



UART & SPI

Lecture 5



UART & SPI

used by RP2350

- Direct Memory Access
- Buses
 - Universal Asynchronous Receiver and Transmitter
 - Serial Peripheral Interface
- Analog and Digital Sensors



DMA

Direct Memory Access



Bibliography

for this section

Raspberry Pi Ltd, RP2350 Datasheet

- Chapter 12 - *Peripherals*
 - Chapter 16.6 - *DMA*

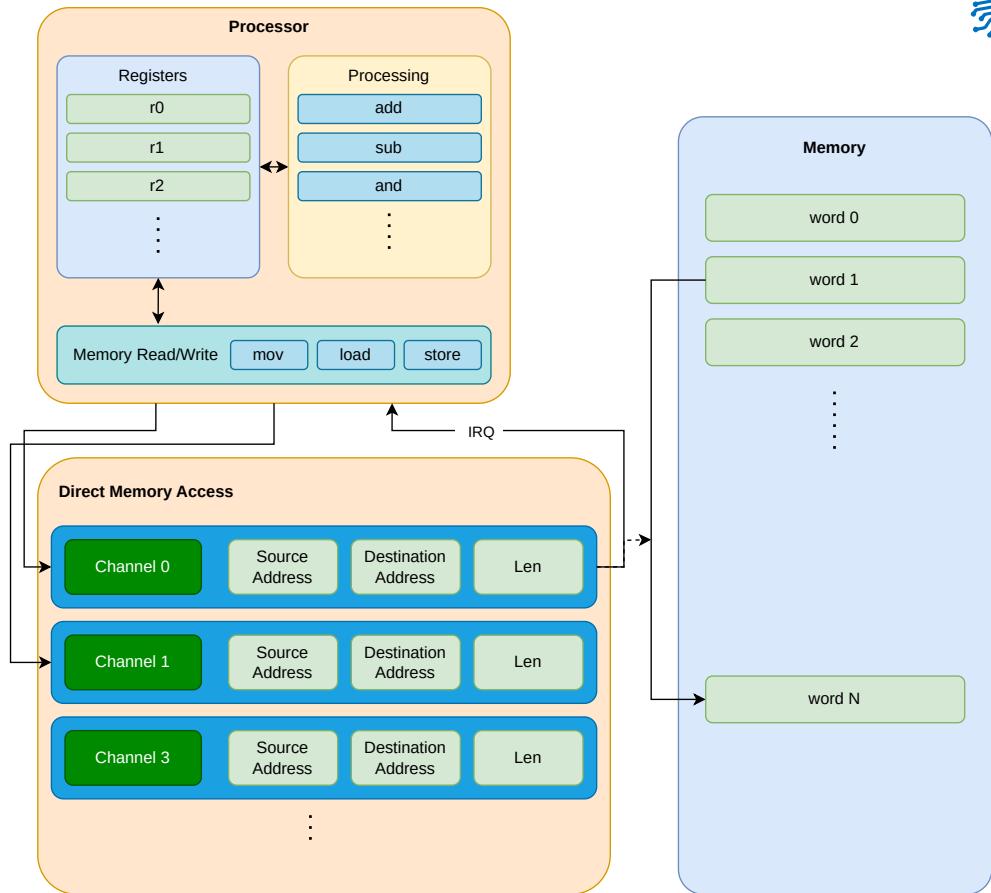


DMA

- offloads the MCU from doing **memory to memory** operations
- due to MMIO, usually implies **transfers from and to peripherals**
- raises an interrupt when a transfer is done

⚠ DMA does not know about the data stored in cache.

- for chips that use cache
 - the DMA buffer's memory region has to be set manually to *nocache* (if MCU knows)
 - or, the cache has to be flushed before and, possibly after, a DMA transfer





UART

Universal Asynchronous Receiver and Transmitter



Bibliography

for this section

1. **Raspberry Pi Ltd**, *RP2350 Datasheet*

- Chapter 12 - *Peripherals*
 - Chapter 12.1 - *UART*

2. **Paul Denisowski**, *Understanding Serial Protocols*

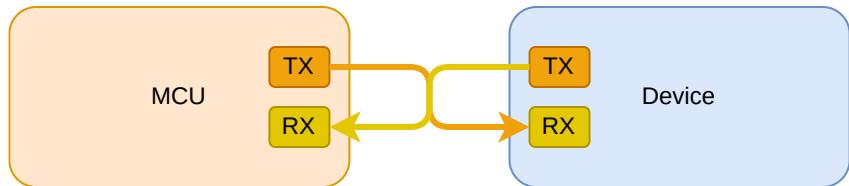
3. **Paul Denisowski**, *Understanding UART*



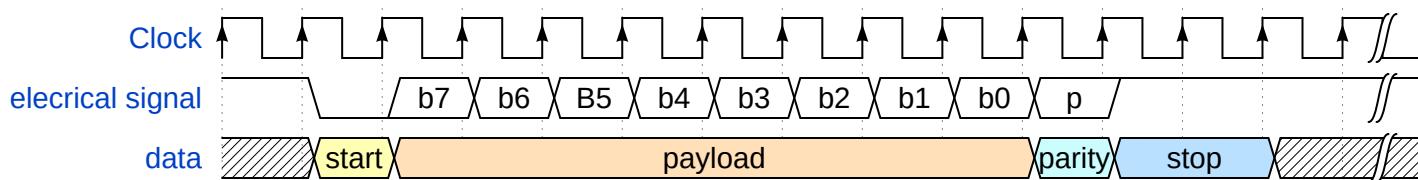
UART

aka serial port

- connects **two devices**
- uses two **independent** wires
 - *TX* - transmission wire
 - *RX* - reception wire
- cross-connected



Transmission example





UART Device

properties

bits

the number of bits in the payload, between 5 and 9

parity

add or not the parity bit

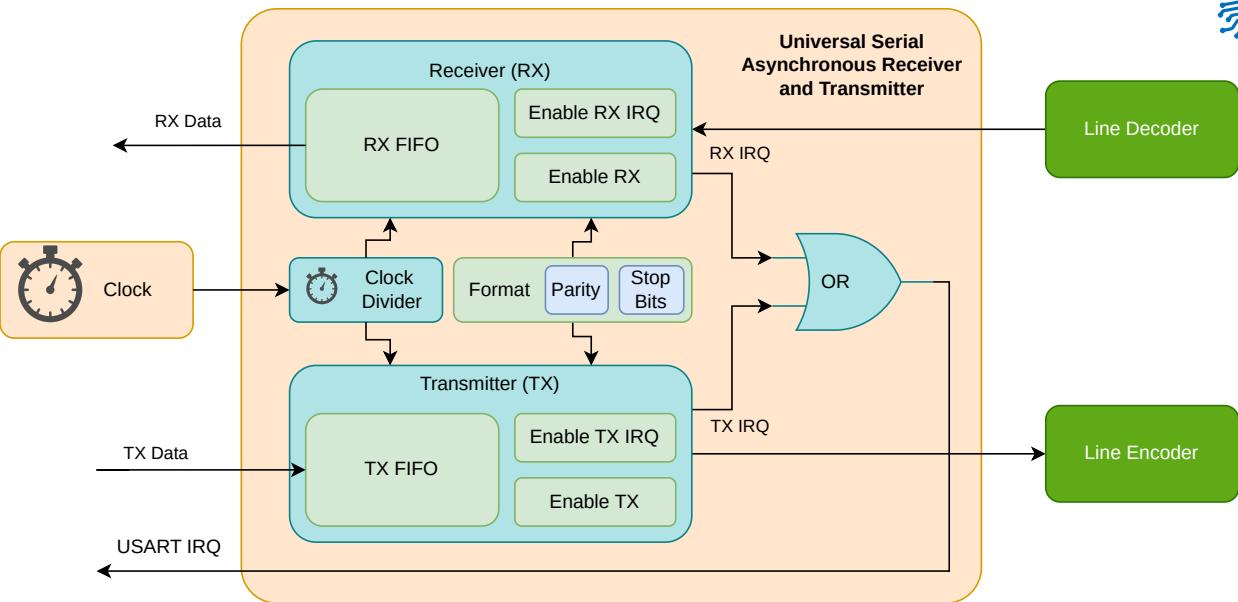
stop

the number of stop bits to add, 1 or 2

baud

rate

number of elements sent per s, most used **9600** or **115200**



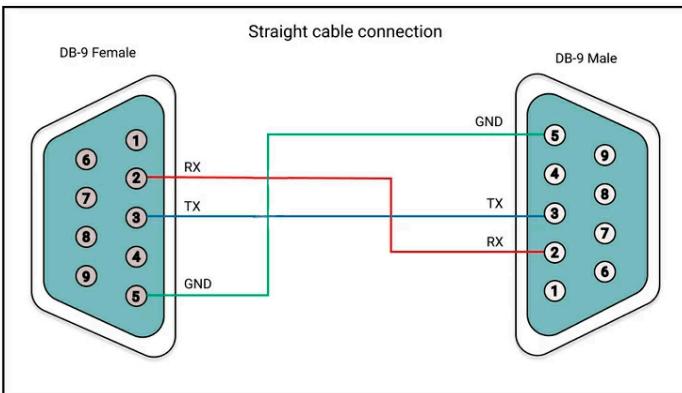
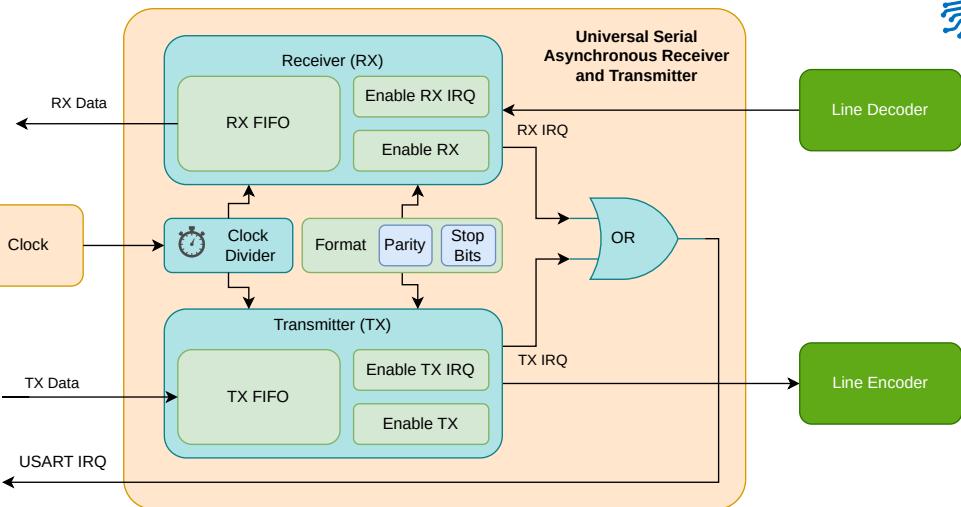
$$baud_{rate} = \frac{f_{clock}}{divider \times (1 + payload_{bits} + parity_{bits} + stop_{bits})}$$



UART Device

types

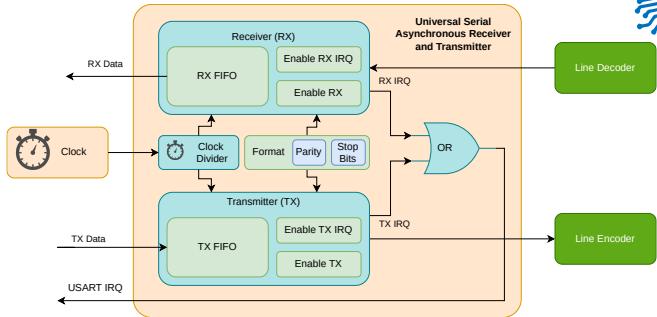
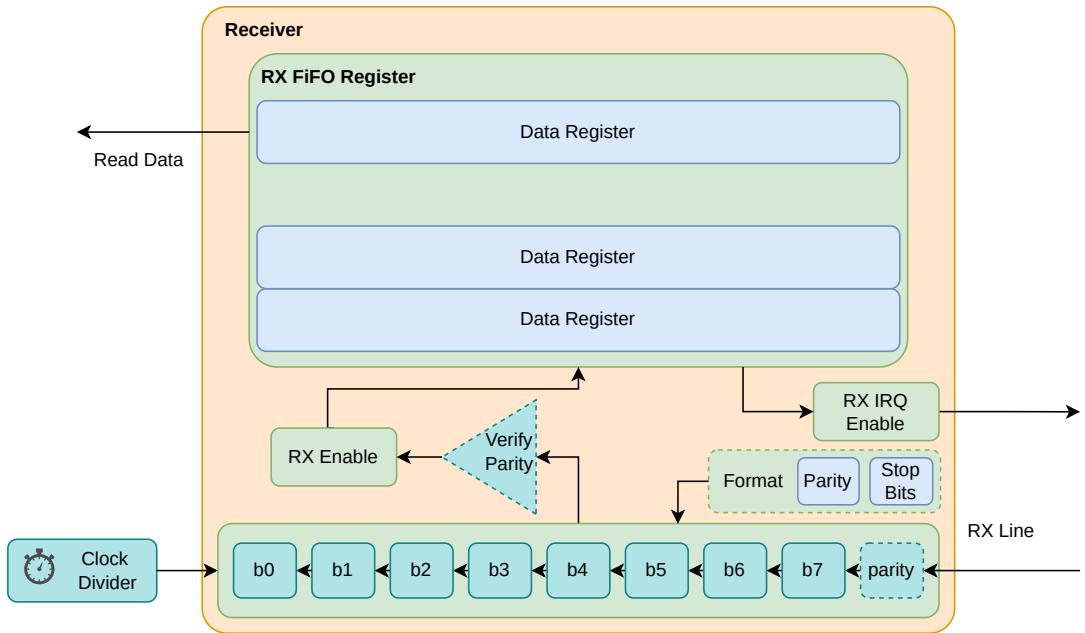
- **TTL - Transistor Transistor Logic**
connects devices at 0 - 3.3V or 0 - 5V, used for short cables and jumper wires
- **RS232** - used for external connections and longer cables, uses -12V to 12V.
- **RS485** - industrial, uses differential voltage





Receiver

RX part of the serial port

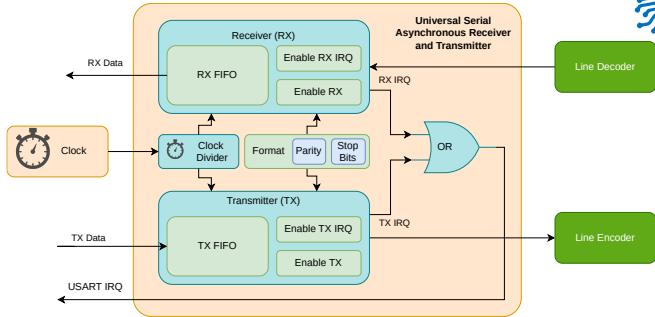
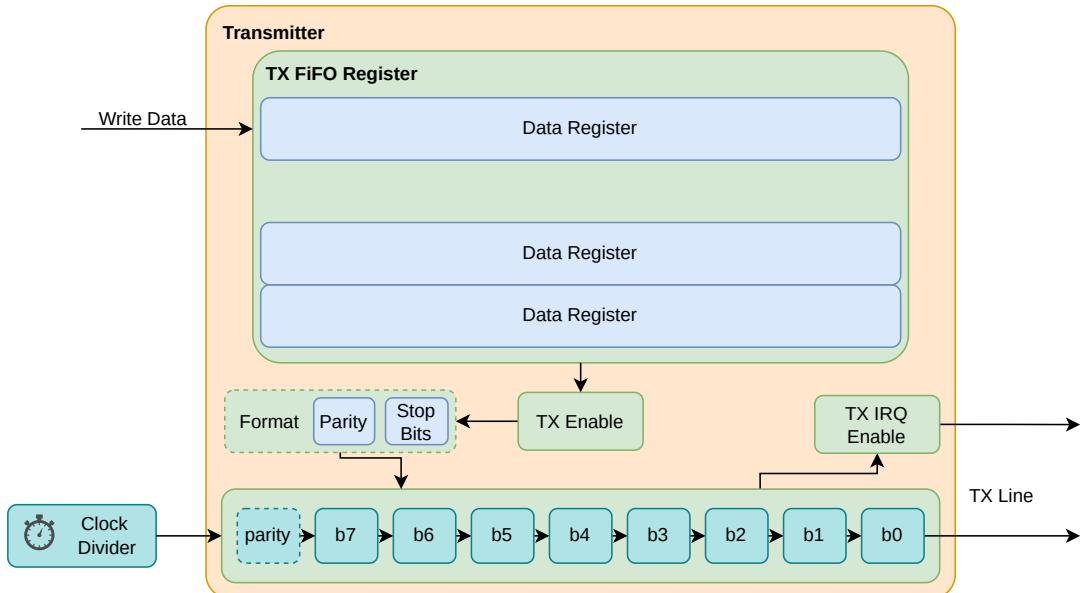


- *Shift Register to read serially every bit*
- Triggers an interrupt
 - when data was received
 - (*optional*) when FIFO is half full
 - (*optional*) when FIFO is full
- FIFO is optional
 - may have a capacity of 1



Transmitter

TX part of the serial port



- *Shift Register to output serially every bit*
- Triggers an interrupt
 - when data was sent
 - (*optional*) when FIFO is half empty
 - (*optional*) when FIFO is empty
- FIFO is optional
 - may have a capacity of 1



Transmission Examples

Setup	Payload	Parity	Stop
8N1	8 bits	no	1 bit
8P2	8 bits	yes	2 bits
9P1	9 bits	yes	1 bit

Diagram illustrating the timing and structure of three transmission formats: 8N1, 8P2, and 9P1.

The diagram shows a common Clock signal at the top, followed by three parallel data streams. Each stream consists of an electrical signal line and a data line. The data is divided into segments labeled b7, b6, B5, b4, b3, b2, b1, b0, and p (parity). A yellow box labeled "start" marks the beginning of each segment. A blue box labeled "stop" marks the end of the payload and the start of the parity. Shaded regions indicate idle time between frames.

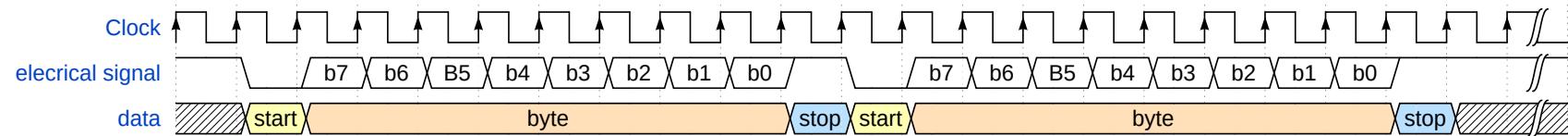
- 8N1:** Shows 8 bits of payload without parity. The data line starts with a hatched region, followed by the start of the payload (b7), then the payload (b6, B5, b4, b3, b2, b1, b0), and ends with a stop (blue hatched region).
- 8P2:** Shows 8 bits of payload with 1 parity bit. The data line starts with a hatched region, followed by the start of the payload (b7), then the payload (b6, B5, b4, b3, b2, b1, b0), then a parity bit (p), and ends with a stop (blue hatched region).
- 9P1:** Shows 9 bits of payload with 1 parity bit. The data line starts with a hatched region, followed by the start of the payload (b8), then the payload (b7, b6, B5, b4, b3, b2, b1, b0), then a parity bit (p), and ends with a stop (blue hatched region).



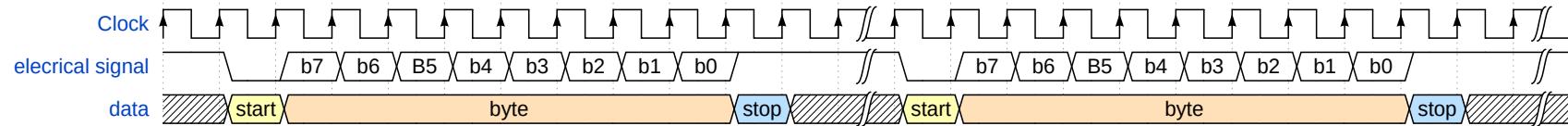
Successive Transmission

using the 8N1 data format

Back to back



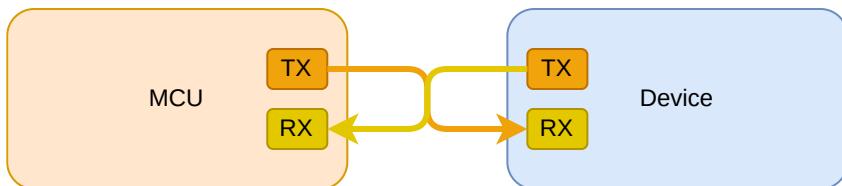
With delay





Facts

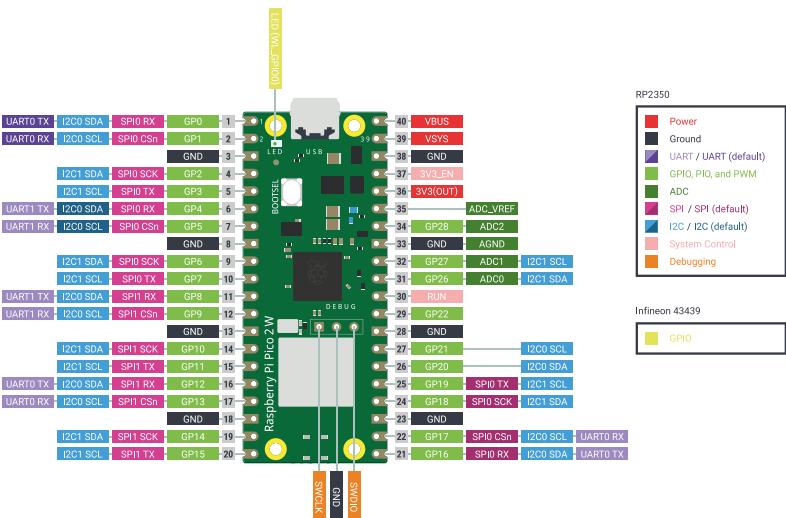
Transmission	<i>duplex</i>	data can be sent in both directions at the same time
Clock	<i>independent</i>	there is no clock sent between the two devices, the receiver has to synchronize its clock with the transmitter to be able to correctly read the received data
Wires	<i>RX / TX</i>	one receive write, one transmit wire, independent of each other
Devices	2	a receiver and a transmitter
Speed	<i>115 KB/s</i>	usually a maximum baud rate of 115200 is used





Usage

- print debug information
- device console
- RP2350 has two USART devices





Embassy API

for RP2350, synchronous

```
pub struct Config {  
    pub baudrate: u32,  
    pub data_bits: DataBits,  
    pub stop_bits: StopBits,  
    pub parity: Parity,  
    pub invert_tx: bool,  
    pub invert_rx: bool,  
    pub invert_rts: bool,  
    pub invert_cts: bool,  
}
```

```
pub enum DataBits {  
    DataBits5,  
    DataBits6,  
    DataBits7,  
    DataBits8,  
}
```

```
pub enum StopBits {  
    STOP1,  
    STOP2,  
}
```

```
pub enum Parity {  
    ParityNone,  
    ParityEven,  
    ParityOdd,  
}
```

```
1  use embassy_rp::uart::Config as UartConfig;  
2  let config = UartConfig::default();  
3  
4  // use UART0, Pins 0 and 1  
5  let mut uart = uart::Uart::new_blocking(p.UART0, p.PIN_0, p.PIN_1, config);  
6  // write  
7  uart.blocking_write("Hello World!\r\n".as_bytes());  
8  
9  // read 5 bytes  
10 let mut buf = [0; 5];  
11 uart.blocking_read(&mut buf);
```



Embassy API

for RP2350, asynchronous

```
1  use embassy_rp::uart::Config as UartConfig;
2
3  bind_interrupts!(struct Irqs {
4      UART0_IRQ => BufferedInterruptHandler<UART0>;
5  });
6
7  let config = UartConfig::default();
8
9  // use UART0, Pins 0 and 1
10 let mut uart = uart::Uart::new(p.UART0, p.PIN_0, p.PIN_1, Irqs, p.DMA_CH0, p.DMA_CH1, config);
11
12 // write
13 uart.write("Hello World!\r\n".as_bytes()).await;
14
15 // read 5 bytes
16 let mut buf = [0; 5];
17 uart.read(&mut buf).await;
```



SPI

Serial Peripheral Interface



Bibliography

for this section

1. Raspberry Pi Ltd, *RP2350 Datasheet*

- Chapter 12 - *Peripherals*
 - Chapter 12.3 - *SPI*

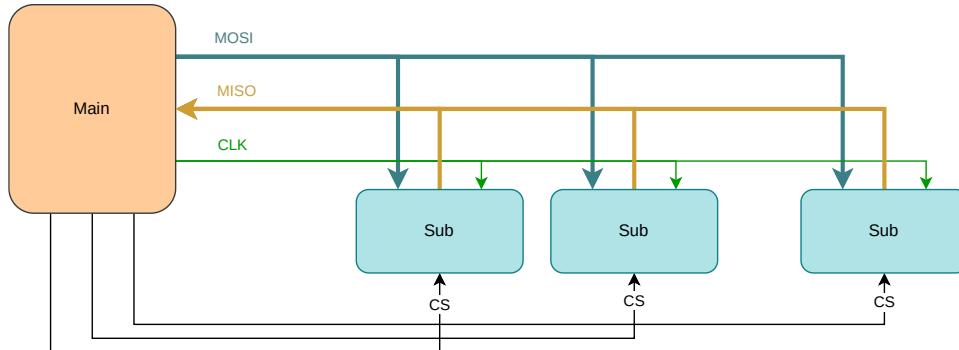
2. Paul Denisowski, *Understanding SPI*



SPI

a.k.a *spy*

- Used for communication between integrated circuits
- Sensors usually expose an *SPI* and an *I2C* interface
- Two device types:
 - *main* (master) - controls the communication (usually MCU)
 - *sub* (slave) - receive and transmit data when the *main* requests (usually the sensor)

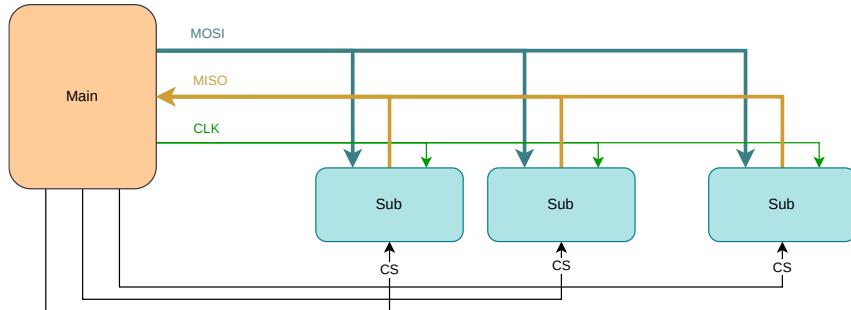
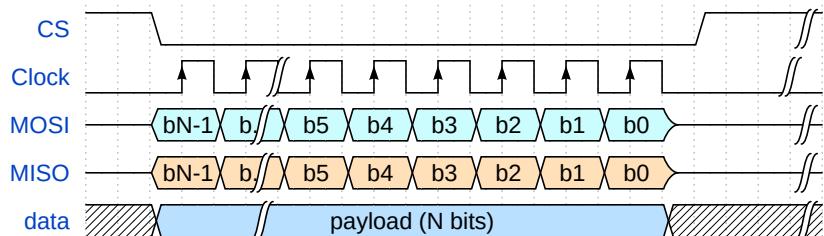




Wires

3 + n

- **MOSI - Main Out Sub In** - carries data from the **main** to the **subs**
- **MISO - Main In Sub Out** - carries data from the active **sub** to the **main**
- **CLK** - Clock signal generated by the **main**, **subs** sample and write data to the bus only on the clock edge
- **CS* - Chip Select** - not actually part of SPI, one wire / sub, activates **one sub at a time**
 - inactive subs have to disconnect from the *MOSI* and *MISO* lines

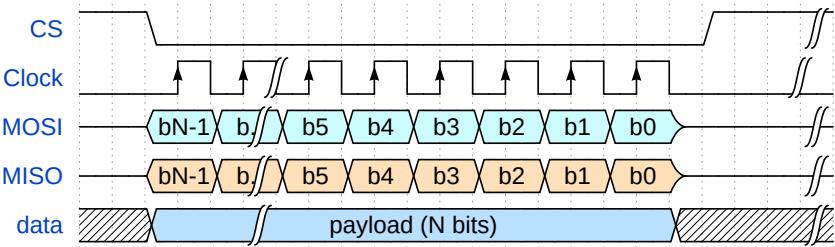




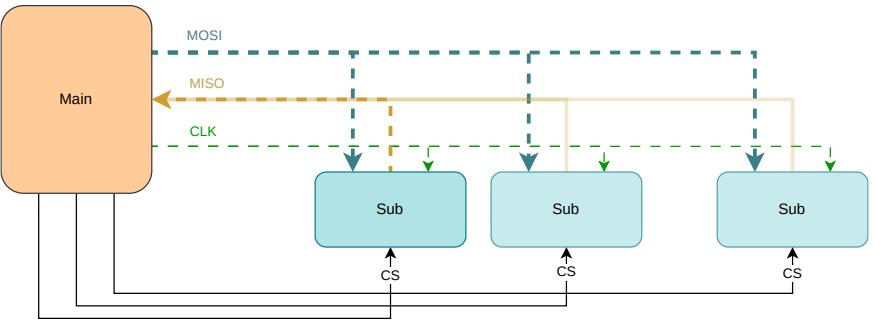
Transmission Example

1. **main** activates the sub device
 - sets the `CS` signal to `LOW`
2. at the same time
 - **main** puts the first bit on the `MOSI` line
 - **sub** puts the first bit on the `MISO` line
3. **main** starts the clock
4. at the *rising edge*
 - **main** reads the data from the `MISO` line
 - **sub** reads the data from the `MOSI` line
5. on the *falling edge*
 - **main** puts the next bit on the `MOSI` line
 - **sub** puts the next bit on the `MISO` line
6. repeat 4 and 5 until **main** decides to stop the clock

SPI Signals



SPI Network

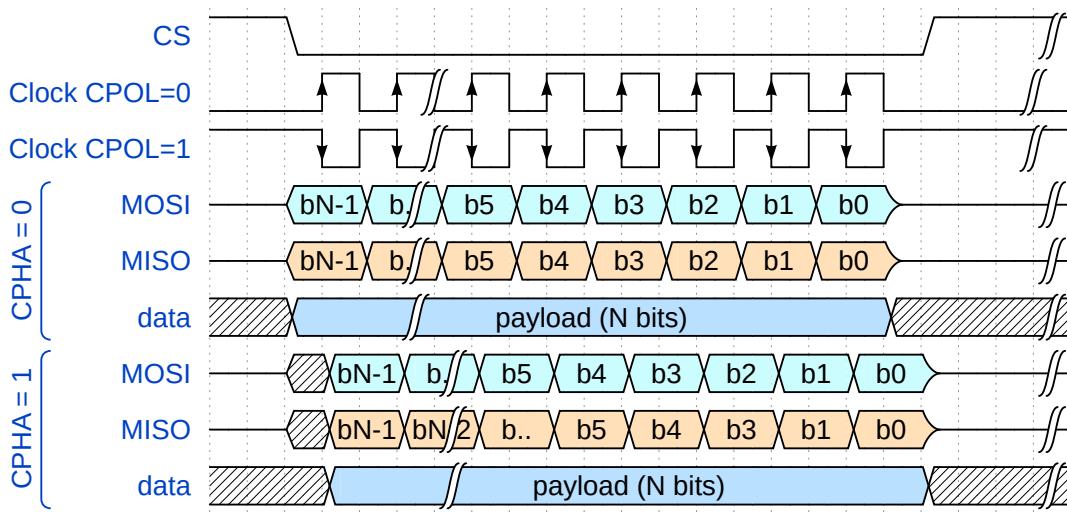




SPI Modes

when data is read and written

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1



defines when the data bit is read

- 0: *rising edge*
- 1: *falling edge*

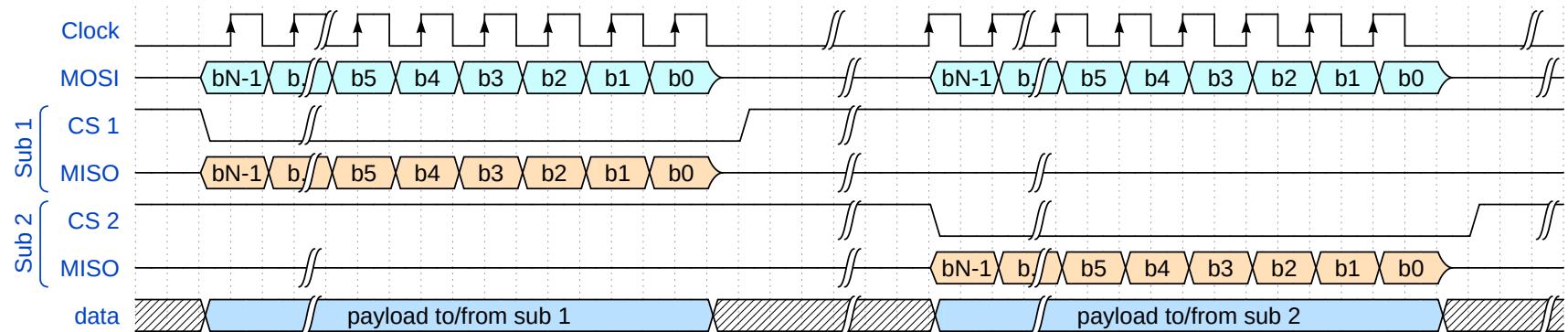
defines when the data is written to the line

- 0: when `CS` activates or *clock edge*
- 1: on *clock edge* (depends on `CPOL`)



Transmission Example

one main, two subs



1. **main** activates the CS pin of **sub 1**
2. **main** writes the first bit on MOSI, **sub 1** writes the first bit on MISO
3. **main** starts the clock
4. **main** and **sub 1** send the rest of the bits
5. **main** stops the clock
6. **main** deactivates the CS pin of **sub 1**

7. **main** activates the CS pin of **sub 2**
8. **main** writes the first bit on MOSI, **sub 2** writes the first bit on MISO
9. **main** starts the clock
10. **main** and **sub 2** send the rest of the bits
11. **main** stops the clock
12. **main** deactivates the CS pin of **sub 2**

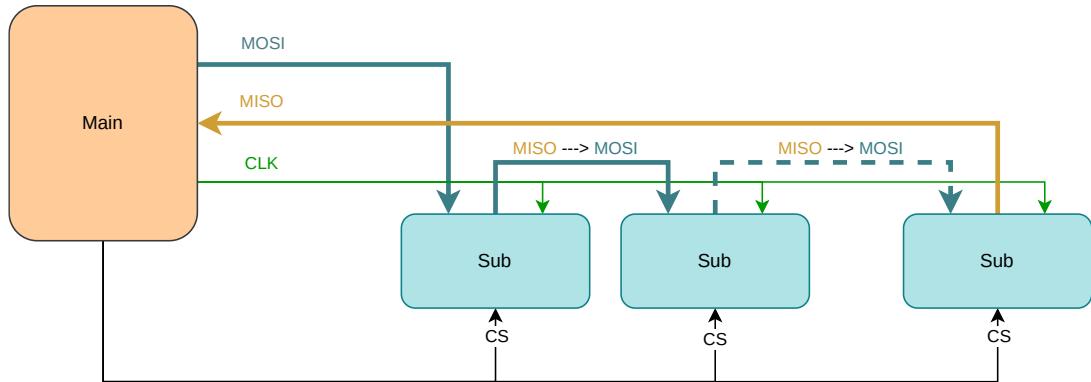


Daisy Chaining

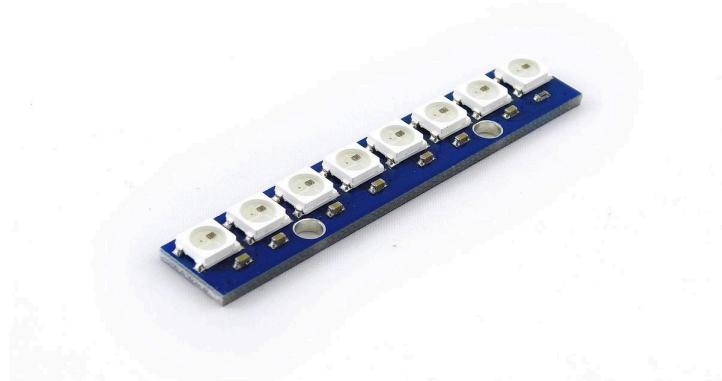
using several SPI devices together

1. **main** activates all the **subs**
2. on the clock edge
 - **main** sends data to **sub 1**
 - **sub 1** sends data to **sub 2**
 - ...
 - **sub n-1** sends data to **sub n**
 - **sub n** sends data to **main**

activate all the **sub** devices



1. usually **subs** send the previous data bit received from **main** to the **next sub** ↵





Facts

Transmission	<i>duplex</i>	data must be sent in both directions at the same time
Clock	<i>synchronized</i>	the main and sub use the same clock, there is no need for clock synchronization
Wires	<i>MISO / MOSI / CLK / CS</i>	different read and write wires, a clock wire and an <i>optional</i> chip select wire for every sub
Devices	<i>1 main several subs</i>	a receiver and a transmitter
Speed	<i>no limit</i>	does not have any limit, it is limited by the main clock and the electronics wirings



Usage

- EEPROMs / Flash (usually in *QSPI* mode)
 - Raspberry Pi Pico has its 2MB Flash connected using *QSPI*
- sensors
- small displays
- RP2350 has two SPI devices



Embassy API

for RP2040, synchronous

```
pub struct Config {  
    pub frequency: u32,  
    pub phase: Phase,  
    pub polarity: Polarity,  
}
```

```
pub enum Phase {  
    CaptureOnFirstTransition,  
    CaptureOnSecondTransition,  
}
```

```
pub enum Polarity {  
    IdleLow,  
    IdleHigh,  
}
```

```
1  use embassy_rp::spi::Config as SpiConfig;  
2  let mut config = SpiConfig::default();  
3  config.frequency = 2_000_000;  
4  
5  let miso = p.PIN_12;  
6  let mosi = p.PIN_11;  
7  let clk = p.PIN_10;  
8  let mut spi = Spi::new_blocking(p.SPI1, clk, mosi, miso, config);  
9  
10 // Configure CS  
11 let mut cs = Output::new(p.PIN_X, Level::Low);  
12  
13 cs.set_low();  
14 let mut buf = [0x90, 0x00, 0x00, 0xd0, 0x00, 0x00];  
15 spi.blocking_transfer_in_place(&mut buf);  
16 cs.set_high();
```



Embassy API

for RP2040, asynchronous

```
1  use embassy_rp::spi::Config as SpiConfig;
2  let mut config = SpiConfig::default();
3  config.frequency = 2_000_000;
4
5  let miso = p.PIN_12;
6  let mosi = p.PIN_11;
7  let clk = p.PIN_10;
8  let mut spi = Spi::new(p.SPI1, clk, mosi, miso, p.DMA_CH0, p.DMA_CH1, config);
9
10 // Configure CS
11 let mut cs = Output::new(p.PIN_X, Level::Low);
12
13 cs.set_low();
14 let tx_buf = [1_u8, 2, 3, 4, 5, 6];
15 let mut rx_buf = [0_u8; 6];
16 spi.transfer(&mut rx_buf, &tx_buf).await;
17 cs.set_high();
```



Sensors

Analog and Digital Sensors



Bibliography

for this section

BOSCH, BMP280 Digital Pressure Sensor

- Chapter 3 - *Functional Description*
- Chapter 4 - *Global memory map and register description*
- Chapter 5 - *Digital Interfaces*
 - Subchapter 5.3 - *SPI Interface*

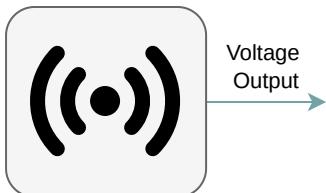


Sensors

analog and digital

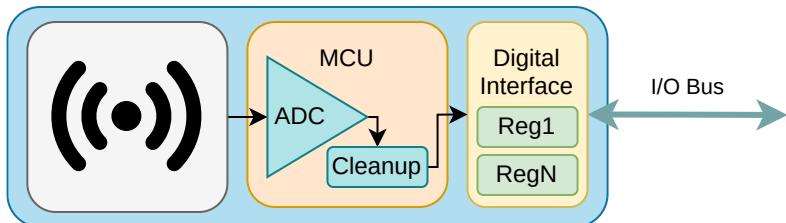
Analog

- only the transducer (the analog sensor)
- outputs (usually) voltage
- requires:
 - an ADC to be read
 - cleaning up the noise



Digital

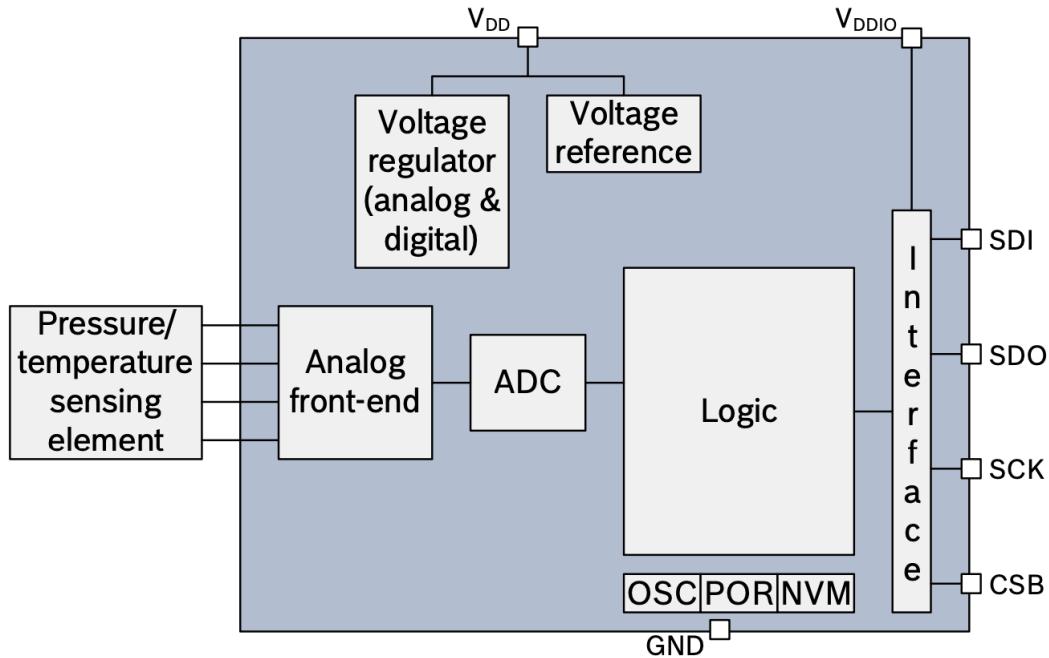
- consists of:
 - a transducer (the analog sensor)
 - an ADC
 - an MCU for cleaning up the noise
- outputs data using a digital bus





BMP280 Digital Pressure Sensor

schematics



Datasheet



BMP280 Digital Pressure Sensor

registers map

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
temp_xlsb	0xFC		temp_xlsb<7:4>			0	0	0	0	0x00
temp_lsb	0xFB				temp_lsb<7:0>					0x00
temp_msb	0xFA				temp_msb<7:0>					0x80
press_xlsb	0xF9		press_xlsb<7:4>			0	0	0	0	0x00
press_lsb	0xF8				press_lsb<7:0>					0x00
press_msb	0xF7				press_msb<7:0>					0x80
config	0xF5	t_sb[2:0]			filter[2:0]			spi3w_en[0]		0x00
ctrl_meas	0xF4	osrs_t[2:0]			osrs_p[2:0]		mode[1:0]			0x00
status	0xF3				measuring[0]			im_update[0]		0x00
reset	0xE0			reset[7:0]						0x00
id	0xD0			chip_id[7:0]						0x58
calib25...calib00	0xA1...0x88			calibration data						individual

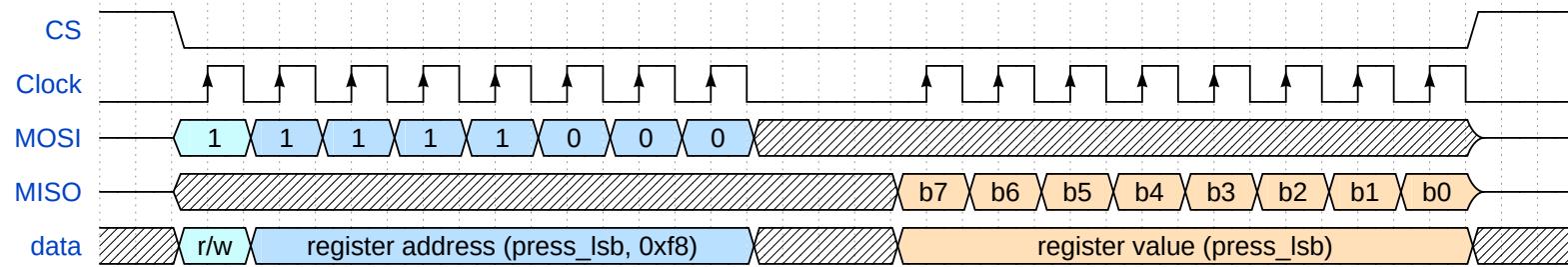
Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Revision	Reset
Type:	do not write	read only	read / write	read only	read only	read only	write only

Datasheet



Reading from a digital sensor

using synchronous/asynchronous SPI to read the `press_lsb` register of BMP280



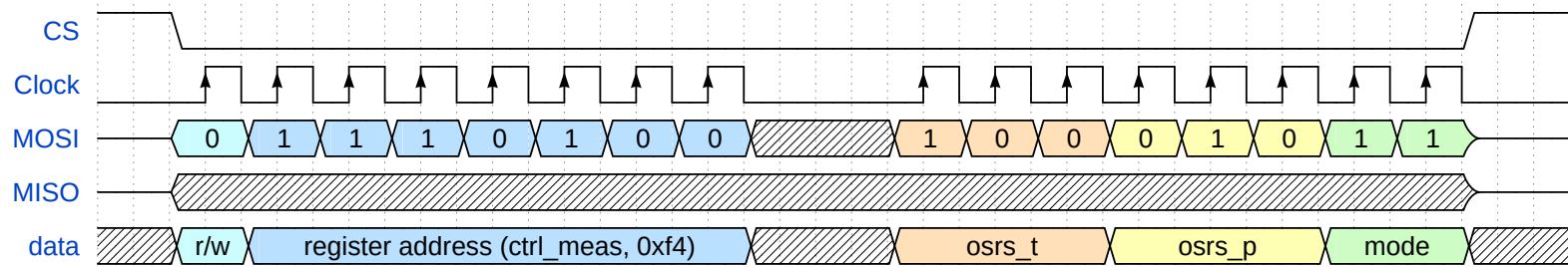
```
1 const REG_ADDR: u8 = 0xf8;
2
3 // enable the sensor
4 cs.set_low();
5
6 // buffer[2]: the address and "empty" value
7 let mut buf = [(1 << 7) | reg, 0x00];
8 spi.blocking_transfer_in_place(&mut buf);
9
10 // disable the sensor
11 cs.set_high();
12
13 // use the value
14 let pressure_lsb = buf[1];
```

```
1 const REG_ADDR: u8 = 0xf8;
2
3 // enable the sensor
4 cs.set_low();
5
6 // two buffers[2], writing and reading
7 let tx_buf = [(1 << 7) | REG_ADDR, 0x00];
8 let mut rx_buf = [0u8; 2];
9 spi.transfer(&mut rx_buf, &tx_buf).await;
10
11 // disable the sensor
12 cs.set_high();
13
```



Writing to a digital sensor

using synchronous/asynchronous SPI to set up the `ctrl_meas` register of the BMP280 sensor



```
1 const REG_ADDR: u8 = 0xf4;  
2  
3 // see subchapters 3.3.2, 3.3.1 and 3.6  
4 let value = 0b100_010_11;  
5  
6 // enable the sensor  
7 cs.set_low();  
8  
9 // buffer[2]: the address and "empty" value  
10 let mut buf = [!(1 << 7) & reg, value];  
11 spi.blocking_transfer_in_place(&mut buf);  
12  
13 // disable the sensor  
14 cs.set_high();
```

```
1 const REG_ADDR: u8 = 0xf4;  
2  
3 // see subchapters 3.3.2, 3.3.1 and 3.6  
4 let value = 0b100_010_11;  
5  
6 // enable the sensor  
7 cs.set_low();  
8  
9 // two buffers[2], writing and reading (ignored)  
10 let tx_buf = [!(1 << 7) & REG_ADDR, value];  
11 let mut rx_buf = [0u8; 2];  
12 spi.transfer(&mut rx_buf, &tx_buf).await;  
13
```



Conclusion

we talked about

- Direct Memory Access
- Buses
 - Universal Asynchronous Receiver and Transmitter
 - Serial Peripheral Interface
- Analog and Digital Sensors