



Zephyr SPI API

v3.1.0 – v3.0.0

v2.9.0 – v2.7.0

v2.6.2 – v2.5.2

The nRF Connect SDK contains the Zephyr SPI API to interface with the SPI peripheral, and we will use it in the exercises in this lesson.

The choice of the API to use with external peripherals depends on the needs and requirements of the application and the capabilities of the peripheral device. Similarly, it is recommended to use a sensor API to communicate with the sensor and some graphics libraries (like LVGL and others) to communicate with a TFT screen.

Nonetheless, the emphasis of this lesson is on learning and utilizing raw SPI transactions, which those APIs ultimately connect to for the SPI communication with the peripheral. Therefore, for our exercises, we will rely on the Zephyr SPI API, which will provide a solid understanding of using SPI in Zephyr on Nordic chips.

Enabling driver

Enable the SPI driver by adding the following Kconfig to the `prj.conf` file:

```
1 CONFIG_SPI=y
```

In the source code, we include the header file of the SPI API.

```
1 #include <zephyr/drivers/spi.h>
```

Changing the devicetree

First, we must add the SPI slave device on the SPI node in the devicetree using an overlay file. Overlay files define which SPI controller we are using, the device bindings, status, and the configurations to be used for MOSI, MISO and SCLK pins. As per bindings, we can specify other properties for the slave, like max-clock speed. We also specify which driver to use for



this device in the compatible property.

A basic overlay is shown below that uses the `spi1` controller, `nordic-spi` bindings, is active (status okay), defines a CS pin on `gpio0` with the active-low flag and uses the default pin configuration of `spi1` for MISO, MOSI and SCKL. The pin configuration for the active and sleep mode (`spi1_default` and `spi1_sleep` respectively) can be configured using `pinctrl` and is not shown here for conciseness. You will find a complete overlay with `pinctrl` defined in the hands-on exercises.

A general SPI device, `gendev`, is added as a subnode of the `spi1` controller in the code snippet below.

```
1 &spi1 {
2     compatible = "nordic,nrf-spi";
3     status = "okay";
4     cs-gpios = <&gpio0 18 GPIO_ACTIVE_LOW>;
5     pinctrl-0 = <&spi1_default>;
6     pinctrl-1 = <&spi1_sleep>;
7     pinctrl-names = "default", "sleep";
8     gendev: gendev@0 {
9         compatible = "vnd,spi-device";
10        reg = <0>;
11        spi-max-frequency = <1600000>;
12        label = "GenDev";
13    };
14 };
```

Initializing the device

The SPI API has an API-specific `struct spi_dt_spec`, with the following signature

```
struct spi_dt_spec
```

```
#include <spi.h>
```

Complete SPI DT information.

Param bus: is the SPI bus

Param config: is the slave specific configuration



This structure contains the device pointer for the SPI device, `const struct device *bus`, and the slave specific configuration `spi_config`



config.

`struct spi_config` has the following signature

`struct spi_config`

`#include <spi.h>`

SPI controller configuration structure.

Public Members

`uint32_t frequency`

Bus frequency in Hertz.

`spi_operation_t operation`

Operation flags.

It is a bit field with the following parts:

- 0: Master or slave.
- 1..3: Polarity, phase and loop mode.
- 4: LSB or MSB first.
- 5..10: Size of a data frame in bits.
- 11: Full/half duplex.
- 12: Hold on the CS line if possible.
- 13: Keep resource locked for the caller.
- 14: Active high CS logic.
- 15: Motorola or TI frame format (optional).

If `CONFIG_SPI_EXTENDED_MODES` is enabled:

- 16..17: MISO lines (Single/Dual/Quad/Octal).
- 18..31: Reserved for future use.

`uint16_t slave`

Slave number from 0 to host controller slave limit.

`struct spi_cs_control cs`

GPIO chip-select line (optional, must be initialized to zero if not used).



- `frequency`: The clock-frequency for SPI communication.
- `operation`: Operation flags, refer to the [API documentation](#) for different flags defined and their bit positions.
- `slave`: The number of the slave device on the bus.



- `cs`: The GPIO chip-select line.

To retrieve this structure, we will use the API-specific function `SPI_DT_SPEC_GET()`, which has the following signature

`SPI_DT_SPEC_GET(node_id, operation_, delay_)`

Structure initializer for `spi_dt_spec` from devicetree.

This helper macro expands to a static initializer for a `struct spi_dt_spec` by reading the relevant bus, frequency, slave, and cs data from the devicetree.

Important: multiple fields are automatically constructed by this macro which must be checked before use. `spi_is_ready` performs the required `device_is_ready` checks.

Deprecated:

Use `spi_is_ready_dt` instead.

- Parameters:**
- **`node_id`** – Devicetree node identifier for the SPI device whose struct `spi_dt_spec` to create an initializer for
 - **`operation_`** – the desired `operation` field in the struct `spi_config`
 - **`delay_`** – the desired `delay` field in the struct `spi_config`'s `spi_cs_control`, if there is one



In the following code snippet, we retrieve the device structure for the gendev SPI slave that we added in the overlay file, with the SPI operation `SPI_WORD_SET(8)` and `SPI_TRANSFER_MSB`, which here is to say that the `spi-word-size` is 8-bit and the MSB should be transferred first (that is how the data over SPI lines should be interpreted).

```
1 #define SPI0P      SPI_WORD_SET(8) | SPI_TRANSFER_MSB
2
3 struct spi_dt_spec spispec = SPI_DT_SPEC_GET(DT_NODELABEL(gendev),
```

Lastly, we will check if the SPI device is ready using the API-specific function `spi_is_ready_dt()`, which has the following signature

`static inline bool spi_is_ready_dt(const struct spi_dt_spec *spec)`

Validate that SPI bus (and CS gpio if defined) is ready.

- Parameters:**
- **`spec`** – SPI specification from devicetree
- Return values:**
- **`true`** – if the SPI bus is ready for use.
 - **`false`** – if the SPI bus (or the CS gpio defined) is not ready for use



```
1 err = spi_is_ready_dt(&spispec);
2 if (!err) {
```



```
3 LOG_ERR("Error: SPI device is not ready, err: %d", err);
4 return 0;
5 }
```

SPI read and write

To read and write data to and from an SPI bus, we have the functions `spi_read_dt()`, `spi_write_dt()`, and `spi_transceive_dt()`. They are very similar, except that `spi_read_dt` only performs the read operation, `spi_write_dt` only performs the write operation, and `spi_transceive_dt` performs both read and write operations.

These functions have the following signatures

```
static inline int spi_read_dt(const struct spi_dt_spec *spec, const struct spi_buf_set *rx_bufs)
```

Read data from a SPI bus specified in `spi_dt_spec`.

This is equivalent to:

```
spi_read(spec->bus, &spec->config, rx_bufs);
```

Parameters:

- **spec** – SPI specification from devicetree
- **rx_bufs** – Buffer array where data to be read will be written to.

Returns: a value from `spi_read()`.



```
static inline int spi_write_dt(const struct spi_dt_spec *spec, const struct spi_buf_set *tx_bufs)
```

Write data to a SPI bus specified in `spi_dt_spec`.

This is equivalent to:

```
spi_write(spec->bus, &spec->config, tx_bufs);
```

Parameters:

- **spec** – SPI specification from devicetree
- **tx_bufs** – Buffer array where data to be sent originates from.

Returns: a value from `spi_write()`.



```
static inline int spi_transceive_dt(const struct spi_dt_spec *spec, const struct spi_buf_set *tx_bufs, const struct spi_buf_set *rx_bufs)
```

Read/write data from an SPI bus specified in `spi_dt_spec`.

This is equivalent to:

```
spi_transceive(spec->bus, &spec->config, tx_bufs, rx_bufs);
```

Parameters:

- **spec** – SPI specification from devicetree
- **tx_bufs** – Buffer array where data to be sent originates from, or NULL if none.
- **rx_bufs** – Buffer array where data to be read will be written to, or NULL if none.

Returns: a value from `spi_transceive()`.



Notice that all functions take in the SPI-specific device structure `spi_dt_spec`, and one or two buffer pointers, for the transmission and the reception.

Below is an example of using the `spi_transceive_dt` function

```
1  uint8_t tx_buffer = 0x88;
2  struct spi_buf tx_spi_buf = {.buf = (void *)&tx_buffer, .len = 1};
3  struct spi_buf_set tx_spi_buf_set = {.buffers = &tx_spi_buf, .count = 1};
4  struct spi_buf rx_spi_bufs = {.buf = data, .len = size};
5  struct spi_buf_set rx_spi_buf_set = {.buffers = &rx_spi_bufs, .count = 1};
6
7
8  err = spi_transceive_dt(&spispec, &tx_spi_buf_set, &rx_spi_buf_set);
9  if (err < 0) {
10     LOG_ERR("spi_transceive_dt() failed, err: %d", err);
11     return err;
12 }
```

The same procedure can be followed if only reading or writing is required using `spi_read_dt` or `spi_write_dt`.

