DTCON – Portable Device Tree Connector
Open Source Hardware thrives

- Many Linux based embedded boards are available to the community.
- Raspberry PI(s)
- Beaglebone and variants
- Linaro’s 96 boards
- C.H.I.P.
- Orange PI (*)
- Minnowboard
- And a ton of others
What do people do with them

- Just tinker about with embedded Linux!
- Learn about how hardware works and how it interfaces with Linux
- Make cool stuff with them.
  - 3D printers (Replicape)
  - Fly drones
  - Water their lawns
  - Make expansion boards to sell them.
Expansion boards

- Expansion boards are plugged into a physical connector
- Single row, or double row.
- Single connector, or multi connector
- Provides power ground and electrical signals
- Direct connection to the pads of the SoC package
- Modern SoCs are *complicated*
What it takes to make a new expansion board work?

- You connect wires!
- One set of wires is your UART, another is your I2C bus, a couple of others is your GPIO lines etc.
- You need to describe those connections and peripheral configuration in a manner than Linux understands.
- Many ways to describe hardware to Linux, we’ll talk about how you do it using Device Tree.
/ {
    foo0: foo@0 {
      compatible = "barcorp,foo";
      status = "okay";
      baz = <12>;
      interrupts = <12>;
      ...
    }
};


Exactly the same as a device soldered on the board

- No difference than when having the expansion board soldered on the main board.

- One needs to be intimately aware of the minutiae of each board and as a consequence of the SoC Linux port.

- You need to get everything right. Maybe there’s a device example configured similar to your own and you can copy.

- If not, you have to figure out everything.
Intermission: Beaglebone example

1. Read the schematic of expansion board. Write down pinout, and jot down the configuration of each pin and pin-number.

2. Lookup the connector chapter reference manual of the Beaglebone and figure out which mode the processor pad that’s connected to the connector pin must be set to.

3. Look in the hardware reference manual of the SoC which pad is configured by which pinmux register.

4. Look in the technical reference manual of the SoC which value to set in the hardware pinmux register looked up before.

5. Fill in the pinmux configuration in the correct format and add in the device tree.

6. Fill in any other special configuration parameters to the device tree nodes controlling the devices present on the expansion board.
Configuration is SoC specific

- The configuration is SoC specific.
- Figuring out the configuration is hard.
- Getting it wrong is frustrating (not easy to debug)
- Many community boards have compatible expansion connectors.
  - Arduino Compatible
  - RPI Compatible
  - Grove
  - 96 boards
Mission: Portable Expansion Boards

- Make expansion boards work in every board that has a compatible expansion connector
- Simplify the expansion board definition
- Remove all SoC specific configuration from the expansion board definition
The ubiquitous Arduino Connector
## The RPi Connector

### Raspberry Pi 3 GPIO Header

<table>
<thead>
<tr>
<th>Pin#</th>
<th>NAME</th>
<th>NAME</th>
<th>Pin#</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>3.3v DC Power</td>
<td>DC Power 5v</td>
<td>02</td>
</tr>
<tr>
<td>03</td>
<td>GPIO02 (SDA1, I2C)</td>
<td>DC Power 5v</td>
<td>04</td>
</tr>
<tr>
<td>05</td>
<td>GPIO03 (SCL1, I2C)</td>
<td>Ground</td>
<td>06</td>
</tr>
<tr>
<td>07</td>
<td>GPIO04 (GPIO_GCLK)</td>
<td>(TXD0) GPIO14</td>
<td>08</td>
</tr>
<tr>
<td>09</td>
<td>Ground</td>
<td>(RXD0) GPIO15</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>GPIO17 (GPIO_GEN0)</td>
<td>(GPIO_GEN1) GPIO18</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>GPIO27 (GPIO_GEN2)</td>
<td>Ground</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>GPIO22 (GPIO_GEN3)</td>
<td>(GPIO_GEN4) GPIO23</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>3.3v DC Power</td>
<td>(GPIO_GEN5) GPIO24</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>GPIO10 (SPI_MOSI)</td>
<td>Ground</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>GPIO09 (SPI_MISO)</td>
<td>(GPIO_GEN6) GPIO25</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>GPIO11 (SPI_CLK)</td>
<td>(SPI_CE0_N) GPIO08</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>Ground</td>
<td>(SPI_CE1_N) GPIO07</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>ID_SD (I2C ID EEPROM)</td>
<td>(I2C ID EEPROM) ID_SC</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>GPIO05</td>
<td>Ground</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>GPIO06</td>
<td>GPIO12</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>GPIO13</td>
<td>Ground</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>GPIO19</td>
<td>GPIO16</td>
<td>36</td>
</tr>
<tr>
<td>37</td>
<td>GPIO26</td>
<td>GPIO20</td>
<td>38</td>
</tr>
<tr>
<td>39</td>
<td>Ground</td>
<td>GPIO21</td>
<td>40</td>
</tr>
</tbody>
</table>

*Image Source: www.element14.com/RaspberryPi*
The Beaglebone Connector
Baseboards are commodities

- The base board is a commodity.
- The expansion boards are what’s important.
- Choose the right baseboard for my application.
- Base board makers compete in a level playing field.
- Increase the community size by reducing fragmentation.
- (ARM) Multiarch kernels + portable connectors -> Same kernel + root filesystem boots on every board.
dtcon: DT Connector

- An extcon driver
- Defines the core connector constructs.
  - connector
  - functions
  - bridge drivers
  - proxy drivers
connector

- connector/#address-cells -> connector addressing
- Each node contains one or more reg addresses
- reg addresses are pin#
- contain properties unique for each connector pin
- i.e. pad name, textual modes, pinmuxes etc.
functions

- Each pin may be assigned a function.
- Defines set of pins that comprise a function, as well as allowable combination.
- Defines properties that configure a bridge driver (i.e. gpio)
- Defines properties that configure a proxy driver, i.e. parameter names and parameter transformations.
Drivers that act a bridge between the connector domain and the base DT domain.

They are drivers that supply services to other drivers in a way that can’t be expressed using a proxy driver.

Example of bridge driver is GPIO, controls a set of pins as a single gpio driver and consolidating all pins that can be set to GPIO mode on the connector as a single driver.
Uses configuration from function it is part of to request pins from the connector and configure them properly for it to work.

It has a device target which is a pointer to the device that’s going to be configured and used.

There is a single proxy driver for all classes of devices that can be supported.

Proxy drivers deal with devices that provide no services to other drivers. Those need bridge drivers.
Real example
Beaglebone capes

- Beaglebone is a relatively popular board
- Mainline kernel with working cape support
- Extensive pinmuxing options stresses the connector infrastructure.
- A very good selection of peripherals covering corner cases
Real example
Beaglebone & a few capes

- Beaglebone is a relatively popular board
- Mainline kernel with working cape support
- Extensive pinmuxing options stresses the connector infrastructure.
- A very good selection of peripherals covering corner cases
proposed driver DT bindings

/ {
    dtcon {
        compatible = "extcon,dt-con"
    }
    status = "okay";
    connector { ... };
    functions { ... };
    plugged { ... };
};
connector node DT bindings

connector {
    #address-cells = <2>;
    #size-cells = <0>;
    GPIO1_6: GPIO1_6 {
        reg = <8 3>;
        pad = "R9";
        pinmuxes = <&gpmc_ad6>, <&mmc1_dat6>,
                   <&gpio1_6_in &gpio1_6_out>;
    }
};

...
interrupting.. pinmuxes

/* the muxes for the dtcon */

&am33xx_pinmux {

    /* P8.3 GPIO1_6 */

    gpiol_6_in: gpiol_6_in {

        pinctrl-single,pins = <

            AM33XX_IOPAD(0x818,PIN_INPUT_PULLUP | MUX_MODE7)

        >;

    };

    ...

};
Connector driver handles all pinmux setting

The list of enabled pins is used to lookup which pinmux fragments to add to the selected state and enable them.

Needs a patch to enable runtime pinmux state construction.

Completely removes the need for a non-board support or SoC developer to set muxes.
function node DT bindings

- For each function there is a node which contains configuration parameters for the function (not the device instance)
- For the bridge drivers, the contents are completely free-form
- For the proxy drivers, contents follow a general format

```plaintext
functions {
    <function-name> { };
};
```
dtcon_gpio: gpio {

    gpio-base = <256>;

    #modes = <2>;

    mode-names = "in", "out";

    gpio = <&gpio0 2 &UART2_RXD &gpio0_2_in &gpio0_2_out>,
         ...
         <&gpio2 11 &GPIO2_11 &gpio2_11_in &gpio2_11_out>,
         ...

};
GPIO cannot be a proxy driver

- GPIO driver provide services to other drivers
  - Interface with the pinctl subsystem.
  - For GPIO controllers that support it they can be an interrupt controller.
  - Many drivers reference a GPIO as part of their operation (i.e. dsr-gpios)
uart {

    params {

        rxd = { required; };

        txd = { required; };

        rts-optional { optional; gpio-property = "rts-gpios"; gpio-mode = "out";

        };

    };

};
Proxy drivers bindings
UART-cont

uart {
    uart@2 {
        device = <&uart5>;
        gpio = <&dtcon_gpio>;
        mux@0 {
            txd = <&UART5_TXD &uart5_txd>;
            rxd = <&UART5_RXD &uart5_rxd>,
                <&UART5_CTSN &uart5_rxd_mux1>;
        }
    }
}
Proxy drivers bindings

i2C

```plaintext
i2c {

  params {

    scl = { required; };

    sda = { required; };

  };

};

- And so on... Proxy drivers are generally the same although special behavior can be tackled with the match OF compatible ID table.
```
Plugged: Just a bus

- compatible = “simple-bus”; actually

- Target for overlays, anything dropped there will be instantiated.

- More secure than vanilla overlays with a security configuration option that disallows any overlays having a target outside of the plugged node.
BB_BONE_UART {

    compatible = "dtcon-uart";

    status = "okay";

    txd = <9 21>;

    rxd = <9 22>;

};
BB-BONE-UART
sequence of events

- A UART compatible proxy driver is instantiated.
- Using the “uart” function node we parse the “parameters” node for connector properties that are representing a connector property.
- We iterate over all children of the uart node for a device target property.
- A match is found when the property points to the connector node that the reg property matches.
BB-BONE-UART

sequence of events (cont)

- The pinmux option that is part of the tuple is enabled
- All properties/child nodes are copied in the target device node.
- Performed using a changeset so that they can be reverted.
- The target device is enabled with ‘status=“okay”’ and it is created.
BB-RELAY-4PORT

BB_RELAY_4PORT {

compatible = "simple-bus";

status = "okay";

gpio_relay_4port: gpio_relay_4port@0 {

compatible = "dtcon-pio";

status = "okay";

gpio-controller;

#gpio-cells = <2>;

pin-list = <9 15>, <9 23>, <9 12>, <9 27>;

};
leds@0 {
    compatible = "gpio-leds";
    status = "okay";
    jp@1 {
        gpios = <&gpio_relay_4port 0 GPIO_ACTIVE_HIGH>;
        default-state = "keep";
    }
};

BB-RELAY-4PORT
sequence of events

- The dtcon-gpio device which is created using the pin-list property to locate the matching connector nodes.
- Note that the numbering of the GPIOs are in the order they are stated in the pin-list property.
- The backend gpiochips are located and kept in a list.
- When the led driver is probed internal API calls are forwarded to the real GPIO drivers present.
- Unfortunately major rewrite of GPIO layer underway, patches do not apply, WIP.
Development status

- Mostly works for standard devices (UART/I2C/SPI etc).
- GPIO is broken due to GPIO rewrite. Needs a cleaner interface for cascaded GPIO operations.
- Pinmuxing needs patch to build a pinmux state dynamically.
- Bridge drivers for other hardware classes. PWM is next.
- Mainline will take quite a few iterations.
Thank you!

Questions?