



# Linaro IoT and Embedded Group (LITE)

Device Tree and CMSIS Discussion  
Andy Gross



**LITE**  
IoT/EMBEDDED

# Introduction to Device Tree

- Device tree is a way of describing hardware configurations in a way that is extensible and easy to utilize in system configuration and device driver initialization.
- Adding device nodes is fairly straightforward and extensible enough to handle any type of device.
- New boards utilizing already supported SoCs can leverage definitions and add or change nodes to correctly describe that board's implementation. This approach promotes reuse of existing configuration information.
- Classical device tree use involves compiling a textual DTS file into a flattened device tree blob (dtb). Bootloaders and later operating systems can access the blob and build up device lists or pull out required information.
- A full description of device tree can be found [in the device tree specification](#).

# Configuration in Zephyr

- Configuration is spread out between two different areas in the Zephyr directory structure. One place resides in the arch/arm/soc/ directory, and another is in the boards/ directory.
- Each board has a defconfig that defines a set of options. Additional options are included from Kconfig files spread out in the two configuration directories.
- Kconfig options are also used to control inclusion of multiple instances of devices/peripherals. A good example of this are the GPIO ports on a number of boards. UART ports is also another good example.
- Configuration is mainly hard coded using the Kconfig options.
- The user does choose the architecture for building (ARM in our case), but this is also true of the other arches.
- Adding new boards from the same SoC family requires copying a lot of previous work.
- Device drivers use the Kconfig options to build up their device data. #ifdef statements in the drivers decide which devices are actually created and initialized.

# Configuration Sources and other Distributions

- CMSIS is a vendor agnostic hardware abstraction layer for the ARM Cortex-M processors with the intent of standardization of interfaces to peripherals, RTOSs, and middleware.
- CMSIS provides some configuration information through different components. A few of the more relevant ones are:
  - CMSIS-Core provides API and definitions for the Cortex M and associated peripherals.
  - CMSIS-Driver provides peripheral driver interfaces. Essentially, it is a HAL.
  - CMSIS-SVD describes all the components of the system using XML.
  - CMSIS-Pack uses a XML based package description to pull together
- ARM's Keil uses a good portion of the CMSIS components and ties it into an IDE.
- MBED uses json for their configuration. Can someone speak to this?

# What can device tree do for Zephyr?

- Adding new boards based on already supported platforms is much easier due to reuse of a lot of the configuration information.
- Device tree is architecturally neutral. One can easily describe hardware components regardless of the architecture of the processor.
- Device tree can be used to generate build options, and also configuration structures that will be used by device drivers and system initialization code.
- Using device tree for configuration would result in a much smaller set of Kconfig options, as most of the options can be derived from the device tree information.
- As device tree is very flexible, it can describe all of the attributes/properties of the devices out there currently, and also handle future changes.
- The same device tree information can also be used by upper layers of software (libs, apps, frameworks, etc). This would reduce variation in these areas due to differences between boards.

# What parts of device tree will we need?

- With Zephyr, the concern is configuration of the SoC and associated peripherals. We are not looking at using the flattened device tree blob, the APIs to access information from that blob, or associating nodes with devices in the system.
- The device tree files will contain device information that will in turn be used to generate build options, structures that are used during device initialization, and perhaps even during initial system bringup.
- Runtime device tree access is not in the scope of the proposed changes. The device tree will only be used to source information during building and board initialization.
- Will versioning be an issue?

# Building a device tree specification for Zephyr

- CPU hierarchy
- Devices and resources
  - Register address space
  - Interrupts
  - GPIOs
  - Pinmux information
  - Clock gating/control
  - Clock frequencies and multipliers/divisors
  - String names for devices
  - Device specific information (PWM, RTC, WDT, etc).
- Static pinmux settings?

# Example DTS for NXP FRDMK64F

```
/dts-v1/;

/ {
    compatible = "nxp,k64f", "nxp,mk64f12";
    #address-cells = <1>;
    #size-cells = <1>;

    cpus {
        cpu@0 {
            compatible = "arm,cortex-m4f";
        };
    };

    memory {
        compatible = "mmio-sram";
        reg = <0x20000000 0x30000>;
    };

    interrupt-controller@e000e100 {
        compatible = "arm,cortex-m4-nvic";
        reg = <0xe000e100 0x3ef>;
        num-irq-prio-bits = <4>;
        num-irqs = <86>;
    };

    timer@e00e010 {
        compatible = "arm,cortex-m4-systick";
        reg = <0xe000e010 0x10>;

        clk-source = <0>; /* AHB or AHB/8 */
    };

    mpu@4000d000 {
        compatible = "nxp,k64f-mpu";
        reg = <0x4000d000 0x824>;

        /* Add regions here that describes address, range, valid, and
        permissions */
    };

    clock-controller@40064000 {
        compatible = "nxp,k64f-mcg";
        reg = <0x40064000 0xd>;
        system-clock-frequency = <120000000>;
    };

    clock-controller@40065000 {
        compatible = "nxp,k64f-osc";
        reg = <0x40065000 0x4>;

        enable-external-reference;
    };
};
```

# Example DTS for NXP FRDMK64F - Continued

```
rtc@4003d000 {
    compatible = "nxp,k64f-rtc";
    reg = <0x4003d000 0x808>;
    clock-frequency = <32768>;          /* fixed 32kHz clk */
};

sim@40047000 {
    compatible = "nxp,k64f-sim";
    reg = <0x40047000 0x1060>;

    clk-divider-core = <1>;
    clk-divider-bus = <2>;
    clk-divider-flexbus = <3>;
    clk-divider-flash = <5>;
};

flash-controller@4001f000 {
    compatible = "nxp,k64f-flash-controller";
    dev-name = "flash_woah";
    reg = <0x4001f000 0x27c>;
    interrupts = <18 0 3>;
};

flash@0 {
    reg = <0 0x100000>;
};

uart@4006a000 {
    compatible = "nxp,k64f-uart";
    reg = <0x4006a000 0x1000>;
    interrupts = <31 0 0>;              /* irq 31 - no flags - prio 0 */
    baud-rate = <115200>;
    pinmux = <1 16 3>,
            <1 17 3>; /* RX/TX - Port B - pins 16/17 - alt func 3 */
    dev-name = "UART_0";
};

uart@4006b000 {
    compatible = "nxp,k64f-uart";
    reg = <0x4006b000 0x1000>;
    interrupts = <33 0 0>;
    baud-rate = <115200>;
    dev-name = "UART_1";
};

pinmux@40049000 {
    compatible = "nxp,k64f-pinmux";
    reg = <0x40049000 0x40ca>;
};
```

# Example DTS for NXP FRDMK64F - Continued

```
gpioa: gpio@400ff000 {
    compatible = "nxp,k64f-gpio";
    reg = <0x400ff000 0x40>;
    interrupts = <59 0 3>;
    dev-name = "GPIO_0";
};
```

```
gpiob: gpio@400ff040 {
    compatible = "nxp,k64f-gpio";
    reg = <0x400ff040 0x40>;
    interrupts = <60 0 3>;
    dev-name = "GPIO_1";
};
```

```
gpioc: gpio@400ff080 {
    compatible = "nxp,k64f-gpio";
    reg = <0x400ff080 0x40>;
    interrupts = <61 0 3>;
    dev-name = "GPIO_2";
};
```

```
gpiod: gpio@400ff0c0 {
    compatible = "nxp,k64f-gpio";
    reg = <0x400ff0c0 0x40>;
    interrupts = <62 0 3>;
    dev-name = "GPIO_3";
};
```

```
gpioe: gpio@400ff100 {
    compatible = "nxp,k64f-gpio";
    reg = <0x400ff100 0x40>;
    interrupts = <63 0 3>;
    dev-name = "GPIO_4";
};
```

```
spi@4002c000 {
    compatible = "nxp,k64f-spi";
    reg = <0x4002c000 0x88>;
    interrupts = <26 0 3>;
    clocks = <0x4004803C 12>; /* clk gate */
    dev-name = "SPI_0";

    cs = <&gpiob 10>, <&gpiob 9>; /* cs0 = PTB10, cs1 = PTB9 */
    pinmux = <1 10 2>, <1 9 2>; /* PTB9/10 set to alt func 2 */
};
```

# Example DTS for NXP FRDMK64F - Continued

```
spi@4002d000 {
    compatible = "nxp,k64f-spi";
    reg = <0x4002d000 0x88>;
    interrupts = <27 0 3>;
    clocks = <0x4004803C 13>;          /* clk gate */
    dev-name = "SPI_1";
};

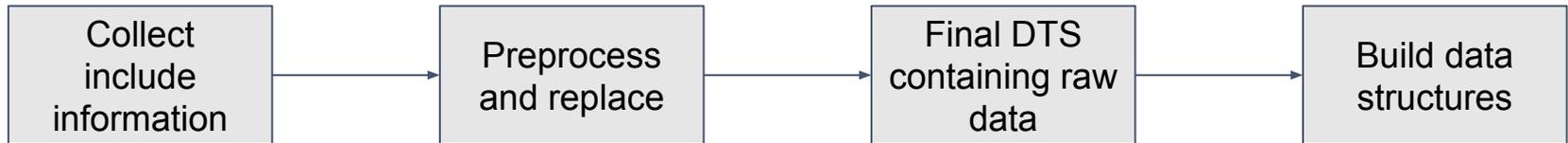
watchdog@40052000 {
    compatible = "nxp,watchdog-k64";
    reg = <0x40052000 16>;
    dev-name = "WDT_0";
    clock-source = <0>;              /* LPO 1kHz or alternate source */
    reload-counter = <40000>;
    start-on-boot;
    prescaler = <2>;
};
```

```
pwm@40038000{
    compatible = "nxp,pwm-k64";
    reg = <0x40038000 0x98>;
    dev-name = "PWM_0";
    prescaler = <2>;
    period = <1000>;
    clock-source = <0>;
    /* channel information needed - fixme */
};

pwm@40039000{
    compatible = "nxp,pwm-k64";
    reg = <0x40039000 0x98>;
    dev-name = "PWM_1";
    prescaler = <2>;
    period = <1000>;
    clock-source = <0>;
    /* channel information needed - fixme */
};
```

# Tooling required for Zephyr Device Tree Usage

- Device tree files can be written using information contained in other files. These files could include the CMSIS files, SoC vendor generated files, and other sources for configuration information. This information needs to be separated out so that it can be included in the device tree file.
- Using the C preprocessor and running the device tree compiler in a DTS to DTS passthrough, the end result is a file that contains all of the required information in a format that is usable to create data structures that would feed device drivers and board initialization functions.



- Using a tool like the `dt_to_config` from the Linux kernel, the device tree file can be parsed and a set of defconfig options will be generated. This gives the advantage of capturing all the options required for a board, and it also reduces duplicate sets of options for device multiples. One simple solution for the matching would involve using a whitelist file for each SoC that gives a compatible to config option mapping.

# Proposed changes to Zephyr Configuration

- With configuration information primarily coming from device tree, the board and arch/arm directories can get squashed down to one. A lot of redundant information could be removed.
- The data structures derived from the device tree information will feed into the device creation API calls. This would remain the same. The only difference being how the information was sourced.
- All of the device tree manipulations and code generation would occur as part of the building of a target.
- Could system memory be recouped using `__init` section for the device information?

# Work for the near term

- Define DTS files for a few platforms, not just the NXP FRDM.
- Get the tooling in place for creating a set of data structures that are consumed by the board init functions. The followup to this is to extend that to also work for the device drivers.
- Generate the config from the dts files and work this into the Makefiles
- Cleanup the configuration directories for the boards as the required existing config and board files are retired. This will most likely involve complete removal of the board/ directories.

Questions?