CHAPTER T1

Zephyr Development with DeviceTree

한동대학교-

테크노니아

마이크로프로세서응용

Agenda

- Kconfig
- Devicetree
- Input Output Files
- Access from C
- Examples

2 Configuration Frameworks

Kconfig

- System configuration
  - Enable or disable global features
- Conditional Compilation
  - In C ode or CMake
- Set default values
- Kernel tuning
- Options visualization
  - menuconfig/guiconfig

Devicetree

- Hardware description
  - Details about devices
  - Peripheral Configuration
  - Memory mappings
  - Interrupt lines
- Platform agnostic
  - Facilitates firmware portability across different hardware platforms by abstracting hw specific details

Build-time configuration

2 Roles

Application Developer

Customize your specific application

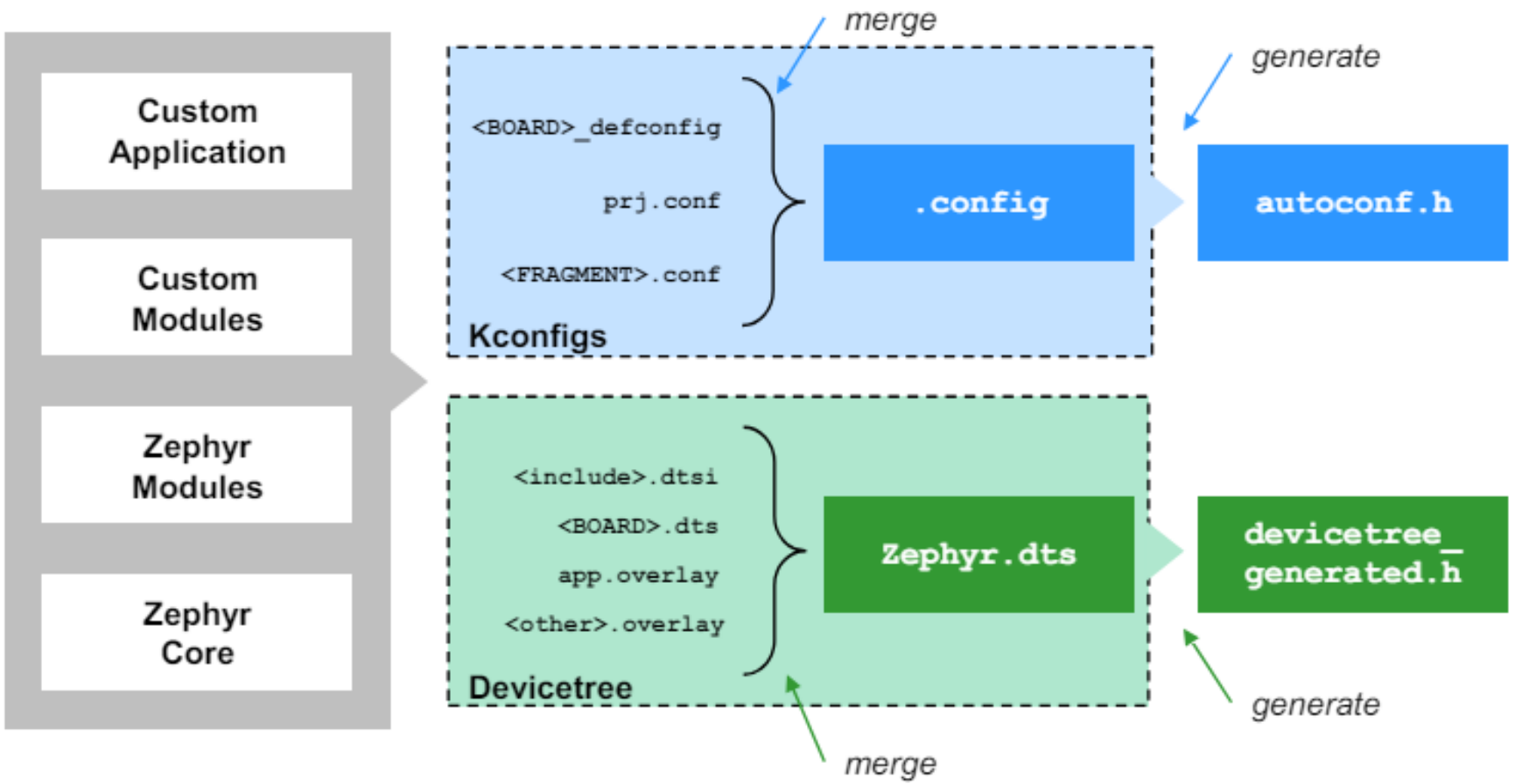
- Configure the environment to suit the needs of a particular project
- Set the values in one of the used files
  - One standard file per project
    - ✔ prj.conf and app.overlay
  - Custom extra overlay/fragment files

Platform developer

Extend Zephyr's Capability

- Introduce new options
  - when adding support for new features, drivers, or modules
- Define new configuration parameters
  - Users can later set according to their project requirements
- Set the values in one of the used files
  - One default file per board
  - Some other files

Overview


Custom Application
Custom Modules
Zephyr Modules
Zephyr Core
merge
<BOARD>_defconfig
prj.conf
<FRAGMENT>.conf
Kconfigs
.config
generate
autoconf.h
<include>.dtsi
<BOARD>.dts
app.overlay
<other>.overlay
Devicetree
Zephyr.dts
devicetree_generated.h
generate
merge

Examples

Kconfig

- In a conf file :

```
CONFIG_SERIAL=y
CONFIG_UART_MCUX_LPUART=y
```

- In CMake :

```
zephyr_library_source_ifdef(
    CONFIG_UART_MCUX_LPUART
    uart_mcux_lpuart.c)
```

- In source code

```
#ifdef CONFIG_UART_MCUX_LPUART
/* some code */
#endif
```

Devicetree

- In devicetree source or overlay

```
flexcomm4_lpuart4: lpuart@b4000 {
  compatible = “nxp,kinetis-lpuart”;
  reg = <0xb4000 0x1000>;
  current-speed = <115200>;
  status = “okay”;
};
```

- In source code

```
DT_INST_PROP(idx,current_speed)
```

Kconfig

Kconfig

- Kconfig options
- They can be used to :
  - Enable or disable specific features in the application
  - Define default values for configuration options
  - Set boundaries, such as minimum or maximum possible values
  - Configure protocols
  - Fine-tune the kernel and scheduler
  - Enable complex conditional configurations without the need for manual adjustments
- Limitations
  - They are not suitable for configuring specific devices or declaring device instances
  - Designed for global configuration settings rather than fine grained device-specific settings

Kconfig advantages

- Flexibility :
  - Customize the project in a convenient way with desired functionalities
- Modularity :
  - Modular code organization by enabling/disabling needed features
- Simplify Configuration Management :
  - A centralized and structured approach to manage project configurations.
- Consistency :
  - Enforce a consistent configuration approach across different projects

Initial configuration file

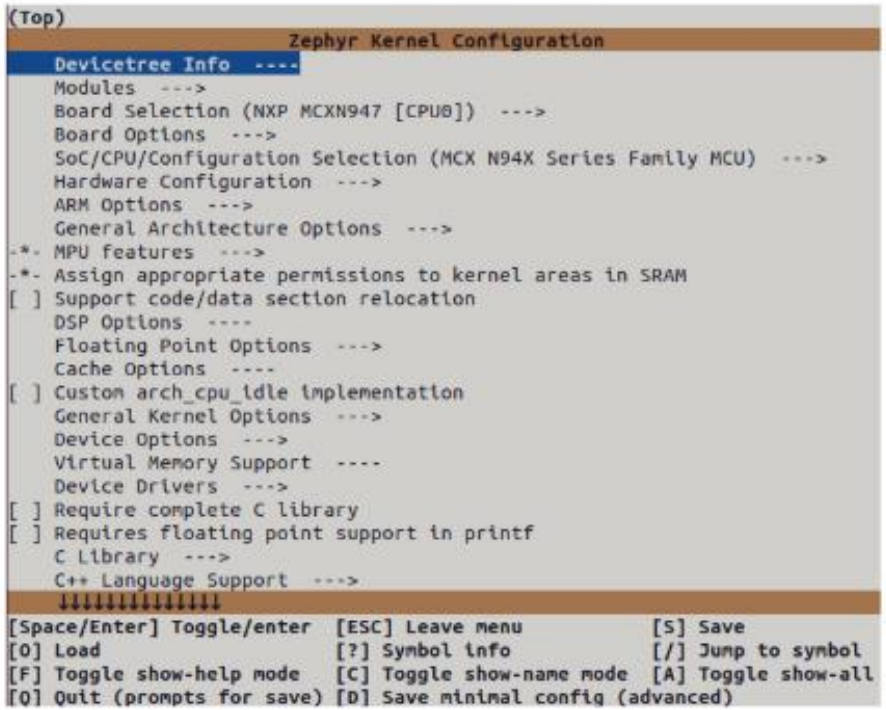
- The application must be configured before being built
- The final configuration is stored in **build/zephyr/.config**
- The initial .config is generated from merging several files :
  - The default config :
    - ✔ **<BOARD>_defconfig** (e.g. **nrf52840dk_nrf52840_defconfig**)
  - prj.conf
    - ✔ Only one file per project
  - Extra config fragment
    - ✔ Any file listed in this variable : **EXTRA_CONF_FILE**

Application developer's role

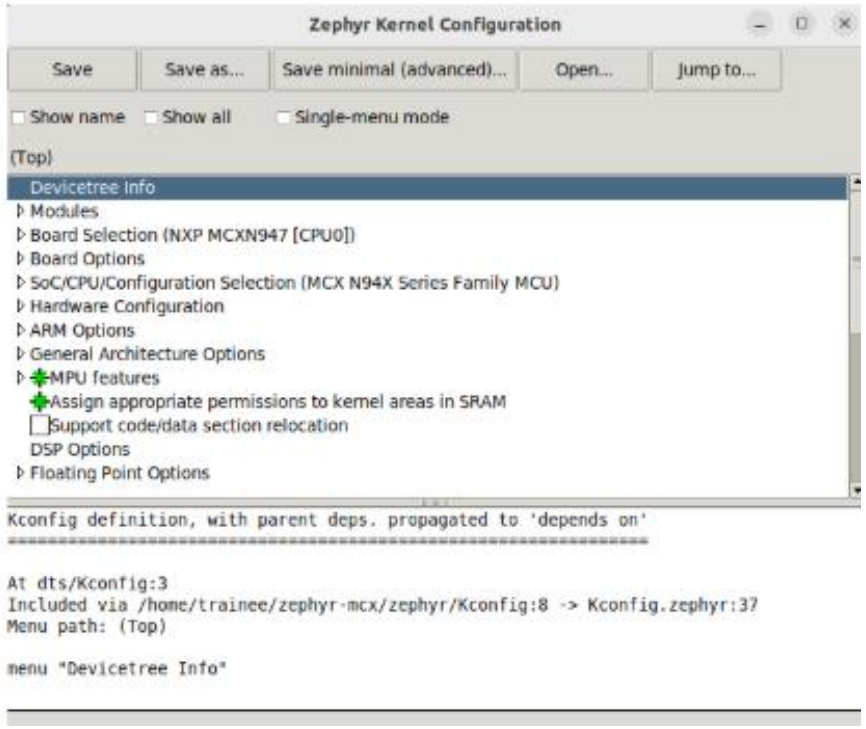
- The vendor will provide a defconfig that has some options enabled by default
  - minimum necessary settings
- The developer usually needs to enable additional features
- The .config can be modified temporarily using and interactive tool
  - Modifications will be discarded after deleting the build directory (pristine build)
    - ✔ $west build –t pristine
- To make the modification persistent, you should place your options either in prj.conf or in an extra configuration fragment
  - Syntax: CONFIG_<symbol name>=<value> (e.g. CONFIG_GPIO=y)

Interactive Kconfig configuration tools

menuconfig



guiconfig



Generation

.config

```
CONFIG_TRACING=y
# CONFIG_TRACING_NONE is not set
CONFIG_PERCEPIO_TRACERECORDER=y
# CONFIG_SEGGER_SYSTEMVIEW is not set
# CONFIG_TRACING_CTF is not set
# CONFIG_TRACING_TEST is not set
# CONFIG_TRACING_USER is not set
# CONFIG_TRACING_SYNC is not set
CONFIG_TRACING_ASYNC=y
CONFIG_TRACING_THREAD_STACK_SIZE=1024
CONFIG_TRACING_THREAD_WAIT_THRESHOLD=100
CONFIG_TRACING_BUFFER_SIZE=2048
CONFIG_TRACING_PACKET_MAX_SIZE=32
CONFIG_TRACING_BACKEND_UART=y
# CONFIG_TRACING_BACKEND_RAM is not set
# CONFIG_TRACING_HANDLE_HOST_CMD is not set
CONFIG_TRACING_CMD_BUFFER_SIZE=32
# CONFIG_TRACING_OBJECT_TRACKING is not set
```

autoconf.h

```
#define CONFIG_TRACING 1
#define CONFIG_PERCEPIO_TRACERECORDER 1
#define CONFIG_TRACING_ASYNC 1
#define CONFIG_TRACING_THREAD_STACK_SIZE 1024
#define CONFIG_TRACING_THREAD_WAIT_THRESHOLD 100
#define CONFIG_TRACING_BUFFER_SIZE 2048
#define CONFIG_TRACING_PACKET_MAX_SIZE 32
#define CONFIG_TRACING_BACKEND_UART 1
#define CONFIG_TRACING_CMD_BUFFER_SIZE 32
#define CONFIG_TRACING_SYSCALL 1
#define CONFIG_TRACING_THREAD 1
#define CONFIG_TRACING_WORK 1
#define CONFIG_TRACING_ISR 1
#define CONFIG_TRACING_SEMAPHORE 1
#define CONFIG TRACING MUTEX 1
```

Permanent config

- To create a permanent config:
  - prj.conf
    - ✔ Only one file per project
  - config fragment (overlay)
    - ✔ Put the options you want to set into a file
      - Either write them directly or use configuration tools
    - ✔ In the main CMakeLists.txt: set(EXTRA_CONF_FILE path/to/my.conf)
      - Note: it should be added before find_package(Zephyr)

Application developer's role

- The vendor will provide a defconfig that has some options enabled by default
  - minimum necessary settings
- The developer usually needs to enable additional features
- The .config can be modified temporarily using and interactive tool
  - Modifications will be discarded after deleting the build directory (pristine build)
    - ✔ $west build –t pristine
- To make the modification persistent, you should place your options either in prj.conf or in an extra configuration fragment
  - Syntax: CONFIG_<symbol name>=<value> (e.g. CONFIG_GPIO=y)

Devicetree

Devicetree in Zephyr

- Single source for hardware information
  - Device drivers obtain configurable hardware descriptions from the devicetree
  - New device drivers use devicetree APIs to create devices based on hardware configurations
- Advantages of device tree in Zephyr
  - Configurability
    - ✔ Devicetree enables hardware descriptions to be easily configurable
    - ✔ No need to change C source code to reconfigure devices
  - Proven concept
    - ✔ A standardized format used by other projects like Linux, u-boot,....

Devicetree in Zephyr

- Devicetree is a description of hardware with text.
  - Contained in *. dts file (**d**evice**t**ree **s**ource)
  - a tree structure where each node has device's characteristics
  - DTS
    - ✔ organizes the hardware outline of your project
    - ✔ provides hardware reference to Device Driver
    - ✔ defines hardware's initial configuration (initialization).
- Concept comes from Linux but implementation is different.
  - Linux dts is converted into precompiled binary (like DLL)
  - Zephyr dts converted into C header file (compiled with kernel)
    - ✔ Avoid wasted code and data size
    - ✔ Avoid runtime overhead
- More information
  - https://docs.zephyrproject.org/latest/build/dts/

Devicetree Syntax and Structure

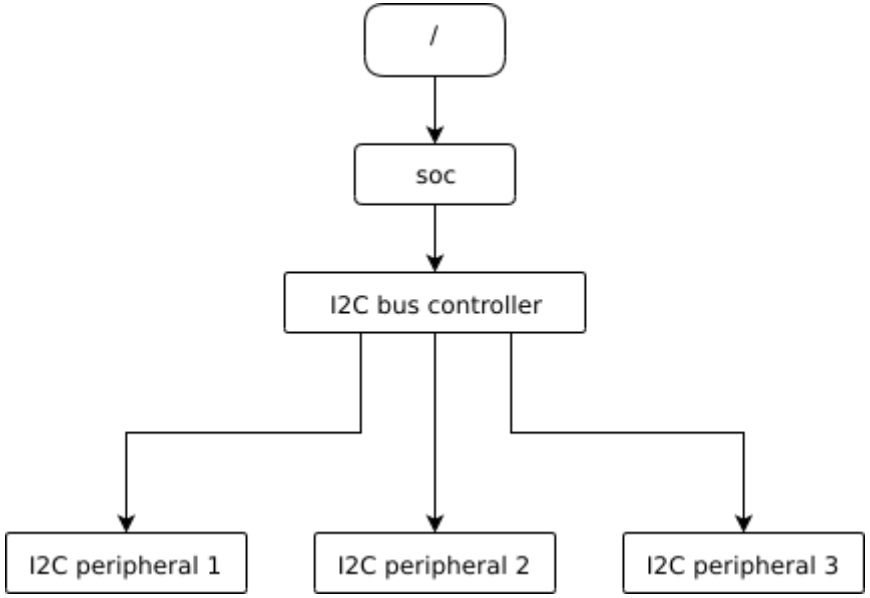
```
/dts-v1/;

/ {
        a-node {
                subnode_nodelabel: a-sub-node {
                        foo = <3>;
                };
        };
};
```

- /dts-v1/ : DTS syntax version 1
- Node : we have 3 nodes
  - root node : “/”
  - “a-node” : child of root node
  - “a-sub-node” : child of “a-node”
    - ✔ Subnode_nodelabel is a node label
      - Node label is a shorthand to refer the node
  - A node has a path like Unix file path notation
    - ✔ Full path to a-sub-node => **/a-node/a-sub-node**
  - A node can have properties
    - ✔ Name/value pair
    - ✔ 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 < and >.

Devicetree reflects hardware

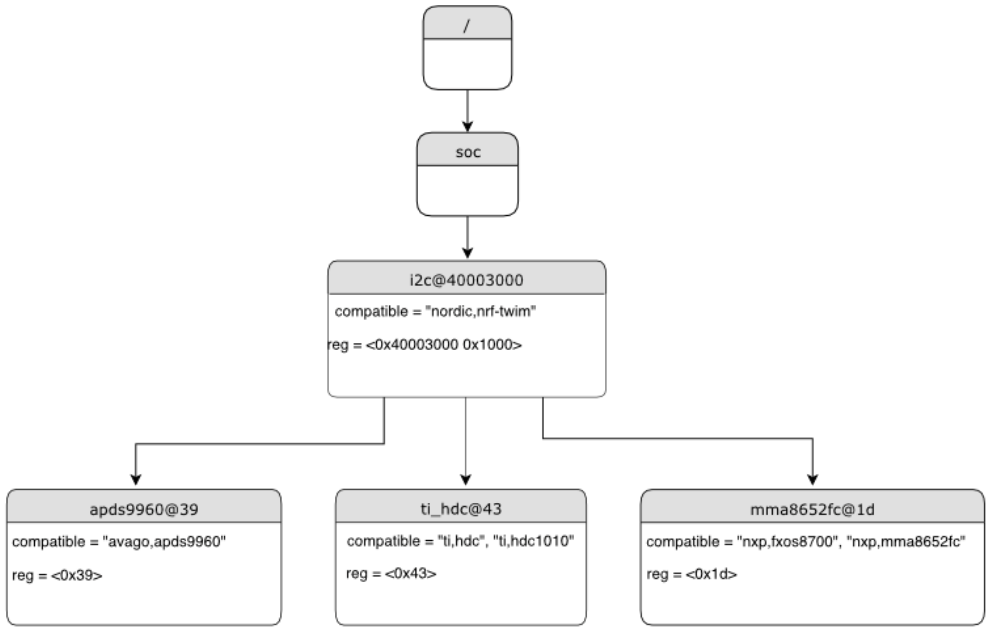
- Devicetree hierarchy reflects the hardware's physical layout
- Example
  - A board has a soc which has an I2C bus controller.
  - The I2C controller has three I2C peripherals.

```
/dts-v1/;

/ {
        soc {
                i2c-bus-controller {
                        i2c-peripheral-1 {
                        };
                        i2c-peripheral-2 {
                        };
                        i2c-peripheral-3 {
                        };
                };
        };
};
```



Properties in practice

- Properties describe (configure) the hardware(node).
  - I2C peripheral has a property “reg” : address of the bus




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

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

Unit address : See next page

Unit address

- Unit address an optional part of node name
  - In the address space of is parent node.
    - ✔ i2c@40003000 : 40003000 is address space of the “soc”

Memory-mapped peripheral	The peripheral’s register map base address	i2c@40003000
I2C peripheral	The peripheral’s address on the I2C bus	apds9960@39
SPI peripheral	The peripheral’s chip select line number.	
Memory	The physical start address (RAM)	memory@2000000
Memory-mapped flash	The physical start address (flash)	flash@8000000
Fixed flash partitions	the devicetree stores a flash partition table.	

Q) What is the base address of the node partition@20000 ?
A) 0x8020000

```
flash@8000000 {
    /* ... */
    partitions {
        partition@0 { /* ... */ };
        partition@20000 {  /* ... */ };
        /* ... */
    };
};
```

Unit address

- Below shows a part of a devicetree
  - Describes a flash module (flash@8000000) which has two paritions
- Quiz : What is the base address of the node partition@20000?

```
flash@8000000 {
    /* ... */
    partitions {
        partition@0 { /* ... */ };
        partition@20000 {  /* ... */ };
        /* ... */
    };
};
```

- Answer : 0x8020000
  - 0x8020000 = 0x8000000 + 0x20000
  - Since the node partition@20000 is a child node of flash@8000000
    - ✔ Unit address is in the address space of is parent node.

Some important Properties

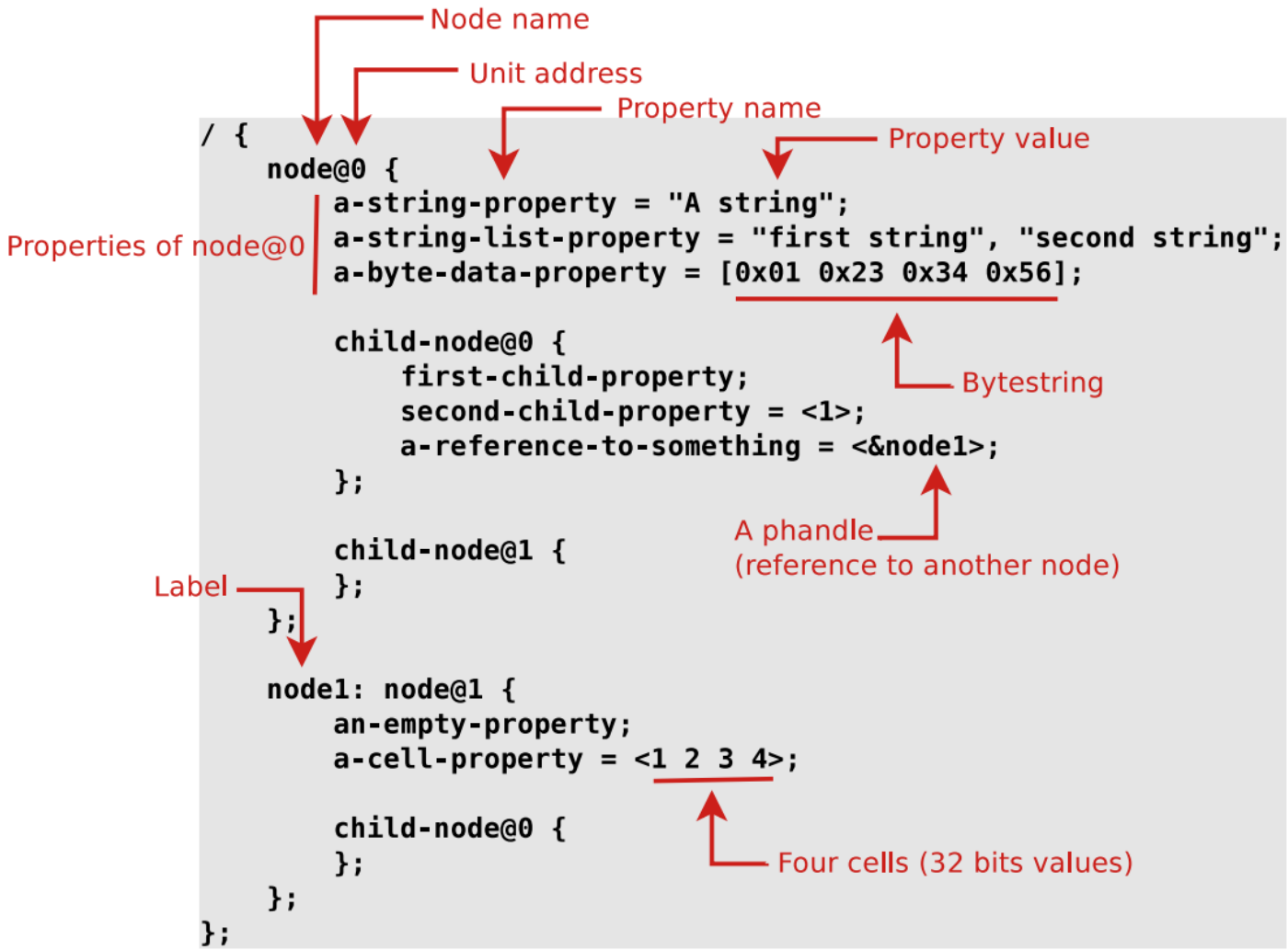
compatible	The name of the hardware device. **“vendor, device”** format like **"nordic,nrf52840dk-nrf52840"** vendor : refer dts/bindings/vendo-prefixes.txt device : from datasheet Build system uses the compatible property to find the right bindings for the node.
reg	A sequence of (address, length) pairs. Each pair is a “register block”. Values are conventionally written in hex. -. Devices accessed via **memory-mapped I/O registers** (like i2c@40003000): address is the base address of the I/O register space. 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.
status	A string which describes whether the node is enabled. "okay“ or "disabled“ (*) “okay” means enabled.
interrupts	Information about interrupts generated by the device, encoded as an array of one or more *interrupt specifiers*.

Property Values

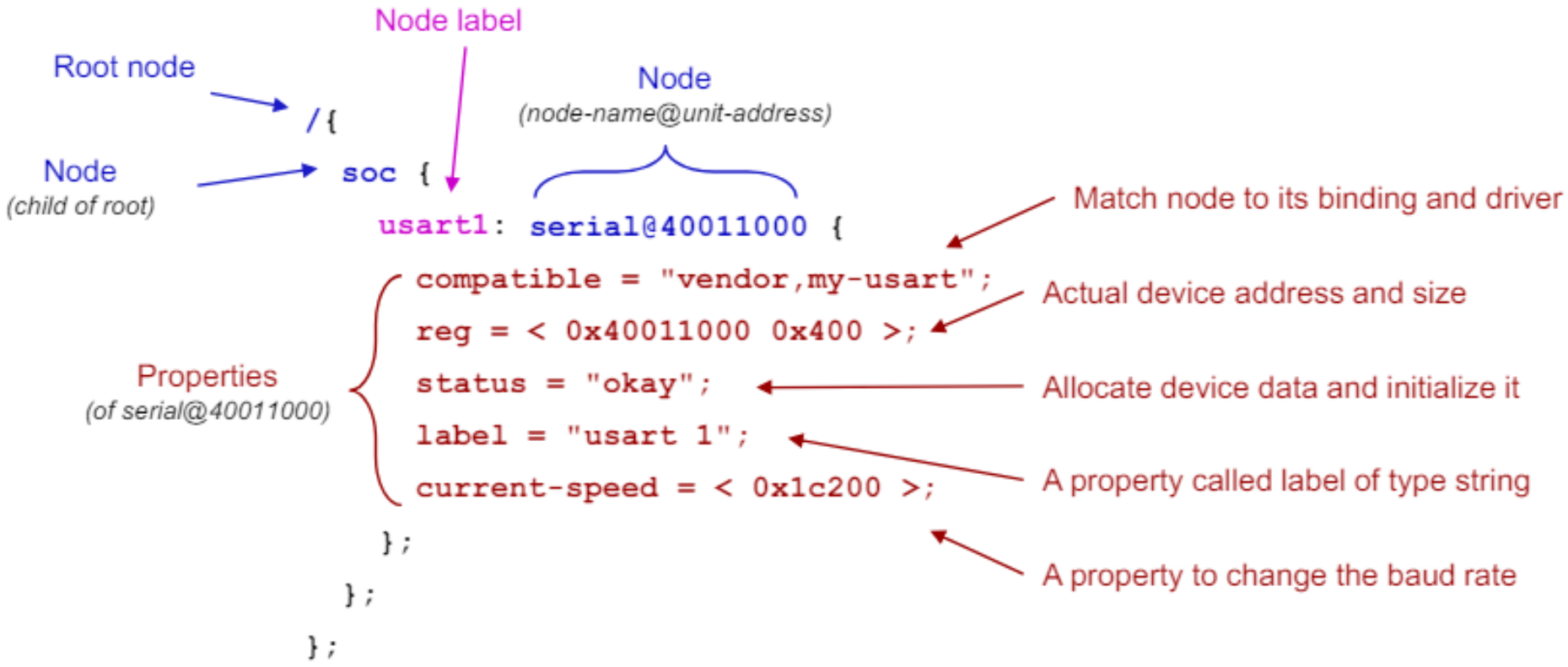
Property type	How to write	Example
string	Double quoted	a-string = "hello, world!";
int	between angle brackets (< and >)	an-int = <1>;
boolean	for true, with no value (for false, use /delete-property/)	my-true-boolean;
array	between angle brackets (< and >), separated by spaces	foo = <0xdeadbeef 1234 0>;
uint8-array	in hexadecimal *without* leading 0x, between square brackets ([and]).	a-byte-array = [00 01 ab];
string-array	separated by commas	a-string-array = "string one", "string two", "string three";
phandle	between angle brackets (< and >)	a-phandle = <&mynode>;
phandles	between angle brackets (< and >), separated by spaces	some-phandles = <&mynode0 &mynode1 &mynode2>;
phandle-array	between angle brackets (< and >), separated by spaces	a-phandle-array = <&mynode0 1 2>, <&mynode1 3 4>;

(*) a phandle (pointer handle) is a 32bit value associated with a node and contains a pointer to another node.

Glance of Devicetree Syntax



Glance of Devicetree Syntax



Aliases and chosen nodes

- /aliases and /chosen provides a shortcut to a node
  - ‘my-uart’ is an alias of ‘/soc/serial@12340000’
  - ‘/chosen’ node to define system-wide values

```
/dts-v1/;
/ {
    chosen {
        zephyr,console = &uart0;
    };
    aliases {
        my-uart = &uart0;
    };
    soc {
        uart0: serial@12340000 {
            ...
        };
    };
};
```

Devicetree Bindings

- Why?
  - Bindings provide the types of the properties used
  - They declare requirements and provide semantic information
- Bindings are YAML files
  - The name of the file should match the compatible node
  - Each file should declare the requirements for each property in that node
- Devicetree bindings define the compatible property
  - in dts/bindings folder, as YAML files
  - Each devicetree node must have a '**compatible**' property
    - ✔ With compatible property value, a node is matched with its binding
  - With binding, validate devicetree node and generate header file

Devicetree Bindings

.

```
/* Node in a DTS file */
bar-device {
    compatible = "foo-company,bar-device";
    num-foos = <3>;
};
```

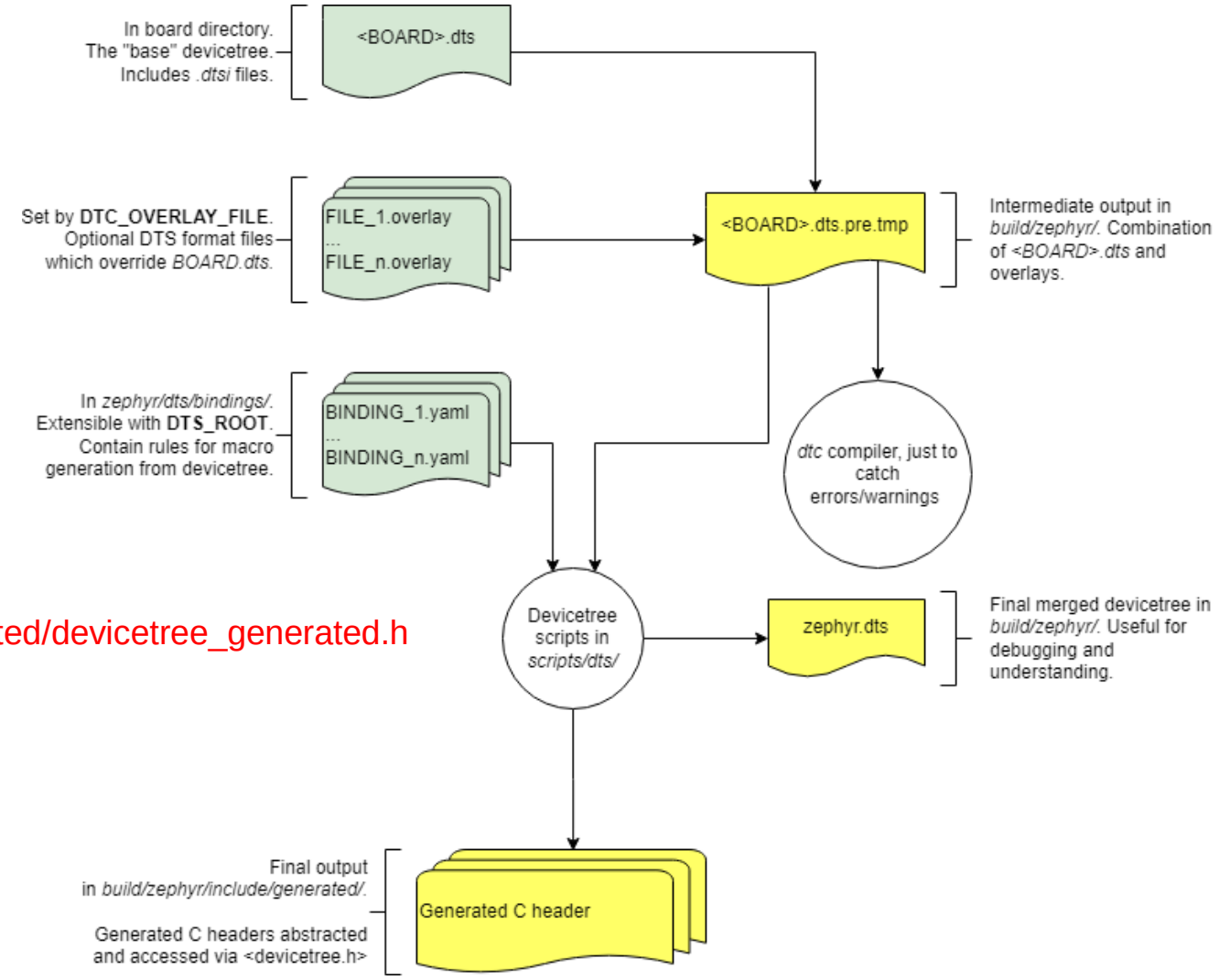
```
# A YAML binding matching the node

compatible: "foo-company,bar-device"

properties:
  num-foos:
    type: int
    required: true
```

https://docs.zephyrproject.org/latest/build/dts/bindings-intro.html#a-simple-example

From *.dts to C header file

- Input Files (green)
  Sources (.dts)
  Includes (.dtsi)
  Overlays(.overlay)
  Bindings (.yaml)

- Output Files (yellow)
  build/zephyr/zephyr.dts.pre
  build>/zephyr/include/generated/devicetree_generated.h
  build/zephyr/zephyr.dts



Input and Output Files

Input Files

- BOARD.dts : every supported board has a BOARD.dts
  zephyr/boards/arm/nrf52840dk_nrf52840/nrf52840dk_nrf52840.dts
- BOARD.dts includes one or more .dtsi files.
  .dtsi files describe CPU or SOC
  nrf52840dk_nrf52840.dts includes nrf52840_qiaa.dtsi
- BOARD.dts can be extended or modified using overlays.
  Zephyr applications can use overlays (app.overlay)
  (1) to enable/disable a peripheral that is disabled/enabled by default.
  (2) when defining Shields
  Zephyr build system combines BOARD.dts and.overlay files by concatenating
  Putting the contents of the .overlay files last allows them to override BOARD.dts
- Devicetree bindings (YAML files) contain rules for macro generation.

Output Files

- build/zephyr/zephyr.dts.pre : preprocessed DTS source. Intermediate file.
  We can check combination of .dts and .overlay
- build/zephyr/include/generated/devicetree_generated.h
  The generated macros and additional comments.
  It will be included by devicetree.h You don't need to include it directly.
- build/zephyr/zephyr.dts
  The final merged devicetree. It is useful for debugging.

Application developer's role

- Any developer will need to customize the default devicetree
  - usually done in an overlay file
    - ✔ app.overlay
    - ✔ Adding an extra overlay
- Use devicetree data in source code
  - Retrieve the device pointer (struct device *)
    - ✔ needed for performing operations on the specific device
  - Retrieve property values
    - ✔ Accessing various hardware configuration parameters

Example : devicetree_generated.h

```
/* Node's full path: */
#define DT_N_S_soc_S_uart_40028000_PATH "/soc/uart@40028000"

/* Node's name with unit-address: */
#define DT_N_S_soc_S_uart_40028000_FULL_NAME "uart@40028000"

/* Node parent (/soc) identifier: */
#define DT_N_S_soc_S_uart_40028000_PARENT DT_N_S_soc

/* Node's index in its parent's list of children: */
#define DT_N_S_soc_S_uart_40028000_CHILD_IDX 48


/* Existence and alternate IDs: */
#define DT_N_S_soc_S_uart_40028000_EXISTS 1
#define DT_N_INST_1_nordic_nrf_uarte  DT_N_S_soc_S_uart_40028000
#define DT_N_NODELABEL_uart1          DT_N_S_soc_S_uart_40028000
#define DT_N_NODELABEL_arduino_serial DT_N_S_soc_S_uart_40028000
```

Access devicetree from source code

- Retrieve the node id
  - DT_PATH(path, to , node)
  - DT_NODELABEL(node_label1)
- Get a property value
  - DT_PROP(node_id, prop)
- Some driver specific macros

  const struct gpio_dt_spec my_led = **GPIO_DT_SPEC_GET**(LED_NODE,gpios)

Devicetree access from C

- How your device tree (**zephyr.dts**) is converted to **devicetree_generated.h**
- **Tree structure -> C macro notation**
  - DT_N means “devicetree node”
  - / becomes S
  - (ex) DT_N_S_foo_S_bar means /foo/bar

```
/{
    parent {
        size = <3>;    /*path : /parent */
        child {
            size = <4>; /*path : /parent/child */
        };
    };
};
```

-. Reference to parent node :
DT_PATH(parent) -> DT_N_S_parent
-. Reference to child node :
DT_PATH(parent, child) -> DT_N_S_parent_S_child
-. Reference to size in child :
DT_N_S_parent_S_child_P_size

https://docs.zephyrproject.org/latest/build/dts/api-usage.html

Devicetree access from C

- Property access with node identifiers

```
#define DT_N_S_parent_P_size 3
#define PARENT_NODE DT_PATH(parent)
```

If you reference its property in your program like

```
DT_PROP(PARENT_NODE, size)
```

Your reference is interpreted as follows:

```
DT_PROP(PARENT_NODE, size)  -> PARENT_NODE ## _P_ ## size
                            -> DT_PATH(parent) ## _P_ ## size
                            -> DT_N_S_parent_P_size
                            -> 3
```

https://docs.zephyrproject.org/latest/build/dts/api-usage.html

Devicetree access from C

- Node identifiers : C macros refer to a node.

By path	DT_PATH() with the node's full path
By node label	DT_NODELABEL()
By alias	DT_ALIAS() /aliases node property
By instance number	DT_INST()
By chosen node	DT_CHOSEN() /chosen node properties.
By parent/child	DT_PARENT() and DT_CHILD()

```
/dts-v1/;

/ {
        aliases {
                sensor-controller = &i2c1;
        };

        soc {
                i2c1: i2c@40002000 {
                        compatible = "vnd,soc-i2c";
                        label = "I2C_1";
                        reg = <0x40002000 0x1000>;
                        status = "okay";
                        clock-frequency = < 100000 >;
                };
        };
};
```

ways to get node identifiers for the i2c@40002000 node:

DT_PATH(soc, i2c_40002000)
DT_NODELABEL(i2c1)
DT_ALIAS(sensor_controller)
DT_INST(x, vnd_soc_i2c) for some unknown number x

Devicetree access from C

- Property access : use proper C macros and APIs according to the node and property.

DT_NODE_HAS_PROP()	Check if a node has a property
DT_PROP(node_id, property)	read basic integer, boolean, string, numeric array, and string array properties.
DT_REG_ADDR(node_id) DT_REG_SIZE(node_id)	Reg properties.
DT_NUM_IRQS(node_id)	Interrupts property
DT_PHANDLE(), DT_PHANDLE_BY_IDX(), or DT_PHANDLE_BY_NAME()	Phandle properties

```
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), clock_frequency)  /* expands to 1 */
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), not_a_property)   /* expands to 0 */
```

```
DT_PROP(DT_PATH(soc, i2c_40002000), clock_frequency)  /* This is 100000, */
DT_PROP(DT_NODELABEL(i2c1), clock_frequency)          /* and so is this, */
DT_PROP(DT_ALIAS(sensor_controller), clock_frequency) /* and this. */
```

struct device *

- All devices should have an instance of type struct device
  - Zephyr device driver model
  - They are typically defined in the devicetree
- Common methods to retrieve a device reference:
  - const struct device * defvice_get_binding(const char *name)
  - It can be used for devices defined within or without a device tree
  - DEVICE_DT_GET(node_id)
  - Get a device reference from a devicetree node identifier

Example: I2C and Node Label

- While we can use DT_PATH to get a node in DT
  - we reference the DT path to that device using DT_NODELABEL in code

```
&i2c1 {
    compatible = “nordic, nrf-twim”;
        status = “okay”;
        sda-pin = <26>;
        scl-pin = <27>;
};
#define  I2C_1_NODE  DT_NODELABEL(i2c1);

const struct device *i2c_dev = DEVICE_DT_GET(I2C_1_NODE);


if(!device_is_ready(i2c_dev)) {                    /* Check for validity */
                printk(“Cannot bind i2c device\n”);
                return -EIO;
}
/* Configure! GPIO_OUTPUT_ACTIVE is combined with flags in DT*/
err = pm_device_action_run(i2c_dev, PM_DEVICE_ACTION_SUSPEND);
                if(err) {
                printk(“Can’t power off : %d\n”, err);
                return err;
}
```

Example: GPIO

- Device tree GPIO definitions are composed of 3 parts
  - port, pin number, flasgs used
- Using a GPIO : using gpio_pin_configure_dt()

```
/{
        zephyr,user {
                latch-en-gpios = <&gpio0 31 GPIO_ACTIVE_HIGH>;
        };
};
```

```
static const struct gpio_dt_spec latch_en = GPIO_DT_SPEC_GET( DT_PATH(zephyr_user), latch_en_gpios );

if(!device_is_ready(latch_en.port)) {                    /* Check port first */
        LOG_ERR("GPIO device not ready");
        return -EINVAL;
}
/* Configure! GPIO_OUTPUT_ACTIVE is combined with flags in DT*/
err = gpiio_pin_configure_dt(&latch_en, GPIO_OUTPUT_ACTIVE);
if(err) {
        LOG_ERR("failed to configure GPIO (err %d)", err);
        return err;
}
/* ……  */
gpio_pin_set_dt(&latch_en, 1);  /* Turn the gpio pin on */

int ret = gpio_pin_get_dt(&latch_en); /* Read pin value it input… 0 or 1 */
```

Example: Aliases

- Aliases are also useful for any device type.
- They are often used to help point to common system aliases that are used across samples
  - Led0 for the default led on your board
  - Sw0 for the default switch

```
aliases {
        led0 = &blue_led;
        bootloader-led0 = &blue_led;
        pwm-led0 = &pwm_led0;
        sw0 = &button0;
};
buttons {
        compatible = “gpio-keys”;
        button0:button_0 {
        gpios = <&gpio0 12 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
        label = “Switch 1”;
        };
};
```

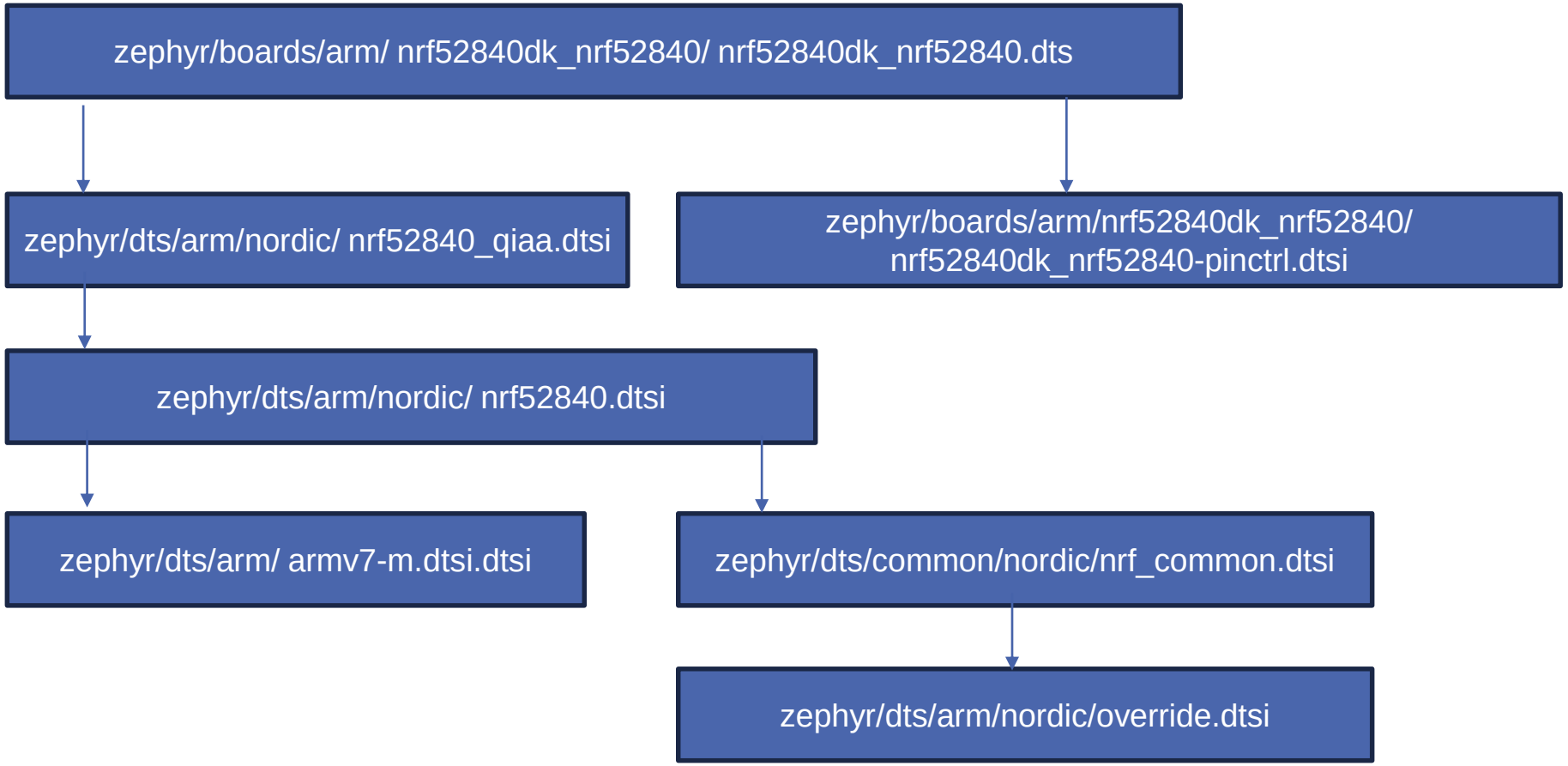
```
static const struct gpio_dt_spec sw0 = GPIO_DT_SPEC_GET(DT_ALIAS(sw0), gpios);
```

Sample Program : nrf52840dk_nrf52840.dts

```
/dts-v1/;
#include <nordic/nrf52840_qiaa.dtsi>
#include "nrf52840dk_nrf52840-pinctrl.dtsi"

/ {
	model = "Nordic nRF52840 DK NRF52840";
	compatible = "nordic,nrf52840-dk-nrf52840";

	chosen {
		zephyr,console = &uart0;
		zephyr,shell-uart = &uart0;
		zephyr,uart-mcumgr = &uart0;
		zephyr,bt-mon-uart = &uart0;
		zephyr,bt-c2h-uart = &uart0;
		zephyr,sram = &sram0;
		zephyr,flash = &flash0;
		zephyr,code-partition = &slot0_partition;
		zephyr,ieee802154 = &ieee802154;
	};

	leds {
		compatible = "gpio-leds";
		led0: led_0 {
			gpios = <&gpio0 13 GPIO_ACTIVE_LOW>;
			label = "Green LED 0";
		};
		led1: led_1 {
			gpios = <&gpio0 14 GPIO_ACTIVE_LOW>;
			label = "Green LED 1";
		};
		led2: led_2 {
			gpios = <&gpio0 15 GPIO_ACTIVE_LOW>;
			label = "Green LED 2";
		};
		led3: led_3 {
			gpios = <&gpio0 16 GPIO_ACTIVE_LOW>;
			label = "Green LED 3";
		};
	};

	pwmleds {
		compatible = "pwm-leds";
		pwm_led0: pwm_led_0 {
			pwms = <&pwm0 0 PWM_MSEC(20) PWM_POLARITY_INVERTED>;
		};
	};

	buttons {
		compatible = "gpio-keys";
		button0: button_0 {
			gpios = <&gpio0 11 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
			label = "Push button switch 0";
		};
		button1: button_1 {
			gpios = <&gpio0 12 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
			label = "Push button switch 1";
		};
```

```
	/* These aliases are provided for compatibility with samples */
	aliases {
		led0 = &led0;
		led1 = &led1;
		led2 = &led2;
		led3 = &led3;
		pwm-led0 = &pwm_led0;
		sw0 = &button0;
		sw1 = &button1;
		sw2 = &button2;
		sw3 = &button3;
		bootloader-led0 = &led0;
		mcuboot-button0 = &button0;
		mcuboot-led0 = &led0;
		watchdog0 = &wdt0;
		spi-flash0 = &mx25r64;
	};
```

.dts and .dtsi for DK board



What QIAA means?

Code ranges and values

Defined here are the nRF52840 code ranges and values.

<PP>	Package	Size (mm)	Pin/Ball count	Pitch (mm)
QI	aQFN™	7 x 7	73	0.5
QF	QFN	6 x 6	48	0.4
CK	WLCSP	3.544 x 3.607	94	0.35

Table 2. Package variant codes

<VV>	Flash (kB)	RAM (kB)	Access port protection
AA	1024	256	Controlled by hardware
AA-F	1024	256	Controlled by hardware and software

Table 3. Function variant codes

Sample Program : nrf52840_qiaa.dtsi

```
#include <mem.h>
#include <nordic/nrf52840.dtsi>

&flash0 {
    reg = <0x00000000 DT_SIZE_K(1024)>;
};

&sram0 {
    reg = <0x20000000 DT_SIZE_K(256)>;
};

/ {
    soc {
        compatible = "nordic,nrf52840-qiaa", "nordic,nrf52840",
                     "nordic,nrf52", "simple-bus";
    };
};
```

nrf52840.dtsi

```
#include <arm/armv7-m.dtsi>
#include <nordic/nrf_common.dtsi>

/ {
    chosen {
        zephyr,entropy = &cryptocell;
        zephyr,flash-controller = &flash_controller;
    };

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;

        cpu@0 {
            device_type = "cpu";
            compatible = "arm,cortex-m4f";
            reg = <0>;
            #address-cells = <1>;
            #size-cells = <1>;

            itm: itm@e0000000 {
                compatible = "arm,armv7m-itm";
                reg = <0xe0000000 0x1000>;
                swo-ref-frequency = <32000000>;
            };
        };
    };
```

```
    soc {
        ficr: ficr@10000000 {
            compatible = "nordic,nrf-ficr";
            reg = <0x10000000 0x1000>;
            #nordic,ficr-cells = <1>;
            status = "okay";
        };

        uicr: uicr@10001000 {
            compatible = "nordic,nrf-uicr";
            reg = <0x10001000 0x1000>;
            status = "okay";
        };

        sram0: memory@20000000 {
            compatible = "mmio-sram";
        };

        clock: clock@40000000 {
            compatible = "nordic,nrf-clock";
            reg = <0x40000000 0x1000>;
            interrupts = <0 NRF_DEFAULT_IRQ_PRIORITY>;
            status = "okay";
        };
```

nrf52840.dtsi

node	compatible property
ficr: ficr@10000000	compatible = "nordic,nrf-ficr";
uicr: uicr@10001000	compatible = "nordic,nrf-uicr";
sram0: memory@20000000	compatible = "mmio-sram";
clock: clock@40000000	compatible = "nordic,nrf-clock";
power: power@40000000	compatible = "nordic,nrf-power";
radio: radio@40001000	compatible = "nordic,nrf-radio";
uart0: uart@40002000	compatible = "nordic,nrf-uarte";
spi0: spi@40003000	compatible = "nordic,nrf-spim";
i2c1: i2c@40004000	compatible = "nordic,nrf-twim";
spi1: spi@40004000	compatible = "nordic,nrf-spim";
nfct: nfct@40005000	compatible = "nordic,nrf-nfct";
gpiote: gpiote0: gpiote@4000600	compatible = "nordic,nrf-gpiote";

nrf52840.dtsi

adc: adc@40007000	compatible = "nordic,nrf-saadc";
timer0: timer@40008000	compatible = "nordic,nrf-timer";
timer1: timer@40009000	compatible = "nordic,nrf-timer";
timer2: timer@4000a000	compatible = "nordic,nrf-timer";
rtc0: rtc@4000b000	compatible = "nordic,nrf-rtc";
temp: temp@4000c000	compatible = "nordic,nrf-temp";
rng: random@4000d000	compatible = "nordic,nrf-rng";
ecb: ecb@4000e000	compatible = "nordic,nrf-ecb";
ccm: ccm@4000f000	compatible = "nordic,nrf-ccm";
wdt: wdt0: watchdog@40010000	compatible = "nordic,nrf-wdt";
rtc1: rtc@40011000	compatible = "nordic,nrf-rtc";
qdec: qdec0: qdec@40012000	compatible = "nordic,nrf-qdec";
comp: comparator@40013000	compatible = "nordic,nrf-comp";

nrf52840.dtsi

egu0: swi0: egu@40014000	compatible = "nordic,nrf-egu", "nordic,nrf-swi";
egu1: swi1: egu@40015000	compatible = "nordic,nrf-egu", "nordic,nrf-swi";
egu2: swi2: egu@40016000	compatible = "nordic,nrf-egu", "nordic,nrf-swi";
egu3: swi3: egu@40017000	compatible = "nordic,nrf-egu", "nordic,nrf-swi";
egu4: swi4: egu@40018000	compatible = "nordic,nrf-egu", "nordic,nrf-swi";
egu5: swi5: egu@40019000	compatible = "nordic,nrf-egu", "nordic,nrf-swi";
timer3: timer@4001a000	compatible = "nordic,nrf-timer";
timer4: timer@4001b000	compatible = "nordic,nrf-timer";
pwm0: pwm@4001c000	compatible = "nordic,nrf-pwm";
pdm0: pdm@4001d000	compatible = "nordic,nrf-pdm";
acl: acl@4001e000	compatible = "nordic,nrf-acl";
flash_controller: flash-controller@4001e000	compatible = "nordic,nrf52-flash-controller";
ppi: ppi@4001f000	compatible = "nordic,nrf-ppi";

nrf52840.dtsi

mwu: mwu@40020000	compatible = "nordic,nrf-mwu";
pwm1: pwm@40021000	compatible = "nordic,nrf-pwm";
pwm2: pwm@40022000	compatible = "nordic,nrf-pwm";
spi2: spi@40023000	compatible = "nordic,nrf-rtc";
i2s0: i2s@40025000	compatible = "nordic,nrf-i2s";
usbd: usbd@40027000	compatible = "nordic,nrf-usbd";
uart1: uart@40028000	**compatible = "nordic,nrf-uarte";**
qspi: qspi@40029000	compatible = "nordic,nrf-pwm";
pwm3: pwm@4002d000	compatible = "nordic,nrf-pwm";
spi3: spi@4002f000	compatible = "nordic,nrf-spim";
gpio0: gpio@50000000	**compatible = "nordic,nrf-gpio";**
gpio1: gpio@50000300	**compatible = "nordic,nrf-gpio";**
cryptocell: crypto@5002a000	compatible = "nordic,cryptocell", "arm,cryptocell-310";

Sample program: gpios.h

- Read and Write GPIO example :
  - Read from GPIO : Button
  - Write to GPIO : LED

```
/* The devicetree node identifier for the "led0" alias. */
#define LED0_NODE DT_ALIAS(led0)
#if !DT_NODE_HAS_STATUS(LED0_NODE, okay)
#error "Unsupported board: led0 devicetree alias is not defined"
#endif
static const struct gpio_dt_spec led0 = GPIO_DT_SPEC_GET(DT_ALIAS(led0), gpios);

// buttons
#define SW0_NODE DT_ALIAS(sw0)
#if !DT_NODE_HAS_STATUS(SW0_NODE, okay)
#error "Unsupported board: sw0 devicetree alias is not defined"
#endif
static const struct gpio_dt_spec button0 = GPIO_DT_SPEC_GET(SW0_NODE, gpios);
```

Sample program : gpios.c

```
int check_and_configure_gpios() {
    bool ret;
    int conf_ret;

    /* Check if all leds are ready */
    ret = gpio_is_ready_dt(&led0);
    if (!ret) {
        printk("GPIO LED0 is not ready\n");
        return -1;
    }
    conf_ret = gpio_pin_configure_dt(&led0, GPIO_OUTPUT_ACTIVE);
    if (conf_ret < 0) {
        printk("Error %d: failed to configure LED0 pin\n", conf_ret);
        return -1;
    }
}
void check_button0() {
    int val = gpio_pin_get_dt(&button0);
    int ret;
    if (val > 0) {
        ret = gpio_pin_toggle_dt(&led0);
        if (ret < 0) {
            return;
        }
    }
}
```

```
int main(void)
{
    int ret = check_and_configure_gpios();
    if (ret < 0) {
            return 0;
    }
    while (1) {
        check_button0();
        check_button1();
        check_button2();
        check_button3();
         k_msleep(SLEEP_TIME_MS);
    }
    return 0;
}
```

Sample program : Practice

(1) Build and Run sample program : check if button and LED work as intended.
(2) When press a button, LED is not toggled sometimes. Guess the reason of the problem.
Modify the C program to improve the response.
(3) In the given example, button input is obtained by polling.
Modify program to detect button change with interrupt.
(4) In the example, LED is only toggled. (0 or 1)
Change LED behavior to support brightness level by using “pwm-led0”

GPIO C API Functions & Macros

GPIO Related C API functions

- gpio_is_ready_dt(const struct gpio_dt *spec)
- gpio_pin_interrupt_configure_dt(const struct gpio_dt_spec *spec, gpio_flag_t flag)
- gpio_pin_configure_dt (const struct gpio_dt *, pgio_flag_t extra_flag)
- gpio_init_callback(struct gpio_callback *cb, gpio_callback_handler_t, gpio_port_pins_t mask)
- gpio_add_callback(const struct device *port, struct gpio_callback *cb)
- gpio_pin_configure_dt(const struct gpio_dt_spec *spec, gpio_flag_t extra_flag)
- gpio_pin_get_dt(const struct gpio_dt_spec *spec)
- gpio_pin_set_dt(const struct gpio_dt_spec *spec, int value)

static inline int **gpio_pin_configure_dt**(*const struct* gpio_dt_spec ***spec**, gpio_flags_t **extra_flags**)

Configure a single pin from a `gpio_dt_spec` and some extra flags.

This is equivalent to:

```
gpio_pin_configure(spec->port, spec->pin, spec->dt_flags | extra_flags);
```

Parameters:
- **spec** – GPIO specification from devicetree
- **extra_flags** – additional flags

Returns: a value from gpio_pin_configure()

static inline **int** **gpio_pin_set_dt(*const struct* gpio_dt_spec *spec, int value)**

Set logical level of a output pin from a `gpio_dt_spec`.

This is equivalent to:

```
gpio_pin_set(spec->port, spec->pin, value);
```

Parameters:
- **spec** – GPIO specification from devicetree
- **value** – Value assigned to the pin.

Returns: a value from gpio_pin_set()

static inline int **gpio_pin_get_dt**(*const struct* gpio_dt_spec ***spec**)

Get logical level of an input pin from a `gpio_dt_spec`.

This is equivalent to:

```
gpio_pin_get(spec->port, spec->pin);
```

Parameters:
- **spec** – GPIO specification from devicetree

Returns: a value from gpio_pin_get()

static inline int **gpio_pin_toggle_dt**(*const struct* gpio_dt_spec *spec)

Toggle pin level from a `gpio_dt_spec`.

This is equivalent to:

```
gpio_pin_toggle(spec->port, spec->pin);
```

Parameters: • **spec** – GPIO specification from devicetree

Returns: a value from gpio_pin_toggle()

static inline **int** **gpio_pin_interrupt_configure_dt(*const struct* gpio_dt_spec *spec, gpio_flags_t flags)**

Configure pin interrupts from a `gpio_dt_spec`.

This is equivalent to:

```
gpio_pin_interrupt_configure(spec->port, spec->pin, flags);
```

The `spec->dt_flags` value is not used.

Parameters:
- **spec** – GPIO specification from devicetree
- **flags** – interrupt configuration flags

Returns: a value from gpio_pin_interrupt_configure()

static inline void gpio_init_callback(*struct* gpio_callback *callback, gpio_callback_handler_t handler, gpio_port_pins_t pin_mask)

Helper to initialize a struct gpio_callback properly.

Parameters:
- **callback** – A valid Application's callback structure pointer.
- **handler** – A valid handler function pointer.
- **pin_mask** – A bit mask of relevant pins for the handler

- **Callback function**: Also known as an interrupt handler or an Interrupt Service Routine (ISR). It runs asynchronously in response to a hardware or software interrupt. In general, ISRs have higher priority than all threads (covered in Lesson 7). It preempts the execution of the current thread, allowing an action to take place immediately. Thread execution resumes only once all ISR work has been completed

static inline int **gpio_add_callback**(*const struct* device *port, *struct* gpio_callback *callback)

Add an application callback.

Note: enables to add as many callback as needed on the same port.

Note

Callbacks may be added to the device from within a callback handler invocation, but whether they are invoked for the current GPIO event is not specified.

Parameters:
- **port** – Pointer to the device structure for the driver instance.
- **callback** – A valid Application's callback structure pointer.

Returns: 0 if successful, negative errno code on failure.

GPIO Pin flag: Logical state vs. Physical level

- If a pin was configured as Active Low, physical level low will be considered as logical level 1 (an active state), physical level high will be considered as logical level 0 (an inactive state).
- GPIO_ACTIVE_LOW: GPIO pin is active (has logical value ‘1’) in low state
- GPIO_ACTIVE_HIGH: GPIO pin is active (has logical value ‘1’) in high state.

GPIO pin Configuration flag (Bias, Drive)

- GPIO_PULL_UP: Enables GPIO pin pull-up
- GPIO_PULL_DOWN: Enables GPIO pin pull-down

- GPIO_OPEN_DRAIN: Configures GPIO output in open drain mode (wired AND).
- GPIO_OPEN_SOURCE: Configures GPIO output in open source mode (wired OR).

GPIO input/output Configuration flags

- GPIO_INPUT: Enables pin as input
- GPIO_OUTPUT : Enables pin as output
- GPIO_DISCONNECTED: Disables pin for both input and output

- GPIO_OUTPUT_LOW: Configure pin a output and initialize it to a Low state
- GPIO_OUTPUT_HIGH: COnfigure pin as output and initialize it to a High state
- GPIO_OUTPUT_INACTIVE: Configure pin a output and initialize it to a logic 0
- GPIO_OUTPUT_ACTIVE : Configure pin as output and initialize it to a logic 1

GPIO Interrupt Configuration flags

- GPIO_INT_DISABLE : Disable GPIO pin interrupt
- GPIO_INT_LEVEL_LOW: configure GPIO interrupt to be triggered on pin physical level low and enable it
- GPIO_INT_LEVEL_HIGH : configure GPIO interrupt to be triggered on pin physical level high and enable it
- GPIO_INT_LEVEL_INACTIVE: configure GPIO interrupt to be triggered on pin logical 0 and enable it
- GPIO_INT_LEVEL_ACTIVE: configure GPIO interrupt to be triggered on pin logical 1 and enable it
- GPIO_INT_EDGE_RISING : Configures GPIO interrupt to be triggered on pin rising edge and enables it
- GPIO_INT_EDGE_FALLING: Configures GPIO interrupt to be triggered on pin falling edge and enables it
- GPIO_INT_EDGE_BOTH: Configures GPIO interrupt to be triggered on pin rising or falling edge and enables it.
- GPIO_INTEDGE_TO_INACTIVE: Configures GPIO interrupt to be triggered on pin state change to logical level 0 and enables it.
- GPIO_INT_EDGE_ACTIVE: Configures GPIO interrupt to be triggered on pin state change to logical level 1 and enables it.

Supplementary for DTS to C Conversion

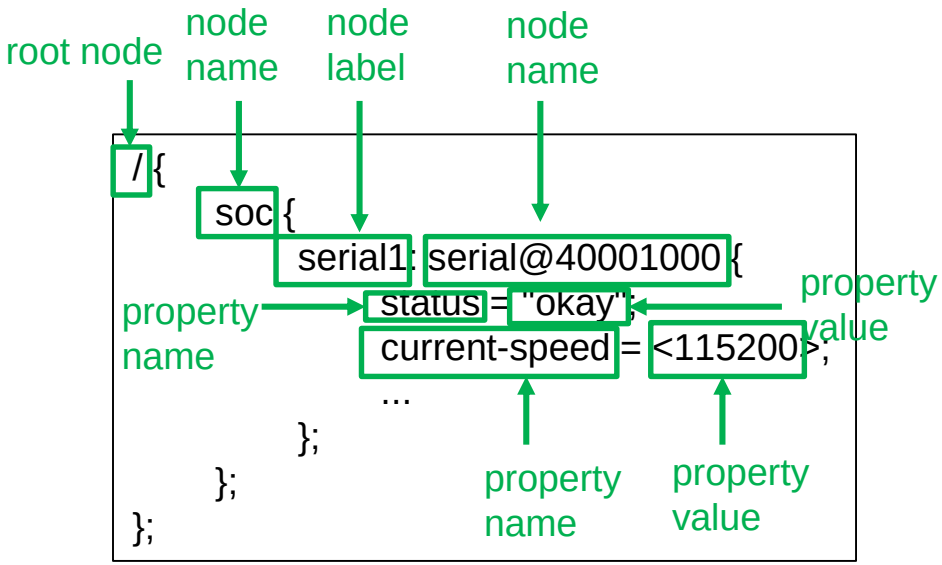
Node identifier
Node name and node label
Node instance index
Node property and value
devicetree source APIs (Macro) List
Configuration with Kconfig

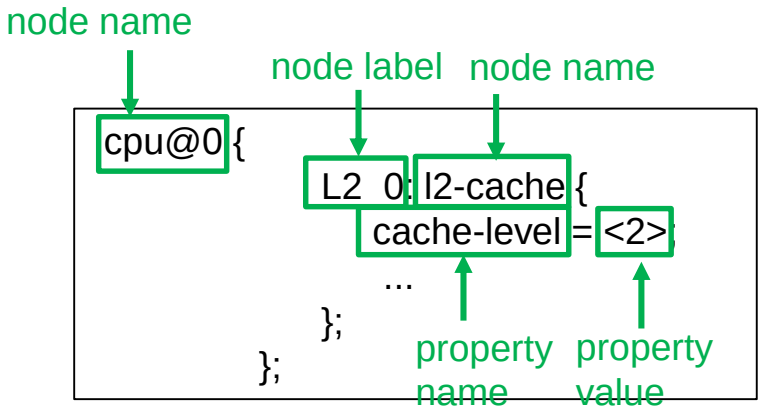
Example (an imaginary hardware)

```
/dts-v1/;

/ {
        aliases {
                sensor-controller = &i2c1;
        };

        soc {
                i2c1: i2c@40002000 {
                        compatible = "vnd,soc-i2c";
                        label = "I2C_1";
                        reg = <0x40002000 0x1000>;
                        status = "okay";
                        clock-frequency = < 100000 >;
                };
        };
};
```

DTS Name to C Name Conversion

- How to convert **Names in device tree source** to **Names in C**
  - lower case letters and **Underscore (_)**
  - non-alphanumeric(@,-) becomes underscore (_) to become valid C token
    - ✔ serial@40001000 → serial_40001000
    - ✔ current-speed → current_speed





What is Node Identifier?

- A *node identifier* is a way to refer to a devicetree node at C preprocessor time.
- Use node identifier to access devicetree data in C value form
- node identifiers are not values
  - There is no way to store node id in a variable
  - You cannot write:

```
/* These will give you compiler errors: */
void *i2c_0 = DT_INST(0, vnd_soc_i2c);
unsigned int i2c_1 = DT_INST(1, vnd_soc_i2c);
long my_i2c = DT_NODELABEL(i2c1);
```

  - If you want to something short to save typing, use C MACRO:

```
/* Use something like this instead: */

#define MY_I2C  DT_NODELABEL(i2c1)
#define INST(i)  DT_INST(i, vnd_soc_i2c)
#define I2C_0  INST(0)
#define I2C_1  INST(1)
```

How to get node identifier

- C Macros to get the devicetree **node identifier**
  - DT_ROOT : ***node id*** for / node
  - DT_PATH(path) : returns *node id* from node path components
    - ✔ path : with each name given as a separate argument
    - ✔ DT_PATH(soc, serial_40001000)
  - DT_NODELABEL(label): *node id* from node label
    - ✔ DT_NODELABEL(serial1)
  - DT_ALIAS(alias): *node id* from alias
  - DT_INST(inst)
  - DT_PARENT(node)
  - DT_CHILD(node)

Property Access

- Property of Node
- Checking Property and Values
  - DT_NODE_HAS_PROP(node_id, prop) to Check if a node has a property (1 if true or 0 otherwise)
    - ✔ node_id : node identifier
    - ✔ prop: lowercase-underscore property name

```
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), clock_frequency)  /* expands to 1 */
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), not_a_property)   /* expands to 0 */
```

Property values

Property type	How to write	Example
string	Double quoted	a-string = "hello, world!";
int	between angle brackets (< and >)	an-int = <1>;
boolean	for true, with no value (for false, use /delete-property/)	my-true-boolean;
array	between angle brackets (< and >), separated by spaces	foo = <0xdeadbeef 1234 0>;
uint8-array	in hexadecimal *without* leading 0x, between square brackets ([and]).	a-byte-array = [00 01 ab];
string-array	separated by commas	a-string-array = "string one", "string two", "string three";
phandle	between angle brackets (< and >)	a-phandle = <&mynode>;
phandles	between angle brackets (< and >), separated by spaces	some-phandles = <&mynode0 &mynode1 &mynode2>;
phandle-array	between angle brackets (< and >), separated by spaces	a-phandle-array = <&mynode0 1 2>, <&mynode1 3 4>;

Property Values

```
foo: device@0 { };
device@1 {
        sibling = <&foo 1 2>;
};
```

- Property values refer to other nodes in devicetree by their phandles, &foo where foo is a node label
- The sibling property of device@1 contains 3 cells, in order
  1. device@0 node's phandle: (&foo)
  2. value 1
  3. value 2

Simple Properties

- Property with string and Boolean works the same way
  - DT_PROP() expands to string literal in the case of string, and the number 0 or 1 in the case of Boolean

```
#define I2C1 DT_NODELABEL(i2c1)
DT_PROP(I2C1, status)  /* expands to the string literal "okay" */
```

- Property with type array, uint8-array, string-array
  - works similary except DT_PROP()

```
foo: foo@1234 {
        a = <1000 2000 3000>; /* array */
        b = [aa bb cc dd];    /* uint8-array */
        c = "bar", "baz";     /* string-array */
};
```

```
#define FOO DT_NODELABEL(foo)

int a[] = DT_PROP(FOO, a);          /* {1000, 2000, 3000} */
unsigned char b[] = DT_PROP(FOO, b); /* {0xaa, 0xbb, 0xcc, 0xdd} */
char* c[] = DT_PROP(FOO, c);        /* {"foo", "bar"} */
```

Reg Properties

- reg property is a sequence of (address, length) pairs, Each pair is called register block. Values are conventionally written in hex
  - For Devices accessed by memory-mapped I/O, address is usually the base address and length is the number of bytes occupied by the registers
- DT_NUM_REGS(node_id) is the total number of register blocks in the node's reg property
- You cannot read register block addresses and length with DT_PROP,
- instead if a node has only one register block, use:
  - DT_REG_ADDR(node_id): the given node's register block address
  - DT_REG_SIZE(node_id): its size
- instead if the node has multiple register blocks, use
  - DT_REG_ADDR_BY_IDX(node_id, idx): address if register block at index idx
  - DT_REG_SIZE_BY_IDX(node_id, idx): size of block at index idx
  - idx cannot be a variable

```
/* This will cause a compiler error. */
for (size_t i = 0; i < DT_NUM_REGS(node_id); i++)
        size_t addr = DT_REG_ADDR_BY_IDX(node_id, i);
```

Alias and Chosen nodes

```
/dts-v1/;

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

    aliases {
        my-uart = &uart0;
    };

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

- /aliases and /chosen nodes do not refer an actual hardware device
- my-uart is an alias for the node **/soc/serial@12340000**
- same node is set as value of chosen **zephyr,console** node
- The /chosen node's properties are used to configure system-wide settings
- a chosen node's label property is used to set the default value of Kconfig option
  - Zephyr-specific chosen properties
    - ✔ zephyr,bt-uart
    - ✔ zephyr,console

APIs in devicetree.h: macro(2)

- DT_PROP (node_id,prop): get devicetree property value
  - node_id: devicetree node identifier
  - prop : property name
  - Ex)
    - ✔ DT_PROP(DT_PATH(soc, serial_40001000), current_speed)
    - ✔ DT_PROP(DT_NODELABEL(serial1), current_speed)
  - types of return value (properties) defined by its binding
    - ✔ string: string literal
    - ✔ Boolean: 0 or 1
    - ✔ int
    - ✔ array, int8-array, string-array ({0,1,2} or {“hello”, “world”}
    - ✔ phandle: a node id for the node
  - some special properties have the **assumed type** even without binding
    - ✔ compatible=sting array
    - ✔ status=string // “okay”, “disable”,
    - ✔ interrupt-controller= boolean // 1, 0

API in devicetree.h: macro(4)

- DT_ALIAS(alias) : Get a node identifier for a node
  - alias: lowercase-underscore alias name
  - Ex) DT_PROP(DT_ALIAS(my_serial), current_speed) // 115200

```
/ {
        aliases {
                my-serial = &serial1;
        };

        soc {
                serial1: serial@40001000 {
                        status = "okay";
                        current-speed = <115200>;
                        ...
                };
        };
};
```

APIs in devicetree.h: macro(5)

- DT_INST(inst, compat): returns node identifier
  - inst : instance number which is zero-based indexes specific to a compatible
    - ✔ for each compatible, instance number starts with 0
    - ✔ exactly one instance number is assigned for each node with compatible
    - ✔ enabled nodes (status= “ok” or missing) are assigned instance numbers starting from 0
    - ✔ disabled nodes have instance numbers greater than those of any enabled node
  - compat : lowercase-underscore version of compatible property value without quote
  - Ex)

```
serial1: serial@40001000 {
        compatible = "vnd,soc-serial";
        status = "disabled";
        current-speed = <9600>;
        ...
};
serial2: serial@40002000 {
        compatible = "vnd,soc-serial";
        status = "okay";
        current-speed = <57600>;
        ...
};
serial3: serial@40003000 {
        compatible = "vnd,soc-serial";
        current-speed = <115200>;
        ...
};
```

```
DT_PROP(DT_INST(0, vnd_soc_serial), current_speed) // 57600

DT_PROP(DT_INST(1, vnd_soc_serial), current_speed) // 115200

DT_PROP(DT_INST(2, vnd_soc_serial), current_speed) // 9600
```

APIs in devicetree.h: macro(6)

- DT_PARENT(node_id) : Get a node id for a parent node
- DP_GPARENT(node_id): Get a node id for a grandparent node

```
gparent: grandparent-node {
        parent: parent-node {
                child: child-node { ... }
        };
};
```

```
// The following are equivalent
DT_NODELABEL(parent)
DT_PARENT(DT_NODELABEL(child))

// The following are equivalent
DT_GPARENT(DT_NODELABEL(child))
DT_PARENT(DT_PARENT(DT_NODELABEL(child))
```

- DT_CHILD(node_id, child): get a node id for a child (node name)

```
/ {
        soc-label: soc {
                serial1: serial@40001000 {
                        status = "okay";
                        current-speed = <115200>;
                };
        };
};
```

```
#define SOC_NODE DT_NODELABEL(soc_label)
DT_PROP(DT_CHILD(SOC_NODE, serial_40001000),
status) // "okay"
```

APIs in devicetree.h: macro(7)

- DT_NODE_PATH(node_id): get a node's full path as a string literal
- DT_NODE_FULL_NAME(node_id): get a devicetree node's name with unit-address as a string literal
  - node_id: node identifier

```
/ {
        soc {
                node: my-node@12345678 { ... };
        };
};
```

```
DT_NODE_PATH(DT_NODELABEL(node))
                                        // "/soc/my-node@12345678"
DT_NODE_PATH(DT_PATH(soc))       // "/soc"
DT_NODE_PATH(DT_ROOT)            // "/"
```

```
DT_NODE_FULL_NAME(DT_NODELABEL(node)) //
"my-node@12345678"
```

Configuration with Kconfig

- Configuration *at build time* to adapt Zeyphr kernel and subsystems for a specific application and platform is handled through **Kconfig** (same as Linux configuration).
- The goal is to support *configuration without having to change any source code*
- configuration options
  - defined in **Kconfig** files
  - specify dependencies between symbols
- Interactive Kconfig interfaces
  - *menuconfig* : cursor-based interface run in the terminal
  - *guiconfig* : graphical configuration interface
  - or manual edit zephyr/.config in application build directory

Visible and invisible Kconfig Symbols

- Visible Symbol is a Symbol defined with a prompt

```
config FPU
  bool "Support floating point operations"
  depends on HAS_FPU
```

```
[ ] Support floating point operations
```

- Invisible Symbol is a symbol without prompt
  - invisible symbols are not shown in the interactive configuration interfaces, and users have no direct control over their value. Instead get the value from defaults or from other symbols

```
config CPU_HAS_FPU
   bool
   help
     This symbol is y if the CPU has a hardware floating
point unit.
```

Setting Symbols in configuration

- Initial configuration is produced by merging *_defconfig for board with application settings from ususally *prj.conf*
- Assignments Syntax in configuration files
  - CONFIG_<symbol name>=<value>
  - There should be no spaces around equal sign
  - bool symbols can be enabled by setting to y or n
    - ✔ FPU symbols from the example above can be enabled CONFIG_FPU=y
  - other symbol types:
    - ✔ CONFIG_SIME_STRING=“cool value”
    - ✔ CONG_SME_INT=123
  - comments: #

Initial Configuration

- initial configuration settings from 3 sources
  - BOAD-specific configuration file in boards/<architecture>/<BOARD>/<BOARD>_defconfig
  - Any CMake cache entries prefix with CONFIG_
  - The application configuration
- Application configuration file in application configuration directory
  1. if CONF_FILE is set in CMake, the configuration file specified in it is merged and used as the application configuration.
  2. Otherwise, if prj_<BOARD>.conf is used if it exists
  3. Otherwise, if boards/<BOARD>.conf exists, the result of merging it with prj.conf is used
  4. Otherwise if board revisions are used and board/<BOARD>_<revision>.conf exists, the result of merging it with prj.conf and boards/<BOARD>.conf is used
  5. Otherwise, prj.conf is used from the application configuration directory