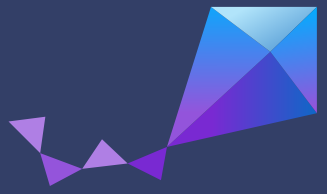


Zephyr® Project
Developer Summit 2022

Mastering Zephyr Driver Development

Gerard Marull-Paretas
gerard.marull@nordicsemi.no

Nordic Semiconductor ASA

7th June 2022

Outline

- Introduction
- Device model crash course
  - Devicetree
  - Zephyr devices
  - Example device driver
- Implementing a sensor driver
  - JM-101 fingerprint sensor
  - Mapping JM-101 to the sensor API
  - Coding session: JM-101 driver
  - Using the JM-101 driver
- Going custom
  - Motivation
  - An API to control locks
  - A servo driven lock
  - Coding session: servo-driven lock
  - Using the servo-driven lock driver
- Conclusions

Introduction

Introduction

- Zephyr can be **intimidating** for many embedded developers
- **Devicetree**, **Kconfig** are often **new** languages for embedded developers
- Sometimes **CMake** may be a **new** tool to learn as well
- A **frequent** Zephyr application developer **task** is to **implement drivers**
- To develop a driver, a minimum knowledge of the **Zephyr device model**, **Devicetree**, **Kconfig** and **CMake** are required

But...

- Zephyr documentation **lacks tutorials** or step by step guides
- **Not** all in-tree drivers follow **best or latest practices**
- Zephyr codebase is **large**, **not** trivial to **understand** what is relevant for my driver

Let's learn by programming a...smart lock!

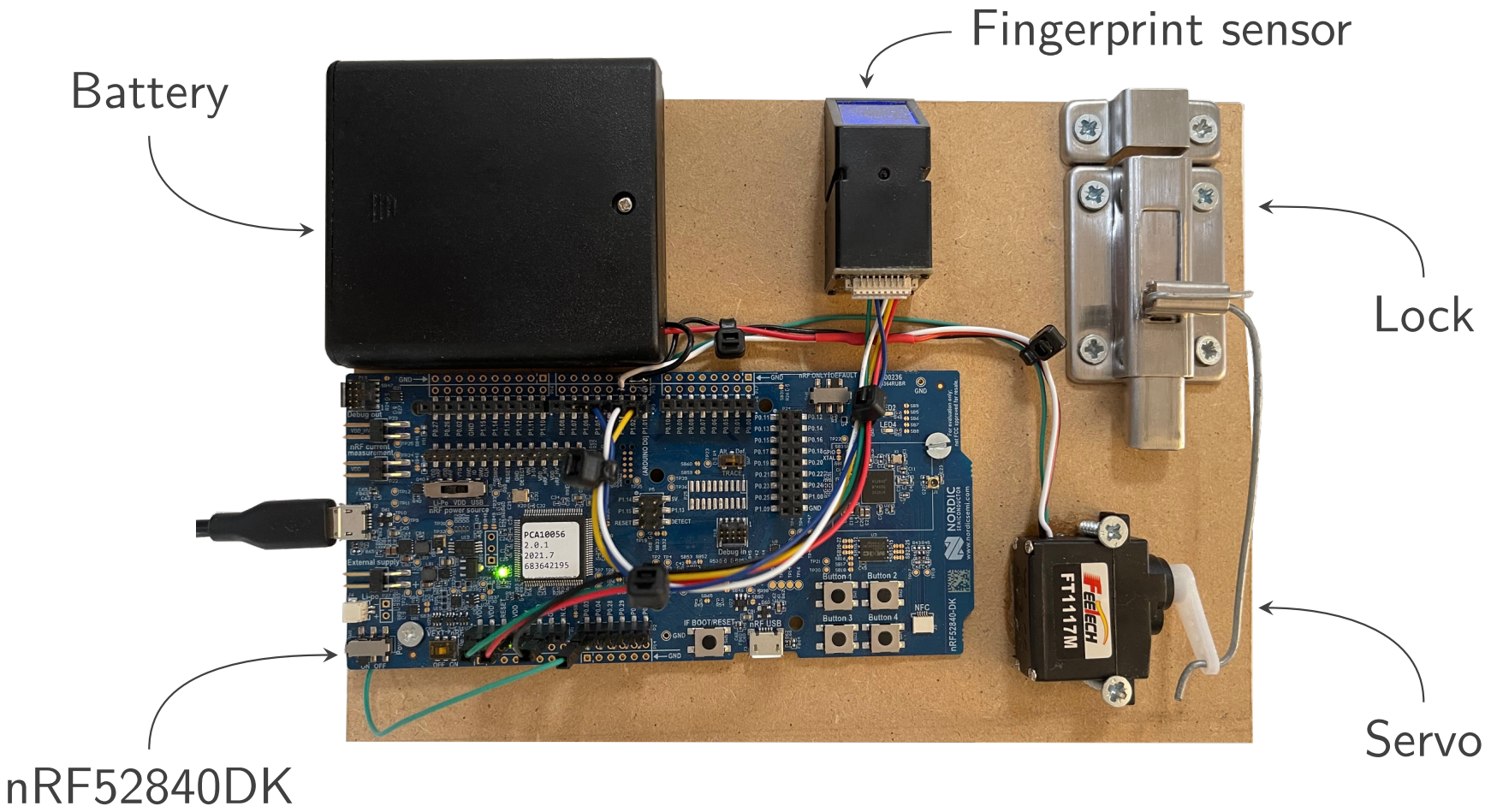


Figure: Smart lock prototype

Device model crash course

Devicetree

Devicetree

- Devicetree is a hierarchical data structure inherited from **Linux Kernel** used to **describe hardware**
- **Essential** to the Zephyr device model: most devices are described in Devicetree
- There are **two** types of Devicetree **input files**: the **sources** and the **bindings**, which describe the content
- Devicetree in Zephyr is **parsed** at **compile time** and converted to **C header** files
- All Devicetree **information** is accessible at **compile time** via the `<zephyr/devicetree.h>` API

Devicetree example

```
/dts-v1/;

/ {
  soc {
    i2c0: i2c@40003000 {
      compatible = "nordic,nrf-twim";
      label = "I2C0";
      reg = <0x40003000 0x1000>;

      apds9960@39 {
        compatible = "avago,apds9960";
        label = "APDS9960";
        reg = <0x39>;
      };

      ti_hdc@43 {
        compatible = "ti,hdc",
                     "ti,hdc1010";
        label = "HDC1010";
        reg = <0x43>;
      };
    };
  };
};
```

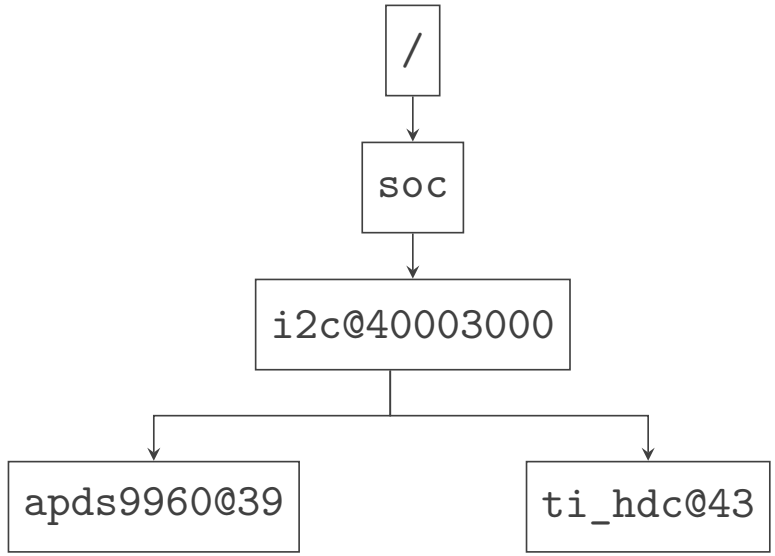


Figure: Devicetree example

Important Devicetree properties

Property	Description
`compatible`	Name of the **hardware** a node **represents**, typically `vendor,device`. Used to find the **bindings** for the node.
`label`	**Name** of the device (unique).
`reg`	Information used to **address** the device (optional). Value meaning depends on the device. In general, it is a **sequence of address-length pairs.**
`status`	**Status** of the device. `okay` (default if not specified) or `disabled`.

Table: Important Devicetree properties

Devicetree data types

Type	Example
`string`	`my-string = "hello,world"`
`int`	`my-int = <1>;`
`boolean`	`enable-my-feature;`
`array`	`my-ints = <0xdeadbeef 1234 0>;`
`uint8-array`	`my-bytes = [00 01 ab];`
`string-array`	`my-strings = "str1", "str2";`
`phandle`	`my-uart = <&uart0>;`
`phandles`	`my-uarts = <&uart0 &uart1>;`
`phandle-array`	`io-channels = <&adc 0>, <&adc 1>;`

Table: Devicetree data types

Devicetree instances

- Multiple **instances** of the same device can be defined in Devicetree, each with its **own** properties

```
/ {
  /* one instance of 'nordic,nrf-twim' */
  i2c0: i2c@40003000 {
    compatible = "nordic,nrf-twim";
    label = "I2C_0";
    reg = <0x40003000 0x1000>;
    clock-frequency = <I2C_BITRATE_STANDARD>;
  };

  /* another instance of 'nordic,nrf-twim' */
  i2c1: i2c@40004000 {
    compatible = "nordic,nrf-twim"
    label = "I2C_1";
    reg = <0x40004000 0x1000>;
    clock-frequency = <I2C_BITRATE_FAST>;
  };
};
```

Listing: `i2c0` and `i2c1` are two instances of `nordic,nrf-twim`

Devicetree composition

- The **final** Devicetree source is a result of **overlaying** multiple sources: SoC sources, board sources, application overlays with overrides, etc.

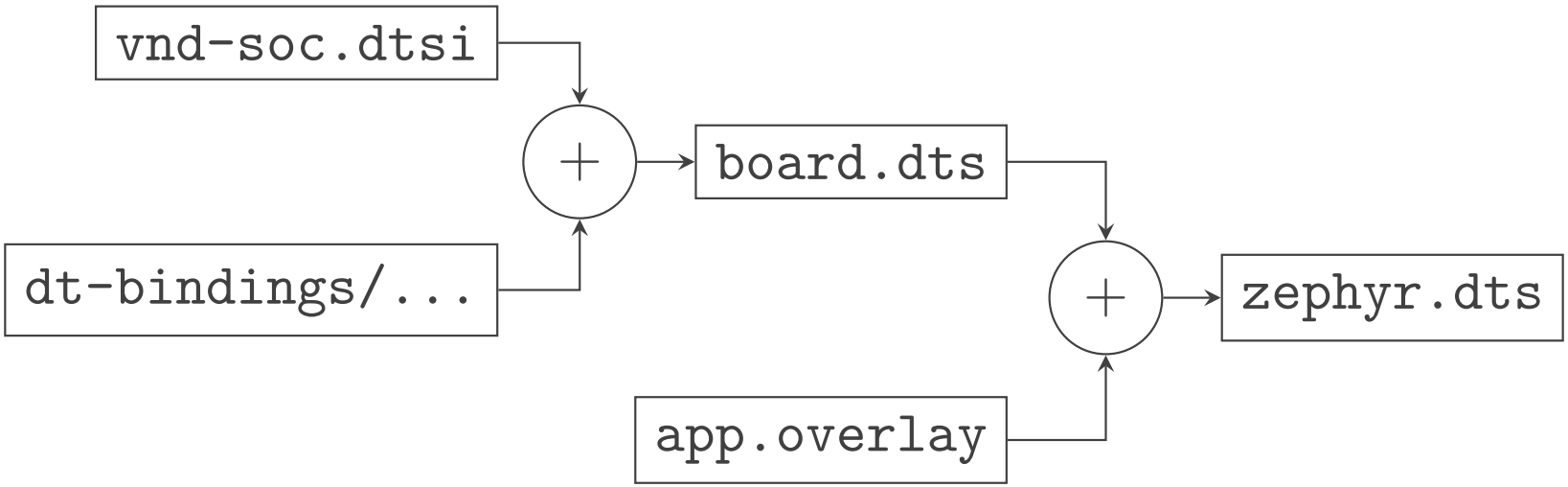


Figure: Devicetree composition example

Devicetree composition: example

```
/* vnd-soc.dtsi */
\ {
  soc {
    uart0: uart@abcd1234 {
      compatible = "vnd,uart";
      reg = <0xabcd1234 0x1000>;
      status = "disabled";
    };
  };
};
```

```
/* board.dts */
#include <vnd-soc.dtsi>

&uart0 {
  status = "okay";
  current-speed = <9600>;
}
```

```
/* zephyr.dts (final) */
\ {
  soc {
    uart0: uart@abcd1234 {
      compatible = "vnd,uart";
      reg = <0xabcd1234 0x1000>;
      status = "okay";
      current-speed = <9600>;
    };
  };
};
```

Listing: Devicetree composition example

Devicetree bindings

- **Content** of Devicetree, including data types, is **described** in **binding** files
- Binding files are used by the Devicetree parser to **validate content**
- Binding files are written in **YAML**, structure custom to Zephyr

```yaml
description: ...

compatible: vnd,dev

include: base.yaml

properties:
  foo:
    type: int
    required: true
```

```
/ {
  dev0: dev@deadbeef {
    compatible = "vnd,dev"
    label = "DEV0";
    reg = <0xdeadbeef>;
    foo = <7>;
  };
};
```

Listing: Binding file (left) and Devicetree source (right)

Devicetree bindings composition

- Devicetree bindings can **include** other base files, allowing to **re-use** common properties

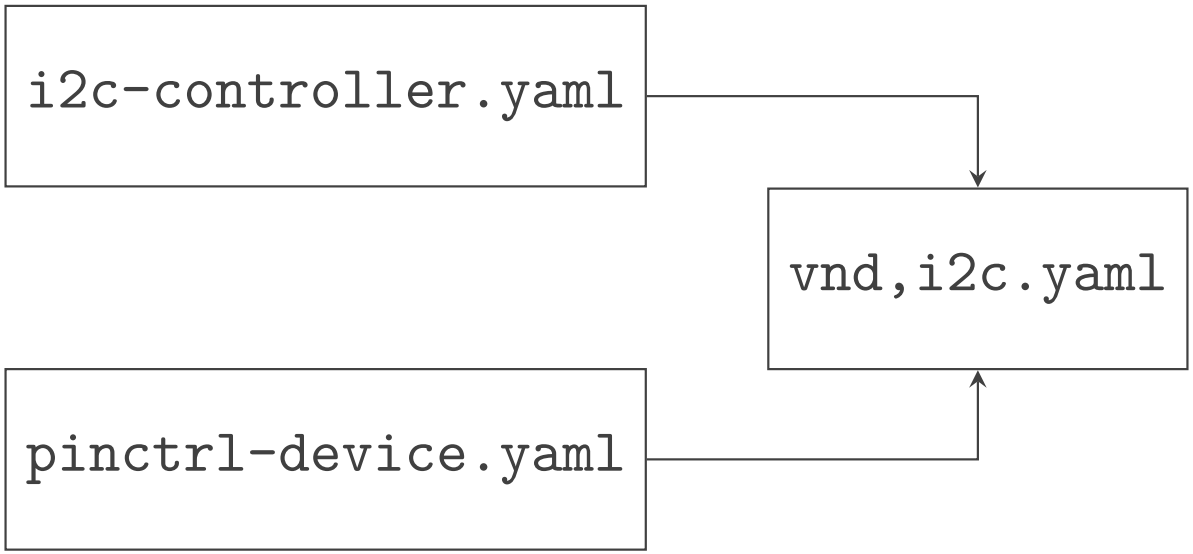


Figure: An I2C controller binding includes `i2c-controller.yaml` (e.g. for bus address/size, `clock-frequency`) and `pinctrl-device.yaml` (e.g. for `pinctrl-0`)

Devicetree output

- Devicetree is **converted** to a **C header** with a series of **C definitions** at compile time: **ZERO OVERHEAD!**
- Devicetree does not ***exist*** after code is compiled, only stored information will *survive*.

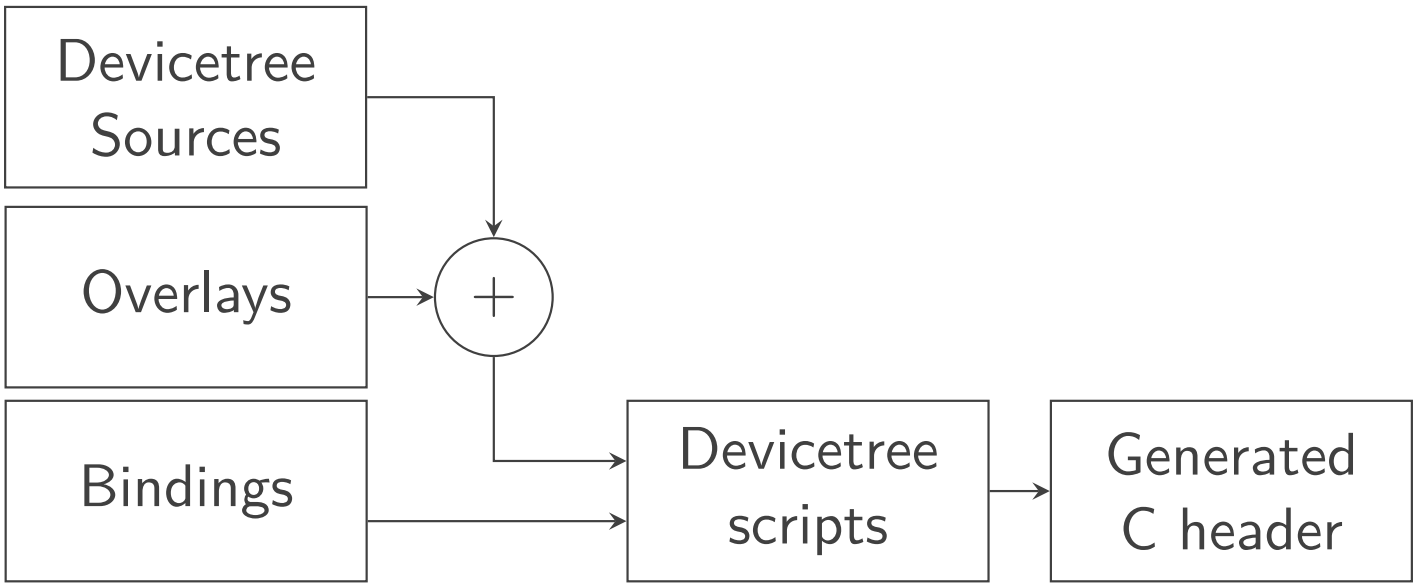


Figure: Generation of C header from Devicetree

Devicetree output: example

The `current-speed` property for the UART device

```
\ {
  soc {
    uart0: uart@abcd1234 {
      ...
      current-speed = <9600>;
    };
  };
};
```

Becomes. . .

```
#define DT_N_S_soc_S_uart_abdc1234_P_current_speed 9600
```

NOTE: Generated definitions must never be used directly in code!

Accessing Devicetree from C

- Devicetree information is accessed using the `<zephyr/devicetree.h>` **API**, **never** using generated definitions
- Devicetree API is **macro-based**, built upon C-preprocessor **token concatenation**
- Information about a particular node, can be obtained using a **node identifier**

How to obtain a node identifier

```
/dts-v1/;

/ {

  chosen {
    zephyr,console = &uart0;
  };

  aliases {
    my-uart = &uart0;
  };

  soc {
    uart0: uart@abcd1234 {
      compatible = "vnd,soc-uart";
      reg = <0xabcd1234 0x1000>;
      current-speed = <9600>;
    };
  };
};
```

```
/* by full path */
DT_PATH(soc, uart_abcd1234)

/* by node label */
DT_NODELABEL(uart0)

/* by using an alias
 * useful for generic samples
 */
DT_ALIAS(my_uart)

/* by using a choice
 * typically used by subsystems
 */
DT_CHOSEN(zephyr_console)

/* by specifying an instance number
 * of a given compatible, e.g. 0
 */
DT_INST(0, vnd_soc_uart)
```

Listing: Examples on how to obtain the node identifier for the `uart@abcd1234` node

Accessing Devicetree from C: examples

```
/dts-v1/;

/ {
    i2c0: i2c@abcd1234 {
      ...
      sensor0: sensor@ff {
        compatible = "vnd,mysensor";
        reg = <0xff>;
        int-gpios = <&gpio0 1
                     GPIO_ACTIVE_HIGH>;
        sample-freq = <1000>;
      };
    };
  };
};
```

```
/* use node label to obtain sensor node
 * identifier
 */
#define SENSOR_0 DT_NODELABEL(sensor0)

/* expands to 0xff */
DT_REG_ADDR(SENSOR_0)

/* expands to gpio0 node identifier */
DT_PHANDLE(SENSOR_0, int_gpios)
/* equivalent (GPIO specific helpers) */
DT_GPIO_CTLR(SENSOR_0, int_gpios)

/* expands to 1 (pin cell) */
DT_PHA(SENSOR_0, int_gpios, pin)
/* equivalent (GPIO specific helpers) */
DT_GPIO_PIN(SENSOR_0, int_gpios)

/* expands to 1000 */
DT_PROP(SENSOR_0, sample_freq)
```

Listing: Examples on how to access Devicetree information

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

Instance-based Devicetree API

- It is often necessary to write code **generic to any device instance**, specially in a **driver context**
- Most Devicetree APIs have their **instance-based version**, e.g. for `DT_PROP` we have `DT_INST_PROP`
- Instance-based macros **rely** on `DT_DRV_COMPAT` being **defined** to the compatible implemented by the driver

```
/* we're working with vnd,mysensor compatible */
#define DT_DRV_COMPAT vnd_mysensor

/* obtain 'sample-freq' property value for instance 0
 * of vnd,mysensor
 */
uint32_t sample_freq = DT_INST_PROP(0, sample_freq)
```

Listing: Example of instance-based Devicetree API

Zephyr devices

Zephyr devices

- Zephyr devices are represented by `struct device`, defined in `<zephyr/device.h>`
- `struct device` just holds a series of **references** to resources **defined by drivers** or **defined internally** by the device definition macros
- Devices are **defined** in **ROM** by a series of **macros**:
  - `DEVICE_DEFINE`: Non-Devicetree devices
  - `DEVICE_DT_DEFINE`/`DEVICE_DT_INST_DEFINE`: Devicetree-based devices
- Devices are **automatically initialized** at boot time, before `main` is reached
- Devices are **initialized** in a specific **order**, determined by a **level** and a manually defined **priorities**

Device structure

```
struct device {
  const char *name;
  const void *config;
  void * const data;
  const void *api;
  ...
};
```

Listing: Core `struct device` members

Field	Purpose
`name`	Device name (unique), corresponds to the `label` property for Devicetree devices
`config`	Reference to **read-only configuration** set at **compile time**, typically used to store Devicetree properties
`data`	Reference to device **data** that needs to be modified at **runtime**, e.g. counter, state, etc.
`api`	Reference to the device **API operations**

Table: Core `struct device` members description

Device definition

```
/* define a device given an instance number */
DEVICE_DT_INST_DEFINE(
  inst,                 /* Devicetree instance number */
  init_fn,              /* Initialization function */
  pm_device,            /* Power management resources (optional) */
  data_ptr,             /* Reference to instance data */
  config_ptr,           /* Reference to instance configuration */
  level,                /* Initialization level (e.g. POST_KERNEL) */
  prio,                 /* Initialization priority (0-99) */
  api_ptr,              /* Reference to API operations */
);
```

Listing: Definition of a Devicetree device[1]

[1] `DEVICE_DT_DEFINE` is also available if using a DT node identifier instead of an instance number

Example device driver

Device driver example: bindings

```yaml
description: ...

compatible: vnd,mysensor

include: base.yaml

properties:
  reg:
    required: true

  int-gpios:
    type: phandle-array
    required: true

  sample-freq:
    type: int
    required: true
```

Listing: Devicetree bindings for `vnd,mysensor`, tipically in `dts/bindings/...`

Device driver example: source (I)

```
/* tells DT INST based macros which compatible to use
 * tipically defined at the top, but can be defined
 * anywhere before using DT INST based macros
 */
#define DT_DRV_COMPAT vnd_mysensor
```

Listing: Definition of driver compatible

Device driver example: source (II)

```c
/* driver configuration (immutable)
 * tipically information retrieved from Devicetree
 */
struct mysensor_config {
  uint8_t reg;
  struct gpio_dt_spec int_gpio;
  uint32_t sample_freq;
}

/* driver data (mutable) */
struct mysensor_data {
  uint8_t buf[32];
  struct k_msgq queue;
};
```

Listing: Definition of driver configuration and data structures

Device driver example: source (III)

```
static int mysensor_sample_fetch(const struct device *dev, ...)
{
  /* implementation of API sample_fetch operation */
}
...

static const struct sensor_driver_api mysensor_api = {
  .sample_fetch = mysensor_sample_fetch,
  ...
};
```

Listing: Definition driver API operations (sensor API here)

```
static int mysensor_init(const struct device *dev)
{
  /* initialize driver (e.g. check for dependent devices readiness) */
}
```

Listing: Definition of driver initialization function

Device driver example: source (IV)

```
/* macro to define an instance of our device
 * this code applies to any instance, so we're using DT_INST_* macros
 */
#define MYSENSOR_DEFINE(i)                                                  \
  static const struct config##i = {                                         \
    .reg = DT_REG_INST_ADDR(i),                                             \
    .int_gpio = GPIO_DT_SPEC_INST_GET(i, int_gpios),                        \
    .sample_freq = DT_INST_PROP(i, sample_freq),                            \
  };                                                                        \
                                                                            \
  static struct mysensor_data data##i;                                      \
                                                                            \
  DEVICE_DT_INST_DEFINE(i, mysensor_init, NULL, &data##i, &config##i        \
                        POST_KERNEL, CONFIG_SENSOR_PRIORITY,                \
                        &mysensor_api);

/* expands MYSENSOR_DEFINE for each device in DT with compatible
 * vnd,mysensor (defined by DT_DRV_COMPAT) and status "okay"
 */
DT_INST_FOREACH_STATUS_OKAY(MYSENSOR_DEFINE)
```

Listing: Device driver definition, multi-instance capable

Device driver example: Devicetree instantiation

```
/dts-v1/;

/ {
    i2c0: i2c@abcd1234 {
      ...
      accel: sensor@ff {
        compatible = "vnd,mysensor";
        label = "ACCEL"
        reg = <0xff>;
        int-gpios = <&gpio0 1 GPIO_ACTIVE_HIGH>;
        sample-freq = <1000>;
      };
    };
  };
};
```

Listing: Instantiation of `vnd,mysensor` in Devicetree

Obtaining references to devices

- **References** to **Devicetree** devices can be obtained at **compile-time** using `DEVICE_DT_GET` family of macros
- **References** can also be obtained at **runtime** by name, using `device_get_binding` (rarely needed nowadays)

```
/* get reference to a device using a node identifier */
const struct device *dev = DEVICE_DT_GET(DT_NODELABEL(accel));

/* get reference to a device using a node identifier, NULL if none */
const struct device *dev = DEVICE_DT_GET_OR_NULL(DT_NODELABEL(accel));

/* obtain one reference of the vnd,my-sensor compatible, NULL if none */
const struct device *dev = DEVICE_DT_GET_ANY(vnd_my_sensor);

/* obtain one reference of the vnd,my-sensor compatible */
const struct device *dev = DEVICE_DT_GET_ONE(vnd_my_sensor);

/* get reference to a device at runtime, using name lookup */
const struct device *dev = device_get_binding("ACCEL");
```

Listing: Examples on how to obtain device references

Implementing a sensor driver

JM-101 fingerprint sensor

JM-101 fingerprint sensor

- JM-101 is a **fingerprint** sensor manufactured by Hangzhou Zhean Technology Co. Ltd.
- Based on **Synodata GC0328** firmware, same as many clones
- Not strictly a sensor, but a **biometric** device
- Allows to **save** and **verify** fingerprints
- **UART** based communications, includes a **touch output**

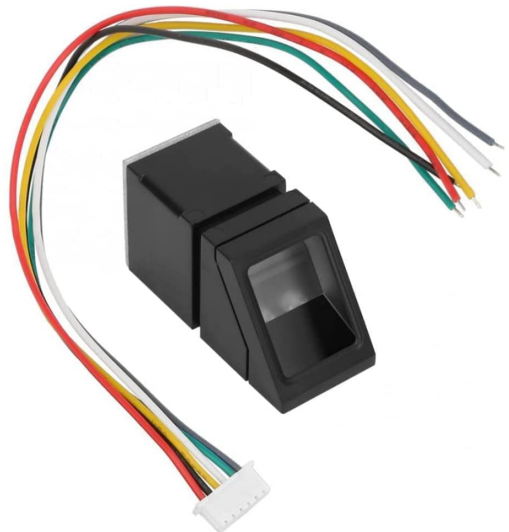


Figure: JM-101 fingerprint sensor

JM-101 fingerprint sensor: wiring example

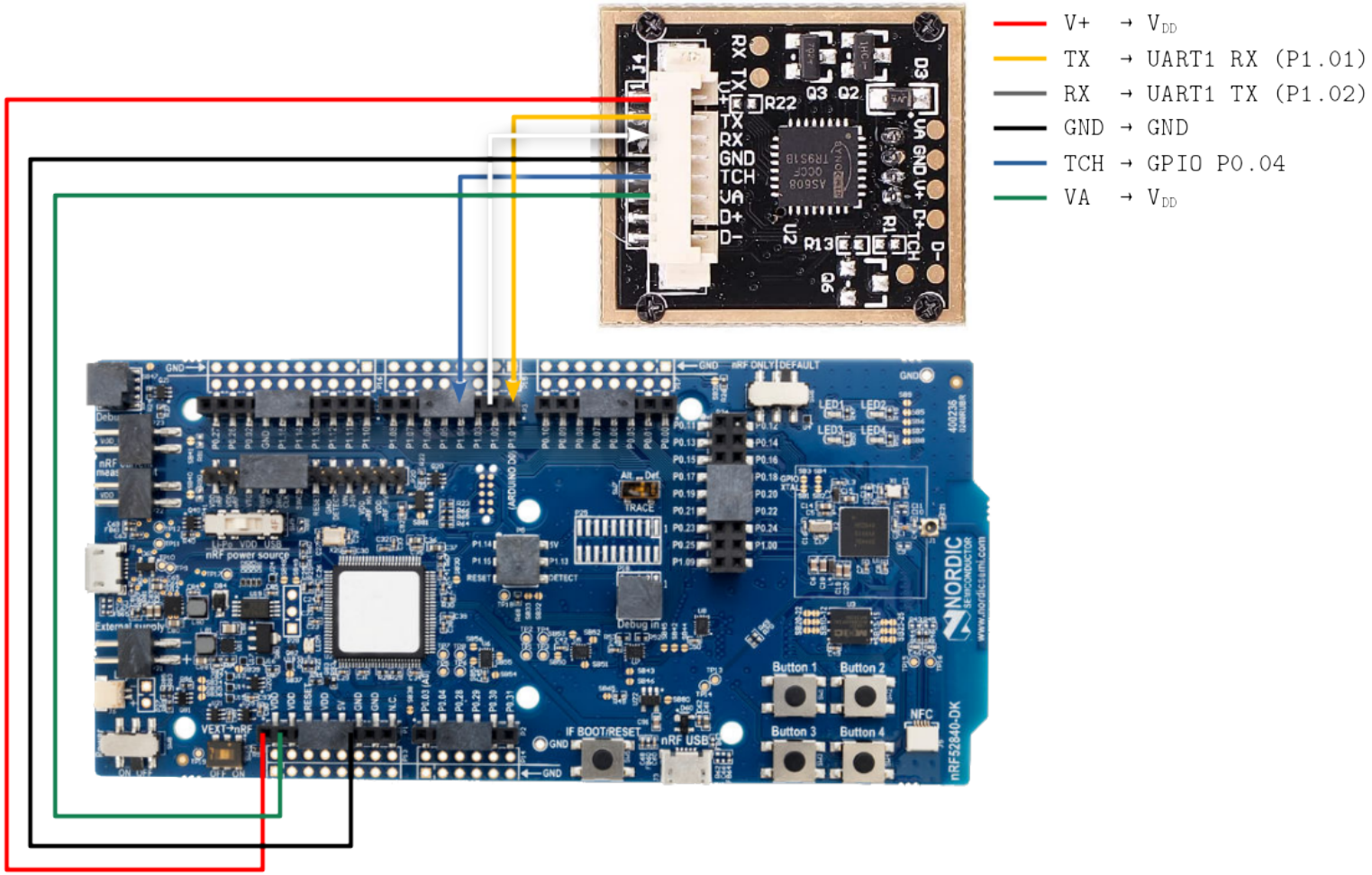


Figure: Wiring diagram for JM-101 fingerprint sensor and nRF52840DK

JM-101 fingerprint sensor: communications (I)

- All data is transmitted **big-endian**
- Communications are **synchronous**
- Communication **parameters** are:

Parameter	Value
Baudrate	57600
Data bits	8
Parity	None
Stop bits	2

Table: Serial communications parameters

JM-101 fingerprint sensor: communications (II)

- Actions are triggered by MCU sending command packets
- Packet is structured as follows:

Header	Address	Identifier	Length	Command	Data	Checksum
2b	4b	1b	2b	1b	Nb	2b
EF01H	`ADDR`	01H	N + 3	`CMD`	`DATA`	$\sum_{i=6}^{10+N} \mathrm{p}[i]$

- Where:
  - `ADDR` is the address value, tipically FFFFFFFFH
  - `CMD` is the command, e.g. 31H for `PS_AutoEnroll`
  - `DATA` are the N bytes of payload

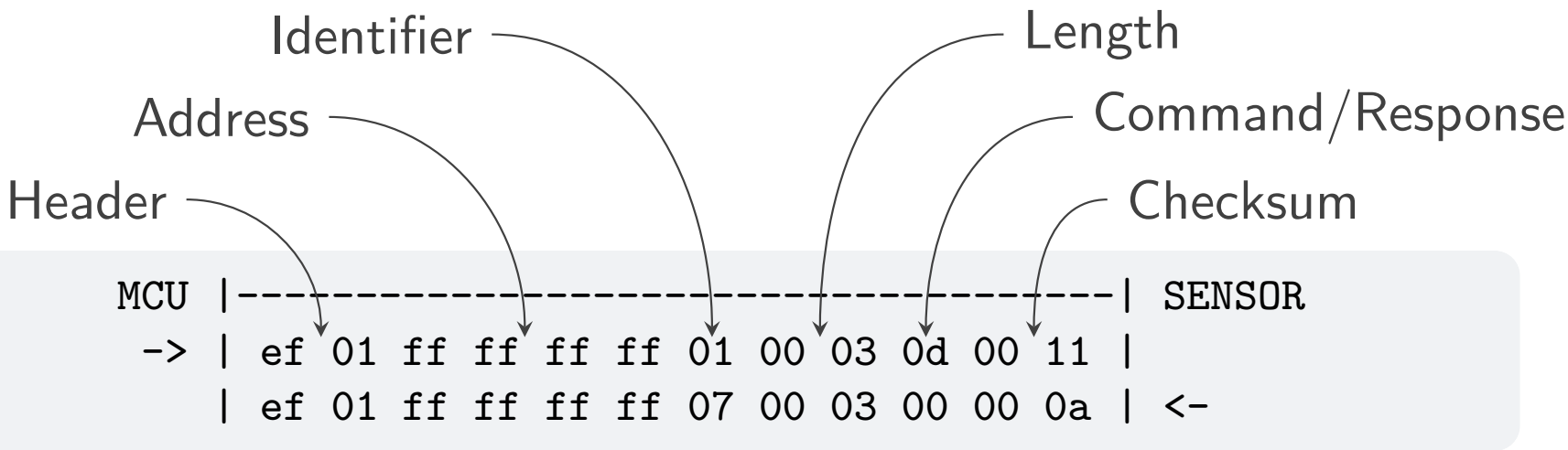
JM-101 fingerprint sensor: communications (III)

- Sensor responds to commands with a response packet/s
- Packet is structured as follows:

Header	Address	Identifier	Length	Result	Data	Checksum
2b	4b	1b	2b	1b	Nb	2b
EF01H	`ADDR`	07H	N + 3	`RES`	`DATA`	$\sum_{i=6}^{10+N} \mathrm{p}[i]$

- Where:
  - `ADDR` is the address value, tipically FFFFFFFFH
  - `RES` is the command result, e.g. 00H = success
  - `DATA` are the N bytes of payload

JM-101 fingerprint sensor: communications (IV)



Listing: Communications sequence for `PS_Empty` (`0DH`) command

JM-101 operation details: verify (I)

- Fingerprint verification sequence is triggered by `PS_AutoVerify` command (32H)
- **Input** parameters are:

Security Level	ID number	Configuration
1b	2b	2b

- ID number can be set to FFFFH for a **1:N search**, or to a specific fingerprint ID for a **1:1 search**
- Sensor sends **up to 3 responses**: acknowledge, fingerprint collection and verification results
- **Response** parameters are:

Verification Step	ID number	Score
1b	2b	2b

JM-101 operation details: verify (II)

1:N Search

PS_AutoVerify

```
MCU |-----------------------------------------------| SENSOR
 -> | ef 01 ff ff ff ff 01 00 08 32 00 ff ff 00 00 02 39 |
    | ef 01 ff ff ff ff 07 00 08 00 00 ff ff 00 00 02 0d | <-
    | ef 01 ff ff ff ff 07 00 08 00 01 ff ff 00 00 02 0e | <-
    | ef 01 ff ff ff ff 07 00 08 00 05 00 01 00 44 00 59 | <-
```

Listing: PS_AutoVerify example sequence

Score: 68

Collection Success

Matches with #1

Collection Step

Verification Step

Verification Success

JM-101 operation details: enroll (I)

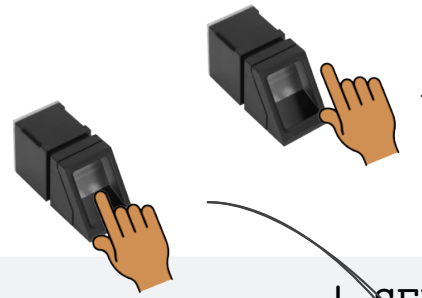
- Enrollment sequence is triggered by `PS_AutoEnroll` command (31H)
- **Input** parameters are:

ID number	# Readings	Configuration
2b	1b	2b

- Chosen ID must be **empty** in the database
- Sensor sends **a minimum of 7 responses**: acknowledge, acquisition, generation, finger away, merge, register and result
- **Response** parameters are:

Enrollment Step	Acquisition count/details
1b	1b

JM-101 operation details: enroll (II)



2 readings

Store to #1

PS_AutoEnroll

```
MCU |-----------------------------------------------| SENSOR
 -> | ef 01 ff ff ff ff 01 00 08 31 00 01 02 00 00 00 3d |
    | ef 01 ff ff ff ff 07 00 05 00 00 00 00 0c          | <-
    | ef 01 ff ff ff ff 07 00 05 00 01 01 00 0e          | <-
    | ef 01 ff ff ff ff 07 00 05 00 02 01 00 0f          | <-
    | ef 01 ff ff ff ff 07 00 05 00 03 01 00 10          | <-
    | ef 01 ff ff ff ff 07 00 05 00 01 02 00 0f          | <-
    | ef 01 ff ff ff ff 07 00 05 00 02 02 00 10          | <-
    | ef 01 ff ff ff ff 07 00 05 00 04 f0 01 00          | <-
    | ef 01 ff ff ff ff 07 00 05 00 05 f1 01 02          | <-
    | ef 01 ff ff ff ff 07 00 05 00 06 f2 01 04          | <-
```

Listing: `PS_AutoEnroll` example sequence

Store Success

Store Step

JM-101 operation details: touch

- When fingerprint sensor is **touched**, `TCH` output goes **high**
- Can be used to **trigger** a fingerprint verification sequence
- Touch sensor has a **standalone circuitry**, powered by `VA` pin

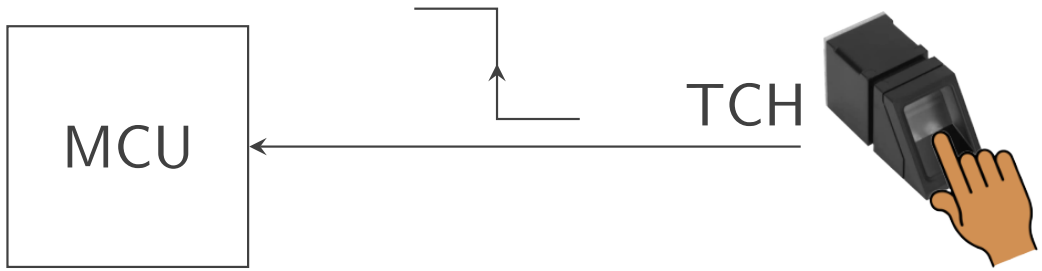


Figure: `TCH` output goes high when the sensor is touched

Mapping JM-101 to the sensor API

The sensor driver API

- Sensor driver API is defined in `<zephyr/drivers/sensor.h>`
- Provides interfaces to:
  - Fetch and get sensor **data**
  - Set and get sensor **attributes**
  - Configure sensor **triggers**
- All operations are performed on a **channel** basis
- **Not a perfect** fit for JM-101, but *it works* if we abuse a little bit of the API

Mapping JM-101 actions to the sensor API

API call	JM-101 action
`sensor_sample_fetch`	▪ Read and verify a fingerprint
`sensor_channel_get`	▪ Obtain fingerprint verification results
`sensor_attr_set`	▪ Configure verification (1:1, 1:N...) ▪ Empty fingerprint database ▪ Trigger enrollment sequence
`sensor_trigger_set`	▪ Configure touch input interrupt

Table: JM-101 actions mapped to the sensor API

Coding session:
JM-101 driver

Coding session

Please watch the session

The following slides only provide some highlights

File tree layout

```
$ROOT: $ZEPHYR_BASE or application module root
|-- drivers
|   |-- CMakeLists.txt
|   |-- Kconfig
|   `-- sensor
|       |-- CMakeLists.txt
|       |-- Kconfig
|       `-- jm101                       # Sensor driver folder
|           |-- CMakeLists.txt          # Driver CMake file
|           |-- Kconfig                 # Driver Kconfig file
|           `-- jm101.c                 # Driver source file
|-- dts
|   `-- bindings
|       `-- sensor
|           `-- zeantec,jm101.yaml      # Devicetree bindings
`-- include
    `-- {app|zephyr}
        `-- drivers
            `-- sensor
                `-- jm101.h             # Sensor custom channels,
                                        # attributes, etc.
```

Listing: File tree with all relevant JM-101 sensor driver files

Devicetree bindings

```yaml
description: ...
compatible: "zeantec,jm101"

include: base.yaml

properties:
  uart:
    type: phandle
    required: true
    description: |
      UART port to which the sensor is connected to.

  address:
    type: int
    required: false
    default: 0xffffffff
    description: |
      The address of the sensor. Defaults to 0xFFFFFFFF, which is the factory
      default.

  touch-gpios:
    type: phandle-array
    required: false
    description: |
      Touch input GPIO (active high). When the sensor is touched, the GPIO is
      set to high. It is used to trigger a fingerprint read.
```

Listing: `$ROOT/dts/bindings/sensor/zeantec,jm101.yaml`

Kconfig (I)

```
menu "Drivers"
...
rsource "sensor/Kconfig"
...
endmenu
```

Listing: $ROOT/drivers/Kconfig

```
if SENSOR
...
rsource "jm101/Kconfig"
...
endif # SENSOR
```

Listing: $ROOT/drivers/sensor/Kconfig

Kconfig (II)

Software Dependencies

Enabled by default if `okay` in DT

```
DT_COMPAT_ZEANTEC_JM101 := zeantec,jm101

config JM101
  bool "JM-101 fingerprint sensor"
  depends on GPIO && SERIAL && UART_INTERRUPT_DRIVEN
  default $(dt_compat_enabled,$(DT_COMPAT_ZEANTEC_JM101))
  help
    JM-101 fingerprint sensor.

config JM101_TRIGGER
  bool "JM-101 touch input support"
  depends on JM101
  help
    When enabled, the JM101 can be provided with a touch input GPIO to
    trigger fingerprint readings.
```

Listing: `$ROOT/drivers/sensor/jm101/Kconfig`

CMake

```
# include JM101 driver folder if enabled
add_subdirectory_ifdef(CONFIG_JM101 jm101)
```

Listing: $ROOT/drivers/sensor/CMakeLists.txt

```
# create a new Zephyr library, add sensor sources to it
zephyr_library()
zephyr_library_sources(jm101.c)
```

Listing: $ROOT/drivers/sensor/jm101/CMakeLists.txt [2]

[2] Some driver classes have a single library, tipically drivers where a single implementation is chosen. In such cases, `zephyr_library_amend` should be used instead.

Driver highlights (I)

- **Standard** channels/attributes are **not sufficient** for JM-101
- **Private/custom** channels are defined in a **public** header

```c
#include <zephyr/drivers/sensor.h>

/** @brief Any identifier (used to do 1:N identification). */
#define JM101_ID_ANY 0xFFFFU

/** @brief JM101 custom channels. */
enum jm101_channel {
  /** Fingerprint verification. */
  JM101_CHAN_FINGERPRINT = SENSOR_CHAN_PRIV_START,
};

/** @brief JM101 custom attributes. */
enum jm101_attribute {
  /** Fingerprint ID used when verifying. */
  JM101_ATTR_ID_NUM = SENSOR_ATTR_PRIV_START,
  /** Run the enrolling sequence. */
  JM101_ATTR_ENROLL,
  /** Emtpies the fingerprint database. */
  JM101_ATTR_EMPTYDB,
};
```

Listing: `$ROOT/include/app|zephyr/sensor/jm101.h`

Driver highlights (II)

```
/* We're implementing a driver for the zeantec,jm101 compatible */
#define DT_DRV_COMPAT zeantec_jm101
```

Listing: Definition of the driver's compatible

Driver highlights (III)

```c
/* We need to define a device: DEVICE_DT_INST_DEFINE */
#include <zephyr/device.h>
/* We need to retrieve information from Devicetree */
#include <zephyr/devicetree.h>
/* We need to control the TOUCH input GPIO */
#include <zephyr/drivers/gpio.h>
/* We need to define the sensor API ops */
#include <zephyr/drivers/sensor.h>
/* We communicate with the sensor via UART */
#include <zephyr/drivers/uart.h>
/* Adding some logging is always useful */
#include <zephyr/logging/log.h>
/* We'll be manipulating big-endian data */
#include <zephyr/sys/byteorder.h>
/* We'll need utilities like IF_ENABLED */
#include <zephyr/sys/util_macro.h>

/* We have custom channels/attributes */
#include <app/drivers/sensor/jm101.h>
```

Listing: All required includes for the JM-101 driver

Driver highlights (IV)

- Driver is made **multi-instance** by combining both `DT_INST_FOREACH_STATUS_OKAY` and `JM101_DEFINE`

```
#define JM101_DEFINE(i)                                                    \
  static struct jm101_data jm101_data_##i = {                              \
    .verify_id = JM101_ID_ANY,                                             \
  };                                                                       \
                                                                           \
  /* pull instance configuration from Devicetree */                        \
  static const struct jm101_config jm101_config_##i = {                   \
    .uart = DEVICE_DT_GET(DT_INST_PHANDLE(i, uart)),                       \
    .addr = DT_INST_PROP(i, address),                                      \
    IF_ENABLED(CONFIG_JM101_TRIGGER,                                       \
               (.touch = GPIO_DT_SPEC_INST_GET_OR(i, touch_gpios, {}), ))\
  };                                                                       \
                                                                           \
  /* define a new device inst. with the data and config defined above */   \
  DEVICE_DT_INST_DEFINE(i, jm101_init, NULL, &jm101_data_##i,              \
                        &jm101_config_##i, POST_KERNEL,                    \
                        CONFIG_SENSOR_INIT_PRIORITY, &jm101_api);

/* expand JM101_DEFINE for all instances defined in Devicetree with
 * status "okay"
 */
DT_INST_FOREACH_STATUS_OKAY(JM101_DEFINE)
```

Listing: JM-101 instantiation

Driver highlights (V)

- All other devices used by the driver must be **checked for readiness** when driver is initialized

```
static int jm101_init(const struct device *dev)
{
  const struct jm101_config *config = dev->config;
  ...

  if (!device_is_ready(config->uart)) {
    LOG_ERR("UART not ready");
    return -ENODEV;
  }

  ...
#ifdef CONFIG_JM101_TRIGGER
  if (config->touch.port != NULL) {
    if (!device_is_ready(config->touch.port)) {
      LOG_ERR("Touch GPIO controller not ready");
      return -ENODEV;
    }
    ...
#endif
  ...
}
```

Listing: JM-101 instantiation

Driver highlights (VI)

- A message queue (`struct k_msgq`) is used to store received packets
- Allows CPU to **idle** while waiting for sensor to respond

```
static void uart_cb_rx_handler(const struct device *dev)
{
  ...
  if (data->buf_ctr >= PKT_LEN(data->rx_data_len)) {
    k_msgq_put(&data->rx_queue, data->buf, K_NO_WAIT);
    ...
  }
}

static int jm101_recv(const struct device *dev, uint8_t *result,
                      uint8_t *rx_data, uint8_t rx_data_len)
{
  ...
  ret = k_msgq_get(&data->rx_queue, buf, RX_TIMEOUT);
  ...
}
```

Listing: Usage of message queues to receive packets

Using the JM-101 Driver

Devicetree device definition

```
/ {
    fpreader: jm101 {
        compatible = "zeantec,jm101";
        label = "JM101";
        uart = <&uart1>;
        touch-gpios = <&gpio1 4 GPIO_ACTIVE_HIGH>;
    };
};
```

Listing: Devicetree definition of one JM-101 device instance

Project configuration

- JM-101 sensor driver is enabled when `CONFIG_JM101=y`
- `CONFIG_JM101` is automatically enabled if:
  - All its **dependencies** are **enabled**, including `CONFIG_SENSOR`
  - One or more instances are `okay` in Devicetree

```
# dependencies required so that JM-101 driver can be enabled
CONFIG_GPIO=y
CONFIG_SERIAL=y
CONFIG_UART_INTERRUPT_DRIVEN=y

# enable the sensor driver class
CONFIG_SENSOR=y

# Optional, only if we need trigger support
CONFIG_JM101_TRIGGER=y
```

Listing: Project configuration (`prj.conf`)

Obtain a reference to a device instance

```c
/* reference resolved at compile time (using unique node label) */
const struct device *fpreader = DEVICE_DT_GET(DT_NODELABEL(fpreader));

/* verify for device readiness before operation */
if (!device_is_ready(fpreader)) {
    printk("Fingerprint sensor device not ready\n");
    return -ENODEV;
}
```

Listing: Obtain a reference to the `fpreader` device and verify if ready

Example: verify a fingerprint

```c
/* configure verification (optional) */
struct sensor_value val = {
  .val1 = JM101_ID_ANY, /* 1:N verification */
};

sensor_attr_set(fpreader, JM101_CHAN_FINGERPRINT,
                JM101_ATTR_ID_NUM, &val);

/* start fingerprint verification */
ret = sensor_sample_fetch_chan(dev, JM101_CHAN_FINGERPRINT);
if (ret < 0) {
  printk("Verification failed\n");
  return ret;
}

/* obtain verification results */
sensor_channel_get(dev, JM101_CHAN_FINGERPRINT, &val);

printk("Verified fingerprint #%d, score: %d\n", val.val1, val.val2);
```

Listing: Example code to verify a fingerprint

Example: register a fingerprint

```c
/* empty database first (optional) */
ret = sensor_attr_set(fpreader, JM101_CHAN_FINGERPRINT,
                      JM101_ATTR_EMPTYDB, NULL);
if (ret < 0) {
  printk("Failed to empty database\n");
  return ret;
}

/* enroll */
struct sensor_value val = {
  .val1 = 1U, /* store as fingerprint ID #1 */
  .val2 = 2U, /* number of readings */
};

ret = sensor_attr_set(fpreader, JM101_CHAN_FINGERPRINT,
                      JM101_ATTR_ENROLL, &val);
if (ret < 0) {
  printk("Failed to enroll\n");
  return ret;
}

printk("Enrollment finished\n");
```

Listing: Example code to register a fingerprint

Example: configure trigger

```c
void fp_trig_handler(const struct device *dev,
                     const struct sensor_trigger *trigger)
{
  /* perform here a fingerprint verification */
}

const struct sensor_trigger trig = {
  .type = SENSOR_TRIG_TAP,
  .chan = JM101_CHAN_FINGERPRINT,
};

/* configure fingerprint trigger */
ret = sensor_trigger_set(fpreader, &trig, fp_trig_handler);
if (ret < 0) {
  printk("Failed to configure trigger\n");
  return ret;
}
```

Listing: Example code to configure touch trigger

Going custom

Motivation

Why creating a custom API?

- **Upstream** APIs tend to be designed for **common** peripherals, e.g. I2C, SPI, GPIO...
- MCUs often have highly **specialized** peripherals, **difficult** to integrate into a vendor **generic** API
- Such peripherals are frequently used in **specific domains**, e.g. power electronics, BMS, etc.
- Applications may have a **wide range of requirements**: difficult to create generic APIs that fit all usecases
- **Upstream** APIs often do **not** solve complex usecases nor use all hardware features (e.g. ADC)

An API to control locks

Lock API design

- The lock API has the following requirements:
  - Ability to **open** and **close** the lock
  - Operations succeed only if the action is **completed**
- **Multiple** drivers can be implemented:
  - Lock driven by a servo with feedback
  - Lock driven by a DC motor with feedback
  - Lock driven by a DC motor with inductive position sensors
  - ...
- A **fake/mock** device can also be implemented for **testing**

File Tree Layout

```
$ROOT: application module root
|-- drivers
|       |-- CMakeLists.txt
|       |-- Kconfig
|       `-- lock                        # Lock drivers folder
|           |-- CMakeLists.txt          # Lock drivers CMake file
|           |-- Kconfig                 # Lock drivers Kconfig file
|           |-- Kconfig.vnd_dev         # Kconfig for vnd,dev
|           `-- vnd_dev.c               # Driver for vnd,dev
|-- dts
|   `-- bindings
|       `-- lock
|           `-- vnd,dev.yaml            # Devicetree bindings for vnd,dev
`-- include
    `-- app
        `-- drivers
            `-- lock.h                  # Public header with the lock API
```

Listing: File tree with all relevant lock API files

Lock API driver ops

```
typedef int (*lock_open_t)(const struct device *dev);
typedef int (*lock_close_t)(const struct device *dev);

__subsystem struct lock_driver_api {
  lock_open_t open;
  lock_close_t close;
};
```

Listing: Lock API driver ops, `$ROOT/include/app/drivers/lock.h`

Lock API interface (I)

```
/**
* @brief Open the lock.
*
* @param dev Lock device instance.
*
* @retval 0 On success.
* @retval -errno Other negative errno in case of failure.
*/
__syscall int lock_open(const struct device *dev);

static inline int z_impl_lock_open(const struct device *dev)
{
  const struct lock_driver_api *api =
    (const struct lock_driver_api *)dev->api;

  return api->open(dev);
}
...
```

Listing: Lock API interface, `$ROOT/include/app/drivers/lock.h`

Lock API interface (II)

```
...
/**
* @brief Close the lock.
*
* @param dev Lock device instance.
*
* @retval 0 On success.
* @retval -errno Other negative errno in case of failure.
*/
__syscall int lock_close(const struct device *dev);

static inline int z_impl_lock_close(const struct device *dev)
{
  const struct lock_driver_api *api =
    (const struct lock_driver_api *)dev->api;

  return api->close(dev);
}

#include <syscalls/lock.h>
```

Listing: Lock API interface, `$ROOT/include/app/drivers/lock.h`

A note about userspace

- Even if optional, adding support **userspace** increases **complexity**
- Sometimes an application may **never need userspace** support
- Some APIs may always be used in **Kernel space**

```c
/**
* @brief Open the lock (no userspace support).
*
* @param dev Lock device instance.
*
* @retval 0 On success.
* @retval -errno Other negative errno in case of failure.
*/
static inline int lock_open(const struct device *dev)
{
  const struct lock_driver_api *api =
    (const struct lock_driver_api *)dev->api;

  return api->open(dev);
}
```

Listing: Lock API interface without userspace support

A servo-driven lock

A servo-driven lock

- A **PWM controlled servo** controls the **position** of the lock: open or closed
- The servo provides **position feedback** as an **analog** signal proportional to its position

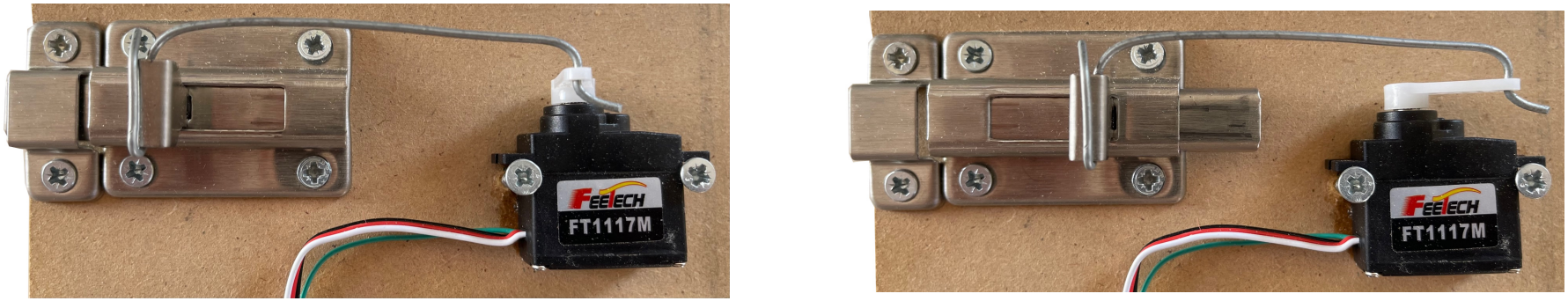


Figure: Servo-driven lock, closed (left) and open (right)

Servo Positioning

- Servos are tipically controlled by a **PWM** signal, making their **position θ proportional** to the **pulse width w**

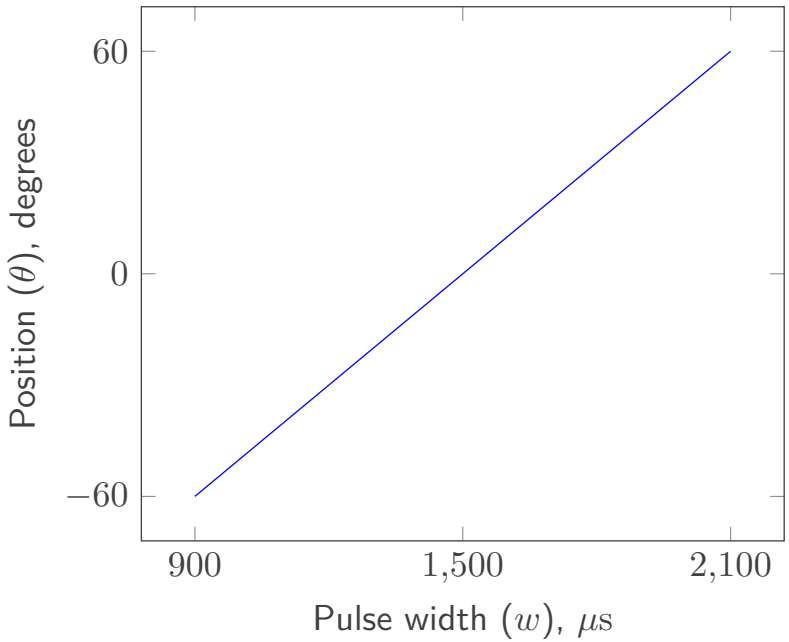


Figure: Position vs. pulse width for the FT1117M servo

Servo Analog Feedback

- The voltage v given by the analog feedback is proportional to servo's current position θ, by a gain factor G with an added offset K:

$$v = G\theta + K$$

- For convenience, and because position is proportional to the pulse width, it can also be written as a function of the pulse width:

$$v = G'w + K$$

Gain	Offset
$-1.765\ \frac{\text{mV}}{\mu\text{s}}$	$4206.250\ \text{mV}$

Table: Gain and offset for the FT1117M servo

Coding session: servo-driven lock

Coding session

Please watch the session

The following slides only provide some highlights

File Tree Layout

```
$ROOT: application module root
|-- CMakeLists.txt
|-- drivers
|       |-- CMakeLists.txt
|       `-- lock                        # Lock drivers folder
|           |-- CMakeLists.txt          # Lock drivers CMake file
|           |-- Kconfig                 # Lock drivers Kconfig file
|           |-- Kconfig.servo           # Kconfig for lock-servo
|           `-- servo.c                 # Driver for lock-servo
`-- dts
    `-- bindings
        `-- lock
            `-- lock-servo.yaml         # Devicetree bindings for lock-servo
```

Listing: File tree with all relevant files for the servo-driven lock driver

Devicetree bindings (I)

```yaml
description: Lock driven by a servo

compatible: "lock-servo"

include: base.yaml

properties:
    pwms:
      required: true
      type: phandle-array
      description: |
        PWM specifier driving the servo motor.

    open-pulse-us:
      required: true
      type: int
      description: |
        Lock open position pulse width (microseconds).

    closed-pulse-us:
      required: true
      type: int
      description: |
        Lock closed position pulse width (microseconds).
    ...
```

Listing: `$ROOT/dts/bindings/lock/lock-servo.yaml`

Devicetree bindings (II)

```
...
io-channels:
  required: true
  type: phandle-array
  description: |
    ADC specifier for servo feedback.

fb-gain:
  required: true
  type: int
  description: |
    Servo feedback gain (uV/usec).

fb-offset:
  required: true
  type: int
  description: |
    Servo feedback offset (uV).
...
```

Listing: $ROOT/dts/bindings/lock/lock-servo.yaml

Devicetree bindings (III)

```
...
max-target-err-us:
  required: true
  type: int
  description: |
    Maximum target pulse error (+/-), in microseconds.

max-action-time-ms:
  required: true
  type: int
  description: |
    Maximum time to complete open/close actions (msec).
```

Listing: $ROOT/dts/bindings/lock/lock-servo.yaml

Kconfig (I)

```
menu "Drivers"
...
rsource "lock/Kconfig"
...
endmenu
```

Listing: `$ROOT/drivers/Kconfig`

Kconfig (II)

```
menuconfig LOCK
bool "Locks"

if LOCK

# define lock logging module
module = LOCK
module-str = lock
source "subsys/logging/Kconfig.template.log_config"

# define lock drivers init priority
config LOCK_INIT_PRIORITY
  int "Lock init priority"
  default 90
  help
    Lock initialization priority.

# include each implementation's Kconfig
rsource "Kconfig.servo"

endif # LOCK
```

Listing: $ROOT/drivers/lock/Kconfig

Kconfig (III)

```
DT_COMPAT_LOCK_SERVO := lock-servo

config LOCK_SERVO
  bool "Servo-controlled lock"
  depends on PWM && ADC
  default $(dt_compat_enabled,$(DT_COMPAT_LOCK_SERVO))
  help
    Enables a servo-controlled lock driver.
```

Listing: $ROOT/drivers/lock/Kconfig.servo

CMake

```
add_subdirectory(drivers)

zephyr_include_directories(include)

# optional, only needed for userspace support
list(APPEND SYSCALL_INCLUDE_DIRS ${CMAKE_CURRENT_SOURCE_DIR}/include)
set(SYSCALL_INCLUDE_DIRS ${SYSCALL_INCLUDE_DIRS} PARENT_SCOPE)
```

Listing: $ROOT/CMakeLists.txt

```
add_subdirectory_ifdef(CONFIG_LOCK lock)
```

Listing: $ROOT/drivers/CMakeLists.txt

```
zephyr_library()
zephyr_library_sources_ifdef(CONFIG_LOCK_SERVO servo.c)
```

Listing: $ROOT/drivers/lock/CMakeLists.txt

Driver highlights (I)

- Driver makes use of the PWM API **DT-spec** facilities
- Allows to **easily collect** all PWM property cells from DT
- Makes code **less verbose**

```
struct lock_servo_config {
  ...
  /* PWM specifier for servo motor */
  struct pwm_dt_spec servo;
  ...
};

/* using *_dt APIs makes code less verbose */
ret = pwm_set_pulse_dt(&config->servo, target_pulse);
...

/* PWM specifier is initialized using PWM_DT_INST_SPEC_GET() */
#define LOCK_SERVO_DEFINE(i)                                    \
  static const struct lock_servo_config lock_servo_config_##i = {  \
    .servo = PWM_DT_INST_SPEC_GET(i),                           \
    ...
```

Listing: Usage of PWM DT-spec facilities within the servo lock driver

Driver highlights (II)

- Driver makes use of the ADC API **DT-spec** facilities
- Allows to **easily collect** all ADC configuration from DT

```
struct lock_servo_config {
  ...
  /* ADC specifier for servo feedback */
  struct adc_dt_spec adc;
  ...
};

/* ADC specifier is initialized using ADC_DT_INST_SPEC_GET() */
#define LOCK_SERVO_DEFINE(i)                                    \
  static const struct lock_servo_config lock_servo_config_##i = {   \
    .adc = ADC_DT_INST_SPEC_GET(i),                             \
    ...
```

Listing: Usage of ADC DT-spec facilities within the servo lock driver

Using the servo-driven lock driver

Devicetree Definition

```
/ {
  lock: lock-servo {
    compatible = "lock-servo";
    pwms = <&pwm0 0 PWM_MSEC(20) PWM_POLARITY_NORMAL>;
    open-pulse-us = <2100>;
    closed-pulse-us = <1250>;
    io-channels = <&adc 0>;
    fb-gain = <(-1765)>;
    fb-offset = <4206250>;
    max-target-err-us = <50>;
    max-action-time-ms = <2000>;
  };
};
```

Listing: Devicetree definition of one servo-driven lock device instance

Configuration of the ADC in Devicetree

```
&adc {
  #address-cells = <1>;
  #size-cells = <0>;

  /* configuration suitable for servo feedback measurements
   * servo feedback range ~= (0.5V - 2.7V)
   * ADC range when using internal ref = (0.0V - 0.6V), gain <= 1/6
   */
  channel@0 {
    reg = <0>;
    zephyr,gain = "ADC_GAIN_1_6";
    zephyr,reference = "ADC_REF_INTERNAL";
    zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
    zephyr,input-positive = <NRF_SAADC_AIN1>;
    zephyr,resolution = <12>;
  };
};
```

Listing: ADC configuration specified in Devicetree (platform specific)

Pin control configuration in Devicetree

```
/* assign PWM0 channel 0 to P1.03 */
&pwm0_default {
  group1 {
    psels = <NRF_PSEL(PWM_OUT0, 1, 3)>;
  };
};

&pwm0_sleep {
  group1 {
    psels = <NRF_PSEL(PWM_OUT0, 1, 3)>;
  };
};
```

Listing: PWM pinctrl overrides for nRF52840DK (platform specific)

Project Configuration

- Servo-driven lock driver is enabled when `CONFIG_LOCK_SERVO=y`
- `CONFIG_LOCK_SERVO` is automatically enabled if:
  - All its **dependencies** are **enabled**, including `CONFIG_LOCK`
  - One or more instances are `okay` in Devicetree

```
# dependencies required so that servo-driven lock driver can be enabled
CONFIG_ADC=y
CONFIG_PWM=y

# enable the lock driver class
CONFIG_LOCK=y
```

Listing: Project configuration (`prj.conf`)

Obtain a Reference to a Device Instance

```c
/* reference resolved at compile time (using unique node label) */
const struct device *lock = DEVICE_DT_GET(DT_NODELABEL(lock));

/* verify for device readiness before operation */
if (!device_is_ready(lock)) {
    printk("Lock device not ready\n");
    return -ENODEV;
}
```

Listing: Obtain a reference to the `lock` device and verify if ready

Example: open and close

```
ret = lock_open(lock);
if (ret < 0) {
  printk("Could not open lock: %d\n", ret);
  return ret;
}

ret = lock_close(lock);
if (ret < 0) {
  printk("Could not close lock: %d\n", ret);
  return ret;
}
```

Listing: Example code to open and close the lock

Conclusions

Conclusions

- **Devicetree** is nowadays **core** to the Zephyr device model
- Devicetree allows to easily **define N instances** of a device, each with its **own** configuration
- Devicetree does not ***exist*** after compiling, **zero overhead**
- Zephyr comes with **APIs**. They can be useful to implement certain driver classes e.g. sensors.
- **Custom** APIs give **freedom** to application developers, while retaining all Zephyr **benefits**
- **Drivers** built upon Zephyr APIs are vendor-agnostic by nature, so **highly portable**
- **Documentation** needs to **improve**: needs tutorials, better API examples, etc.

THANK YOU!

Questions?

https://github.com/teslabs/zds-2022-drivers

https://github.com/teslabs/zds-2022-drivers-app