, for ARM based boards.

Describe the SoC IPs, and which ones to enable (and configure) for a given board.

The proper solution would be to modify this device tree.

This can be done with device tree overlays!
The 1-wire bus
Overview

- Single signal.
- Low-speed data and signaling.
- Only two wires needed:
  - Data.
  - Ground.
- Uses a capacitor to store charge and power the device when the data line is active.
  - The capacitor needs to be charged!
  - We had weird side effects because of this in U-Boot → the line needs to be pulled long enough firstly.
- Two speed modes: normal and overdrive (speed x10).
- Four operations: read, write 0, write 1 and reset.
- Can be used over a GPIO.
  - drivers/w1/master/w1-gpio.c
1. Drive the bus low.
2. Wait 6\(\mu\)s.
3. Release the bus.
4. Wait 9\(\mu\)s.
5. Sample the bus to read the bit send by the slave.
6. Wait 55\(\mu\)s.
Write operation

► To write 0:
  1. Drive the bus low.
  2. Wait 60µs.
  3. Release the bus.
  4. Wait 10µs.

► To write 1:
  1. Drive the bus low.
  2. Wait 6µs.
  3. Release the bus.
  4. Wait 64µs.
Reset operation

Reset the bus slave devices and ready them for a command.

1. Drive the bus low.
2. Wait 480µs.
3. Release the bus.
4. Wait for 70µs.
5. Sample the bus:
   - 0: one or more slave devices present.
   - 1: no slave device present.
Each devices have a 64-bit unique identifier.

Used to address them individually by the master.

Binary tree search.
Kernel support

- drivers/w1/
- Not actively maintained.
- No interface to the 1-wire framework.
  - Slave drivers should be in drivers/w1/slaves/
  - Difficult to use the bus from outside the subsystem.
Introduction to Device Tree Overlays
Overview

- The device tree is a data structure.
- It’s organized as a tree: there are nodes.
- Not aimed to be generated dynamically.
- Loaded at boot time by the bootloader, or embedded in the kernel image.
- Nice for describing a SoC or a board... but not suitable for hot-pluggable stuff.
Device Tree Overlays

- Allows modification of the device tree at runtime:
  - To add a node.
  - To modify a property.

- Not persistent across reboots.

- Examples:
  - Turn on or off an hardware block by updating a node `status` property.
  - Modifying the pinmux.
  - Adding a hardware controller description.
Upstream status

- In-kernel support: `CONFIG_OF_DYNAMIC`.
- No U-Boot support (at the time of writing)... but patches sent while in the plane on our way to ELC :-)
- DTC (device tree compiler) needs a patch to enable dynamic phandle resolution.
  - Required to use device tree overlays.
  - Still not available upstream.
  - This means the one used by the kernel build system cannot handle overlays!
Overlay example: adding a new node

```
/dts-v1/;
/plugin/;
/
{
  compatible = "nextthing,chip","allwinner,sun5i-r8";

  fragment@0 {
    target-path = "'/soc@01c00000'";

    __overlay__ {
      leds {
        compatible = "gpio- leds";
        pinctrl-names = "default";
        pinctrl-0 = <&chip_test_led>;

        led0 {
          label = "Test led";
          gpios = <&pio 3 4 0>; /* PD4 */
          default-state = "on";
        }
      }
    }
  }
}
```
Overlay example: modifying a property

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

/ {

    compatible = "nextthing,chip","allwinner,sun5i-r8";

    fragment@0 {
        target = <&mmc0>;
        __overlay__ {
            status = "okay";
        }
    }

};
```
To be applied a device tree overlay fragment needs a target. Describes where to apply the changes. Two possibilities:

- **target-path**: the argument is a path.
- **target**: the argument is a phandle.

When using `target`, the phandle resolution should be dynamic.
Compiling

- `dtc -O dtb -o foo.dtb -@ foo.dts`
- The `-@` option comes from an out-of-tree patch.
- It will generate extra nodes under the root node:
  - `__symbols__` in the base tree.
  - `__symbols__, __fixups__` and `__local_fixups__` in the overlay.
  - Contains metadata used for symbol resolution.
- `/plugin/` marks device tree overlay.
Example: the base tree
/dts-v1/;

/
{
  compatible = "example";
  foo = <&bar>;

  bar: bar00 {
    compatible = "example,bar";
  }
};
Device Tree object without dynamic symbols

/dts-v1/;

/ {
    compatible = "example";
    foo = <0x00000001>;

    bar@0 {
        compatible = "example,bar";
        linux,phandle = <0x00000001>;
        phandle = <0x00000001>;
    }
};
Device Tree object with dynamic symbols

/dts-v1/;
/
{
    compatible = "example";
    foo = <0x00000001>;
    bar@0 {
        compatible = "example,bar";
        linux,phandle = <0x00000001>;
        phandle = <0x00000001>;
    };

    __symbols__ {
        bar = "/bar@0";
    };
};
Example: the overlay
/dts-v1/;
/plugin/;
/ {
    compatible = "example";
    fragment@0 {
        target-path = "/";

        __overlay__ {
            quux = <&qux>;

            qux: qux@0 {
                property = <&foo>;
            }
        }
    }
}.
Device Tree Overlay Object

```
/dts-v1/;
/ {
    compatible = "example";
    fragment@0 {
        target-path = "/";
        __overlay__ {
            quux = <0x00000001>;
            qux@0 {
                property = <0xdeadbeef>;
                linux,phandle = <0x00000001>;
                phandle = <0x00000001>;
            }
        }
    }
};
```
__symbols__ {
    qux = "]/fragment@0/__overlay__/qux@0";
};
__local_fixups__ {
    fragment@0 {
        __overlay__ {
            quux = <0x00000000>;
        }
    }
};
__fixups__ {
    foo = "]/fragment@0/__overlay__/qux@0:property:0";
};
phandle resolution
1. Get the max base tree phandle value, and add 1.
2. Adjust the overlay phandle values, then use the \texttt{__local_fixups__} node to fix local references.
3. Use the \texttt{__fixups__} node to resolve the overlay phandles referencing objects in the base tree.
4. Update these references.
Applying a Device Tree Overlay
request_firmware()

Load a firmware into memory.

The firmware is actually a Device Tree Overlay blob, stored in /lib/firmware/.

Takes the name of the firmware as an argument.

- It should be guessed from the cape’s header.
- dip-<vendor_id>-<product_id>-<product_version>.dtbo
- If not found, fallback to: dip-<vendor_id>-<product_id>.dtbo
of_fdt_unflatten_tree()
  Unflatten the overlay loaded previously.
  Create a tree of device nodes from a blob: the live tree format.

of_resolv_phandles()
  Resolves the phandles against the live tree.
  Dynamic resolution, using nodes added thanks to dtc’s -@ option.
At this point, we can use the `of_*` helpers.

Time to make some checks!

Is the overlay compatible with the machine used?
of_overlay_create()

- Creates and applies an overlay.
- Keeps track of the overlay applied.
- Can be reverted with of_overlay_destroy()
  - When removing stacked overlays, this needs to be done in reverse order.
The cape manager
Overview

- Responsible for detecting a cape, identifying it and applying the corresponding overlay.
- Uses all components described before:
  - The 1-wire bus.
  - The EEPROM in which the cape’s header is stored.
  - The device tree overlay mechanism.
- Implemented in the kernel space.
We patched the 1-wire framework to add callbacks when a new device is detected on the bus.

- Allows to read the header stored on the cape’s EEPROM as soon as the cape is detected.

- The EEPROM driver for the DS2431 was available in `drivers/w1/slaves/`

- Cannot be used outside of the 1-wire framework!

- We redefined its read function in the cape manager.
The cape manager 2/2

- Works fine for most uses.
- Our first test was with a LED and a PWM.
- This can’t work when adding / enabling devices handled by subsystems without hotplug support.
  - Like DRM/KMS.
- Quick solution: add the overlay support in the bootloader.
  - Maxime Ripard patched U-Boot.
  - Not yet upstreamed.
- Would be better to patch directly DRM/KMS.
1. A new salve device is detected on the 1-wire bus.
2. If the new device family is recognized by the cape manager, a callback is called.
3. The cape manager reads the header stored on the EEPROM.
4. The cape manager parses the header and decides which cape to load.
5. A DT overlay is loaded from userspace.
6. The overlay is applied on the live tree.
Current status
Status

► Implemented recently.
► Solution not fully upstreamed yet.
► The best thing would be to also support other boards with capes, like the Beaglebone family.
► DTC still needs to be patched.
  ► We’re not sure what to do.
► We plan to send our patches to the Linux and U-Boot communities.
Thanks! Questions?

Slides under CC-BY-SA 3.0