A short introduction to DeviceTrees

V1.1, SOTR 22-23

Paulo Pedreiras
pbrp@ua.pt
UA/DETI/IT

What is a Device Tree

- From the Devicetree Specification Release v0.3-40-g7e1cc17:

  “A devicetree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent.

  A DTSpec-compliant devicetree describes device information in a system that cannot necessarily be dynamically detected by a client program”

  - **That is, a device tree describes hardware that <u>may or may not</u> be detectable by other means, e.g. probing**.

Where do I find DeviceTrees

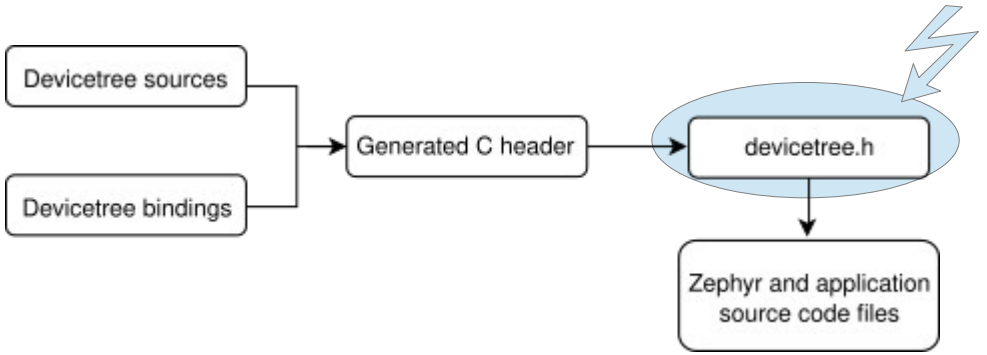
- Device Trees are used in many systems
  - Linux (embedded or not)
  - STM products and boards
  - The Zephyr Embedded OS
    - And consequently on the Nordic nrF Connect SDK
  - Etc.
- **The focus of this presentation is on Zephyr/Nordic**

Introduction

- A devicetree is a hierarchical data structure that describes hardware.
- The Devicetree specification defines its source and binary representations.
- **Zephyr uses devicetree to describe the hardware available on its Supported Boards, as well as that hardware's initial configuration.**
- There are two types of devicetree input files:
  - devicetree sources and devicetree bindings.
    - The sources contain the devicetree itself.
    - The bindings describe its contents, including data types.

Introduction

- The build system uses devicetree sources and bindings to produce a generated C header.
- The generated header's contents are abstracted by the devicetree.h API, which you can use to get information from your devicetree.
- **All Zephyr and application source code files can include and use devicetree.h**. This includes device drivers, applications, tests, the kernel, etc.



Syntax and Structure

- As the name indicates, a devicetree is a (hierarchical) tree.
- The human-readable text format for this tree is called DTS (DeviceTree Source), and is defined in the Devicetree specification.
- Example DTS file:

```
/dts-v1/;
/ {
        a-node {
            subnode_label: a-sub-node {
                    foo = <3>;
            };
        };
};
```

- /dts-v1/
  - file's contents are in version 1 of the DTS syntax
- The tree has 3 nodes
  - "/": root node
  - A node named "a-node" (child of root)
  - A node named "a-sub-node" (child of "a-node")
- Nodes can be given labels that can be used to refer to the labeled node elsewhere in the devicetree.
  - "a-sub-node" has label "subnode_label"

Syntax and Structure

- Devicetree nodes have **paths** identifying their locations in the tree and work like Unix file system paths
  - E.g. the full path to “a-sub-node” is “/a-node/a-sub-node”

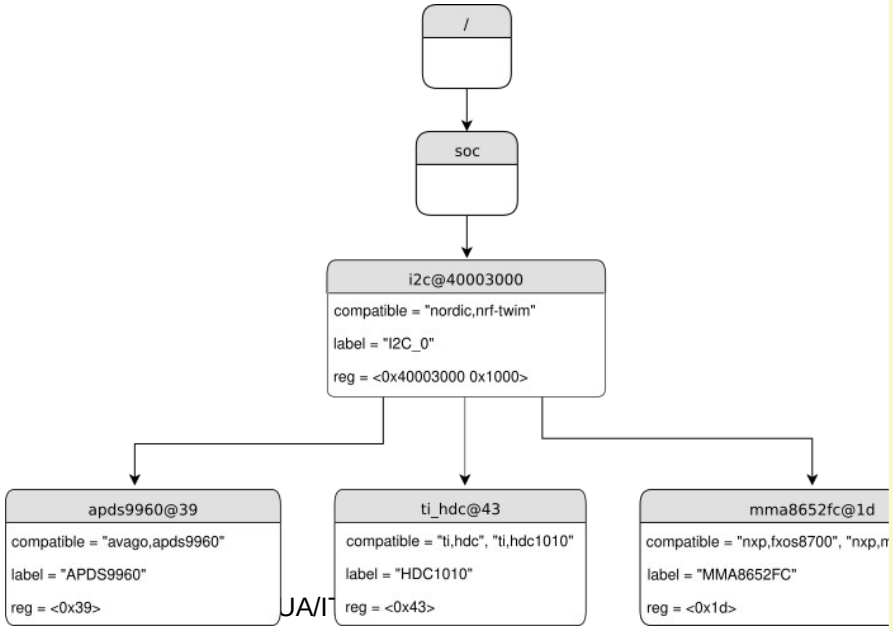
- Devicetree nodes can also have **properties**.
  - **Properties** are **name/value** pairs.
  - Values can be any sequence of bytes.
  - In some cases, the values are an array of what are called cells. A cell is just a 32-bit unsigned integer.

```
/dts-v1/;
/ {
      a-node {
         subnode_label: a-sub-node {
                foo = <3>;
         };
      };
};
```

  - Node “a-sub-node” has a property named “foo”, whose value is a cell with value 3. The size and type of foo‘s value are implied by the enclosing angle brackets (< and >) in the DTS

Syntax and Structure

- In practice, devicetree nodes usually correspond to some hardware, and the node hierarchy reflects the hardware's physical layout. For example:




```
/dts-v1/;

/ {
    soc {
        i2c@40003000 {
                compatible = "nordic,nrf-twim";
                label = "I2C_0";
                reg = <0x40003000 0x1000>;

                apds9960@39 {
                    compatible = "avago,apds9960";
                    label = "APDS9960";
                    reg = <0x39>;
                };
                ti_hdc@43 {
                    compatible = "ti,hdc", "ti,hdc1010";
                    label = "HDC1010";
                    reg = <0x43>;
                };
                mma8652fc@1d {
                    compatible = "nxp,fxos8700", "nxp,mma8652fc";
                    label = "MMA8652FC";
                    reg = <0x1d>;
                };
        };
    };
};
```

Syntax and Structure

- In addition to showing more realistic names and properties, the DTS example introduces a new devicetree concept - **unit addresses**.
  - Unit addresses are the parts of node names after an “at” sign (@), like 40003000 in i2c@40003000 or 39 in apds9960@39.
  - Unit addresses are optional: the soc node does not have one.

```
/dts-v1/;

/ {
    soc {
        i2c@40003000 {
                compatible = "nordic,nrf-twim";
                label = "I2C_0";
                reg = <0x40003000 0x1000>;

                apds9960@39 {
                    compatible = "avago,apds9960";
                    label = "APDS9960";
                    reg = <0x39>;
                };
                ti_hdc@43 {
                    compatible = "ti,hdc", "ti,hdc1010";
                    label = "HDC1010";
                    reg = <0x43>;
                };
                mma8652fc@1d {
                    compatible = "nxp,fxos8700", "nxp,mma8652fc";
                    label = "MMA8652FC";
                    reg = <0x1d>;
                };
        };
    };
};
```

Syntax and Structure

- Unit addresses examples
  - **Memory-mapped peripherals:**
    - The peripheral's register map base address. For example, the node named i2c@40003000 represents an I2C controller whose register map base address is 0x40003000.
  - **I2C peripherals**
    - The peripheral's address on the I2C bus. For example, the child node apds9960@39 of the I2C controller in the previous section has I2C address 0x39.
  - **SPI peripherals**
    - An index representing the peripheral's chip select line number. If there is no chip select line, 0 is used.
  - **Memory**
    - The physical start address. For example, a node named memory@2000000 represents RAM starting at physical address 0x2000000.
  - There are others, e.g. flash ...

Syntax and Structure

- Some important properties
  - **compatible**
    - The name of the hardware device the node represents. The recommended format is "vendor,device", like "avago,apds9960", or a sequence of these, like "ti,hdc", "ti,hdc1010".
    - The vendor part is an abbreviated name of the vendor.
    - It is also sometimes a value like gpio-keys, mmio-sram, or fixed-clock when the hardware's behavior is generic.
    - **The build system uses the compatible property to find the right bindings for the node. Device drivers use devicetree.h to find nodes with relevant compatibles, in order to determine the available hardware to manage.**
    - **Within Zephyr's bindings syntax, this property has type string-array.**

Syntax and Structure

- Some important properties
  - **label**
    - The device's name according to Zephyr's Device Driver Model.
    - The value can be passed to *device_get_binding()* to retrieve the corresponding driver-level *struct device**.
    - This pointer can then be passed to the correct driver API by application code to interact with the device.
    - **For example, calling device_get_binding("I2C_0") would return a pointer to a device structure which could be passed to I2C API functions like i2c_transfer().**
    - The generated C header will also contain a macro which expands to this string.

Syntax and Structure

- Some important properties
  - **reg**
    - Information used to address the device. The value is specific to the device.
    - The reg property is a sequence of (address, length) pairs. Each pair is called a “register block”.
    - Some common patterns:
      - Devices accessed via memory-mapped I/O registers (like i2c@40003000)
        - Address is usually the base address of the I/O register space, and length is the number of bytes occupied by the registers.
      - I2C devices (like apds9960@39 and its siblings):
        - Address is a slave address on the I2C bus. There is no length value
      - SPI devices: address is a chip select line number; there is no length.
    - **There are some evident similarities between the reg property and common unit addresses described before. It is not a coincidence ad the reg property can be seen as a more detailed view of the addressable resources within a device than its unit address.**

Syntax and Structure

- Some important properties
  - **status**
    - A string which describes whether the node is enabled.
    - The devicetree specification allows this property to have values "okay", "disabled", "reserved", "fail", and "fail-sss". Only the values "okay" and "disabled" are currently relevant to Zephyr.
    - A node is considered enabled if its status property is either "okay" or not defined .
    - Nodes with status "disabled" are explicitly disabled.
    - **Devicetree nodes which correspond to physical devices must be enabled for the corresponding struct device in the Zephyr driver model to be allocated and initialized.**
  - **interrupts**
    - Information about interrupts generated by the device, encoded as an array of one or more interrupt specifiers. Each interrupt specifier has some number of cells.

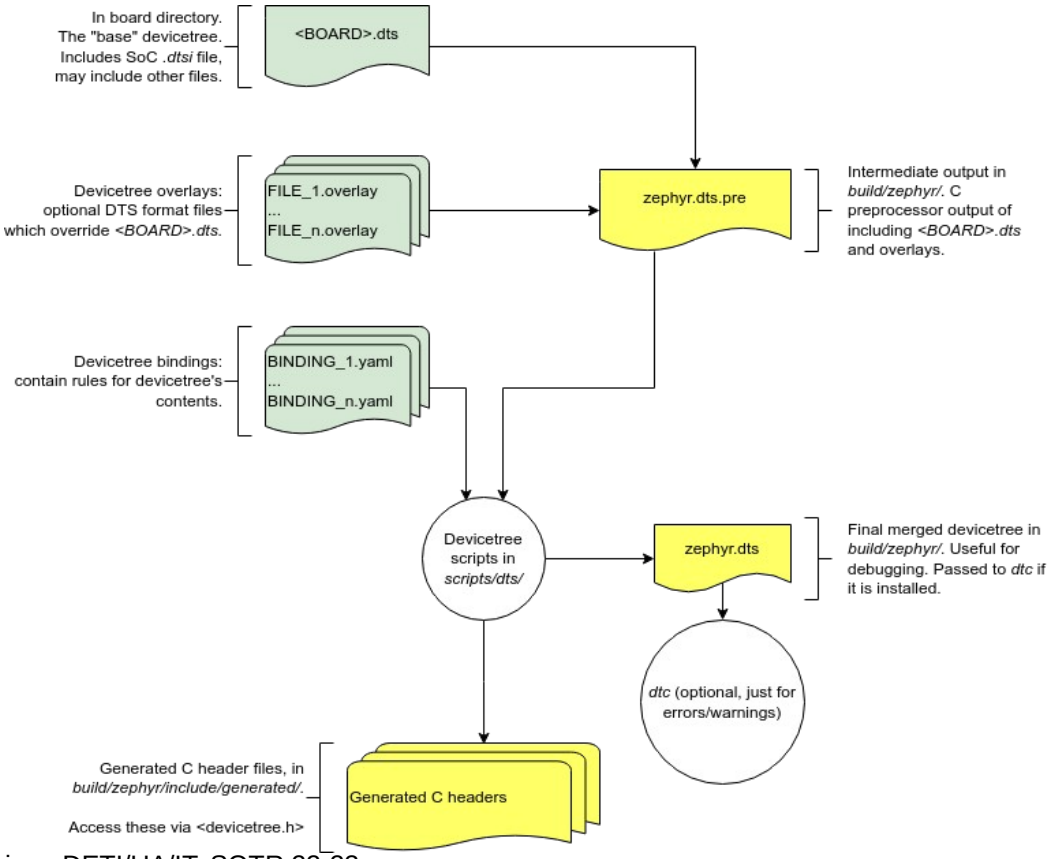
Syntax and Structure

- Aliases and chosen nodes
  - There are two additional ways beyond node labels to refer to a particular node without specifying its entire path:
    - by alias, or by chosen node.
  - Example devicetree which uses both:

```
/dts-v1/;

/ {
     chosen {
             zephyr,console = &uart0;
     };

     aliases {
             my-uart = &uart0;
     };

     soc {
             uart0: serial@12340000 {
                     ...
             };
     };
};
```

- The /aliases and /chosen nodes do not refer to an actual hardware device.
- Their purpose is to specify other nodes in the devicetree.
  - “my-uart” is an alias for the node with path /soc/serial@12340000
- Aliases are often used to override particular hardware.
  E.g. the “Blinky” demo uses it to adapt to LEDs on different ports

Syntax and Structure



- Devicetree input (green) and output (yellow) files
- Overlays are often used to extend/cutsomize the base hardware that comes e.g. with a devkit

Devicetree access from C

- There are many different ways of accessing the DT from C
  - E.g. it is possible to get a Node Identifier: By path, By node label, By alias, By instance number, By chosen node, ...
- To reduce complexity we will focus on one single method.
- Example: blink LED0, which is connected to P0.13

```
...
#define GPIO0_NID DT_NODELABEL(gpio0)
...
void main(void) {

    /* Local vars */
    const struct device *gpio0_dev;         /* Pointer to GPIO device structure */
    ...
    /* Bind to GPIO 0 */
    gpio0_dev = device_get_binding(DT_LABEL(GPIO0_NID));
    if (gpio0_dev == NULL) {
        printk("Failed to bind to GPIO0\n\r");
      return;
    }
    else {
        printk("Bind to GPIO0 successfull \n\r");
    }
```

Devicetree access from C

- Despite not strictly related with DTs, let us take the opportunity to show the libraries provided by Zephyr/Nordic

```
….
/* Configure PIN */
ret = gpio_pin_configure(gpio0_dev, BOARDLED1, GPIO_OUTPUT_ACTIVE);
if (ret < 0) {
    printk("gpio_pin_configure() failed with error %d\n\r", ret);
  return;
}

/* Blink loop */
while(1) {

    /* Toggle led status */
    if(ledstate == 0)
        ledstate = 1;
    else
        ledstate = 0;
    gpio_pin_set(gpio0_dev, BOARDLED1, ledstate);

    /* Pause  */
    k_msleep(BLINKPERIOD_MS);
}
```

Devicetree access from C

- The board DTS is available as part of the VS Code or emStudio project, so the programmer can analyze it to check for the available hardware, labels, etc.
  - Sample contents of zephyr.dts:

```
/dts-v1/;
/ {
       #address-cells = < 0x1 >;
       #size-cells = < 0x1 >;
       model = "Nordic nRF52840 DK NRF52840";
       compatible = "nordic,nrf52840-dk-nrf52840";
       chosen {
              zephyr,entropy = &cryptocell;
              zephyr,flash-controller = &flash_controller;
              zephyr,console = &uart0;
…

aliases {
              led0 = &led0;
              led1 = &led1;
              led2 = &led2;
...

gpio0: gpio@50000000 {
                     compatible = "nordic,nrf-gpio";
                     gpio-controller;
                     reg = < 0x50000000 0x200 0x50000500 0x300 >;
...
```

Devicetree access from C

- Please note that libraries must also be enabled in prj.conf
- E.g.

```
CONFIG_PRINTK=y
CONFIG_HEAP_MEM_POOL_SIZE=256
CONFIG_ASSERT=y
CONFIG_GPIO=y
CONFIG_PWM=y
CONFIG_ADC=y
CONFIG_TIMING_FUNCTIONS=y
CONFIG_USE_SEGGER_RTT=y
CONFIG_RTT_CONSOLE=n
CONFIG_UART_CONSOLE=y
```

Last notes

- This a complex topic, with many details
- It is necessary to acquire at least a basic understanding to be able to “navigate” on the framework, identifying the peripherals, attributes, etc.
- Must resort heavily to:
  - Documentation (see Bibliography)
  - Examples
    - The toolchain brings many examples that illustrate the use of many of the peripherals

Bibliography

- Nordic nrF Connect SDK documentation
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/guides/dts/index.html
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/reference/devicetree/index.html
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/guides/dts/bindings.html#dt-bindings
  - https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/zephyr/guides/dts/api-usage.html#dt-from-c

- Devicetree Specification, Release v0.3-40-g7e1cc17
  - https://www.devicetree.org/specifications/

- Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support
  - https://elinux.org/images/f/f9/Petazzoni-device-tree-dummies_0.pdf