Embedded Linux from a Trusted Source

Overcoming Your Hesitation to Migrate to a Linux Device Tree Kernel

2015 July 28

Thank you for attending our Hangout On Air today.

This broadcast will start at the top of the hour (2pm US EDT).

Timesys: Reducing Embedded Linux Risk and Complexity

First, the prior approach (pre-Device Tree)

- Bootloader loads and boots a single Linux kernel binary image
- The complete description of the hardware is fully contained in the Linux kernel source tree, in the “board” file
  - e.g., arch/arm/mach-mx6/board-mx6_nitrogen6x.c
  - Identifies SOC and all components, along with some glue code
- At boot time, the bootloader prepares additional information (ATAGS) for the kernel
  - Passes a pointer to ATAGS via r2 register
  - ATAGS contains information such as memory size and location, kernel command line, etc.
  - Passes machine type via register r1

- Pros: Code is written in C
- Cons: Linux kernel configuration is fixed, so any changes to the hardware configuration require the kernel to be recompiled

More specifically for ARM (prior to 2011)

- The ARM architecture wasn't using the Device Tree approach, and a large portion of the SoC support was located in ***arch/arm/mach-<foo>,*** e.g., mach-mx6.
- Each board supported by the kernel was associated with a unique machine ID.
- The entire list of machine ID's can be accessed at http://www.arm.linux.org.uk/developer/machines/download.php, and anyone could freely register an additional one.
- 5044 unique machine ID's registered
- Linux kernel defined a machine structure for each board, which associated the machine ID with a set of information and callbacks
- Bootloader passed the machine ID to the kernel in the r1 register

And now, the Device Tree approach

- The bootloader now loads two binaries:
  - Kernel image
  - Device Tree Blob (.dtb file) for the specific hardware
- So what is a device tree?

  A description of the hardware in the form of a data structure, that is passed to the kernel at boot time
- More specifically, it is a data structure, that consists of a tree of nodes that models the hierarchy of devices in the system, from the devices inside the core/processor to the devices outside the SOC, on the board itself.
- Each node can have a number of properties describing the devices: addresses, interrupts, clocks, etc.
- The bootloader now passes the DTB address through r2. It no longer cares about the r1 register

Compare to a PC, where initial registers are hardcoded, and the BIOS supplies the rest of the hardware information

A Little Bit of Device Tree History

- Device Tree is not new, has it’s roots in Open Firmware, first developed by Sun for SPARC in the 90’s, evolved into standalone Flattened Device Tree (FDT)
- 2005: PowerPC architecture support added
- 2011: MIPS and ARM architecture support added
- Ongoing: Semiconductor companies add SOC and dev kit support, board vendors add SOM/board support
  - 3.0: Basic empty infrastructure for DT support added for ARM
  - 3.5: Marvell Kirkwood
  - 3.8: Atmel SAM
  - 3.9: NVIDIA Tegra
  - 3.11: APM, Freescale Vybrid, Samsung
  - 3.12: Freescale i.MX6

and so on…

STATS from kernel-4.1.3:

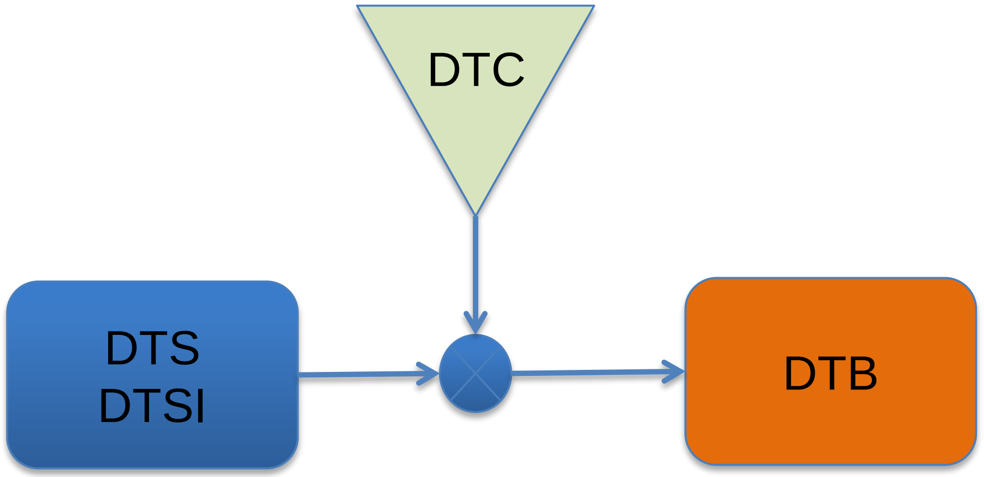
- 566 boards using Device Tree in arch/arm/boot/dts
- 194 boards using Device Tree in arch/powerpc/boot/dts

Benefits of Device Tree Approach

- Leverages hardware similarities
  - The USB controller on i.MX28 is similar to the one used on i.MX6
  - Serial port on Raspberry PI is the same as generic UART developed by ARM (used also in QEMU)
- Decouples kernel code from SoC/board-specific data
- No need to maintain 3 different kernel versions for 3 different hardware configurations. A single Linux kernel binary is sufficient. Separate DTB files will take care of hardware configuration. Result: less code to maintain
- Faster board ports -- you don't have to develop it from scratch! Architecture (e.g., ARM) and SoC vendors (e.g., Freescale, TI) have done the majority of the work, if the SoC is DT-enabled

Device Tree Process

- The Device Tree Source (DTS and DTSI) files contain SoC-level and board-level definitions
  - Are located in arch/arm/boot/dts
- The Device Tree Compiler (DTC) compiles the source into a binary form
  - DTC located in scripts/dtc
- The Device Tree Blob (DTB) is the result of a compilation
- The arch/arm/boot/dts/Makefile defines which DTBs should be generated at kernel build time



Device Tree Inheritance

- Each particular hardware platform has its own device tree.
- However, several hardware platforms use the same processor/SoC, and often various processors in the same family share a number of similarities.
- To allow this, a device tree file can include another one. The trees described by the including file overlays the tree described by the included file. This can be done:
  - Either by using the /include/ statement provided by the Device Tree language.
  - Or by using the #include statement, which requires calling the C preprocessor before parsing the Device Tree.

Device Tree Inheritance - example

```
/ {
        compatible = "ti,am33xx";
        [...]
        ocp {
                uart0: serial@44e09000 {
                        compatible = "ti,omap3-uart";
                        reg = <0x44e09000 0x2000>;
                        interrupts = <72>;
                        status = "disabled";
                };
        };
};
                                                  am33xx.dtsi
```

+

```
#include "am33xx.dtsi"

/ {
        compatible = "ti,am335x-bone", "ti,am33xx";
        [...]
        ocp {
                uart0: serial@44e09000 {
                        pinctrl-names = "default";
                        pinctrl-0 = <&uart0_pins>;
                        status = "okay";
                };
        };
};
                                                  am335x-bone.dts
```

=

Compiled DTB

```
/ {
        compatible = "ti,am335x-bone", "ti,am33xx";
        [...]
        ocp {
                uart0: serial@44e09000 {
                        compatible = "ti,omap3-uart";
                        reg = <0x44e09000 0x2000>;
                        interrupts = <72>;
                        pinctrl-names = "default";
                        pinctrl-0 = <&uart0_pins>;
                        status = "okay";
                };
```

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

timesys®

Evolution of ARM support in the Linux kernel

- As the ARM architecture gained signicantly in popularity, some major refactoring was needed:
  - First, the Device Tree was introduced on ARM: instead of using C code to describe SoCs and boards, a specialized Device Tree language is used.
  - Second, a number of driver infrastructures were created, to replace custom code in arch/arm/mach-<foo>:
    - The common clock framework in drivers/clk
    - The pinctrl subsystem in drivers/pinctrl
    - The irqchip subsystem in drivers/irqchip
    - The clocksource subsystem in drivers/clocksource

Adding support for your custom board

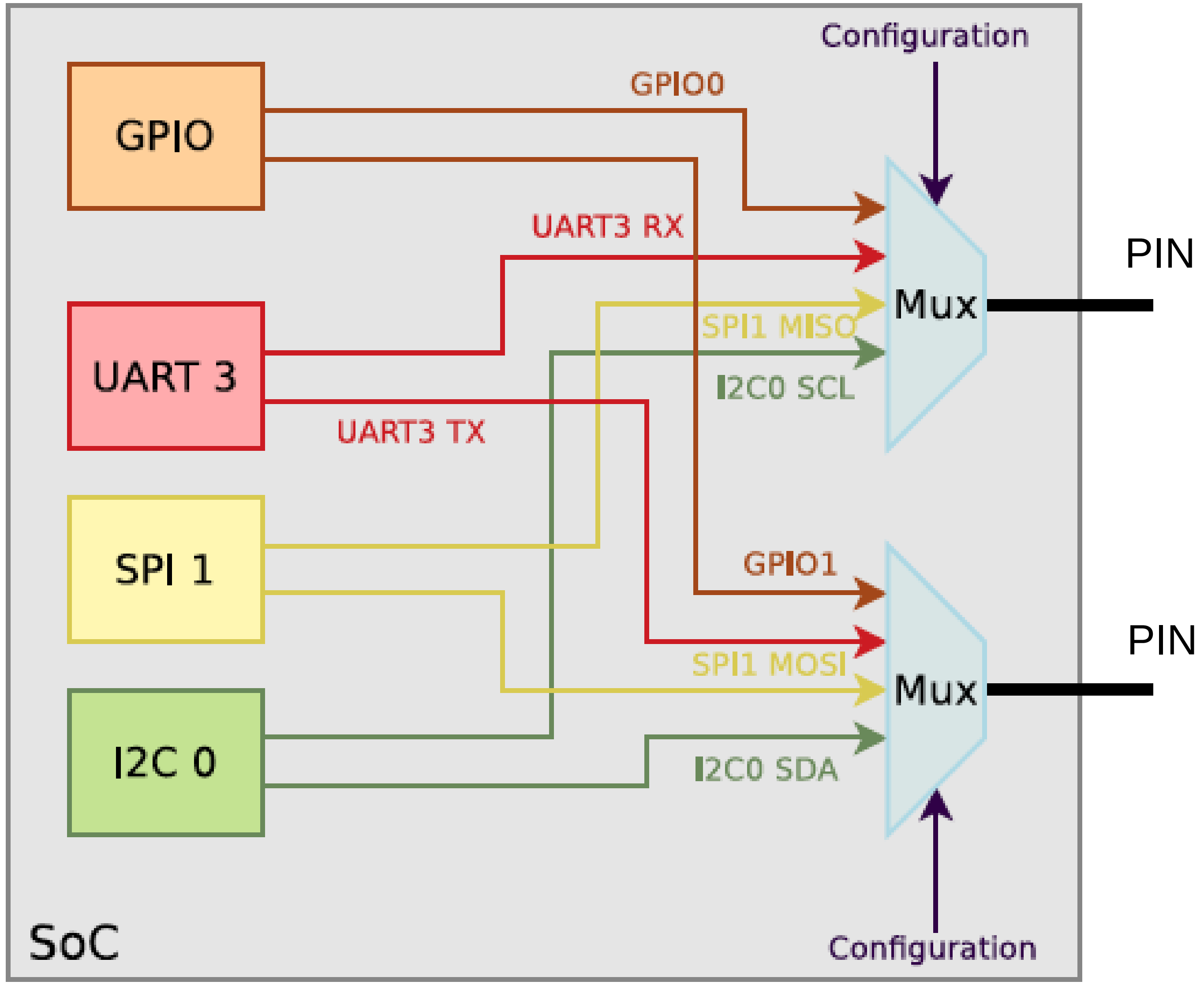
Thanks to the code refactoring, the process is straightforward. It is divided into SoC and Board parts (assuming the SoC on your board is supported by the Linux kernel):

1. Create a Device Tree in ***arch/arm/boot/dts***, generally named ***<soc-name>-<board-name>.dts,*** and include the relevant SoC .dtsi file.
   - Your Device Tree will describe all the SoC peripherals that are enabled, the pin muxing, as well as all the devices on the board.
2. Modify ***arch/arm/boot/dts/Makefile*** to make sure your Device Tree gets built as a DTB during the kernel build.
3. If needed, develop the missing device drivers for the devices that are on your board and outside the SoC.

Additional Consideration when bringing up a Device Tree Kernel on a custom board: Pin Muxing

- New SoCs (System on Chip) come with increasing number of hardware blocks, many of which need to interface with the outside world using pins.
- The physical size of the chips remains small, and therefore the number of available pins is limited.
- The problem: # hardware blocks > # available pins
- The solution: pins are multiplexed, exposing either the functionality of hardware block A or the functionality of hardware block B.
- This multiplexing is usually software configurable.

Pin Muxing diagram



Pinctrl subsystem

- A pinctrl subsystem has been added in Linux 3.2
- This subsystem (drivers/pinctrl) provides a generic subsystem to handle pin muxing. It provides:
  - A pin muxing consumer interface, for device drivers.
  - A pin muxing driver interface, to implement the system-on-chip specific drivers that configure the muxing.
- Most pinctrl drivers provide a Device Tree binding, and the pin muxing must be described in the Device Tree
  - The exact Device Tree binding depends on each driver. Each binding is documented in Documentation/devicetree/bindings/pinctrl

Defining pinctrl configurations

- The different pinctrl configurations must be defined as child nodes of the main pinctrl device (which controls the muxing of pins).
- The configurations may be defined at:
  - the SoC level (.dtsi file), for pin congurations that are often shared between multiple boards
  - at the board level (.dts file) for configurations that are board specific.
- The pinctrl-<x> property of the consumer device points to the pin configuration it needs through a DT handle.

Example – LM75 temperature sensor

- Enabling LM75 temperature sensor on AM335x BeagleBone Black
- The I2C0 bus is not brought out in the expansion headers, so we will use I2C2

```
i2c2: i2c@4819c000 {
	/* Setting up multiplexing*/
	pinctrl-names = "default";
	pinctrl-0 = <&i2c2_pins>;
	/* Device is on*/
	status = "okay";
	/* Driver variable - I2C works at 400kHz */
	clock-frequency = <400000>;

	/* Adding LM75 sensor under address 0x4f*/
	lm75@4f {
		compatible = "lm75";
		reg = <0x4f>;
	};
};
```

Example – LM75 temperature sensor (cont’d)

- **In order to setup I2C multiplexing we need to look in a documentation**

```
i2c2_pins: pinmux_i2c2_pins {
        pinctrl-single,pins = <
        0x178 (PIN_INPUT_PULLUP | MUX_MODE3) /* uart1_ctsn.i2c2_sda */
        0x17c (PIN_INPUT_PULLUP | MUX_MODE3) /* uart1_rtsn.i2c2_scl */
        >;
};
```

- **Now we need to configure the appropriate driver in**

```
  Device Drivers --->
                <*> Hardware Monitoring support --->
                                <*> National Semiconductor LM75 and compatibles
```

- **After kernel and DTB are ready, we boot the board and we can check if the LM75 sensor works as desired**

```
cat /sys/class/hwmon/hwmon0/device/temp1_input
```

timesys®

Adding support for custom pieces

Assuming the SoC is supported, the following core components need to be added/enabled:

- Support for pin muxing, through the pinctrl subsystem. See drivers/pinctrl/
- Support for the clocks. Usually requires some clock drivers, as well as DT representations of the clocks. See drivers/clk/
- Support for pin muxing, through the pinctrl subsystem. See drivers/pinctrl/
- Support for GPIOs, through the GPIO subsystem. See drivers/gpio/
- Finally, drivers for the different hardware blocks:
  - Ethernet driver, in drivers/net/ethernet/
  - SATA driver, in drivers/ata/
  - I2C driver, in drivers/i2c/busses/
  - SPI driver, in drivers/spi/

etc.

References / Other Resources

- http://devicetree.org/Main_Page
- Documentation/devicetree (in kernel source tarball)
- http://elinux.org/Device_Tree
- http://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf
- http://events.linuxfoundation.org/sites/events/files/slides/Engaging_Device_Trees_0.pdf
- https://lwn.net/Articles/572692/
- https://lwn.net/Articles/573409/
- https://linuxlink.timesys.com/docs/wiki/linuxlink-docs/HOWTO-modify-devicetree

About this presentation

Contact Us

Contact Al Feczko at:

- al.feczko@timesys.com
- 1-866-392-4897 (toll-free)
- 1-412-325-6390 (direct)

Contact Technical Support at:

- https://linuxlink.timesys.com/support linuxlink.timesys.com/support

Contact Sales by sending email to sales@timesys.com