



# I2C & USB 2.0

Lecture 7



# I2C & USB 2.0

used by RP2040

- Buses
  - Inter-Integrated Circuit
  - Universal Serial Bus v2.0



# I2C

Inter-Integrated Circuit



# Bibliography

for this section

## 1. **Raspberry Pi Ltd**, *RP2040 Datasheet*

- Chapter 4 - *Peripherals*
  - Chapter 4.3 - *I2C*

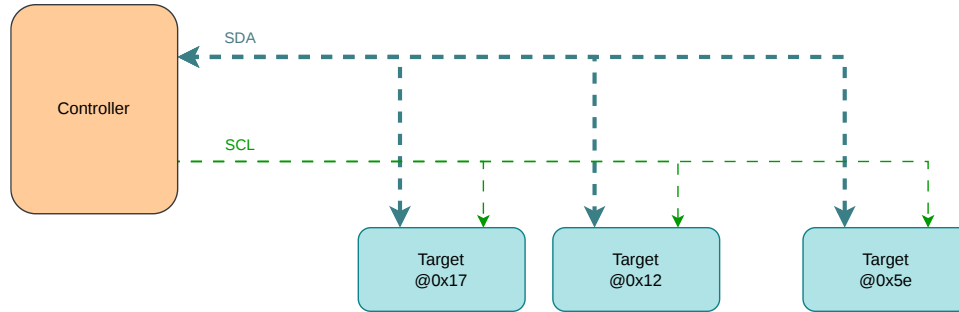
## 2. **Paul Denisowski**, *Understanding I2C*



# I2C

a.k.a *I square C*

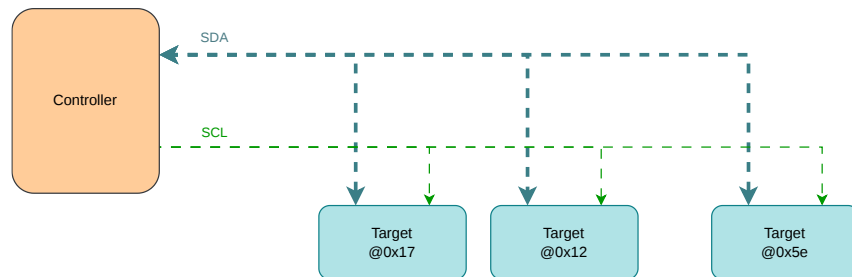
- Used for communication between integrated circuits
- Sensors usually expose an *SPI* and an *I2C* interface
- Two device types:
  - *controller* (master) - initiates the communication (usually MCU)
  - *target* (slave) - receive and transmit data when the *controller* requests (usually the sensor)





# Wires & Addresses

- **SDA** - **S**erial **D**Ata line - carries data from the **controller** to the **target** or from the **target** to the **controller**
- **SCL** - **S**erial **C**Lock line - the clock signal generated by the **controller**, **targets**
  - *sample* data when the clock is *low*
  - *write* data to the bus only when the clock is *high*
- each *target* has a unique address of **7 bits** or **10 bits**
- wires are never driven with **LOW** or **HIGH**
  - are always *pull-up*, which is **HIGH**
  - devices *pull down* the lines to *write* **LOW**





# Transmission Example

7 bit address

1. **controller** issues a **START** condition
  - pulls the **SDA** line **LOW**
  - waits for  $\sim 1/2$  clock periods and starts the clock
2. **controller** sends the address of the **target**
3. **controller** sends the command bit ( **R/W** )
4. **target** sends **ACK** / **NACK** to **controller**
5. **controller** or **target** sends data (depends on **R/W** )
  - receives **ACK** / **NACK** after every byte
6. **controller** issues a **STOP** condition
  - stops the clock
  - pulls the **SDA** line **HIGH** while **CLK** is **HIGH**

Address Format



Transmission



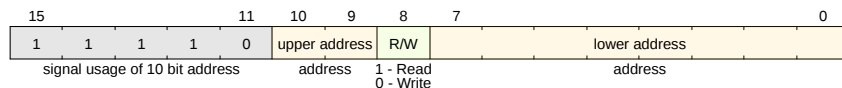


# Transmission Example

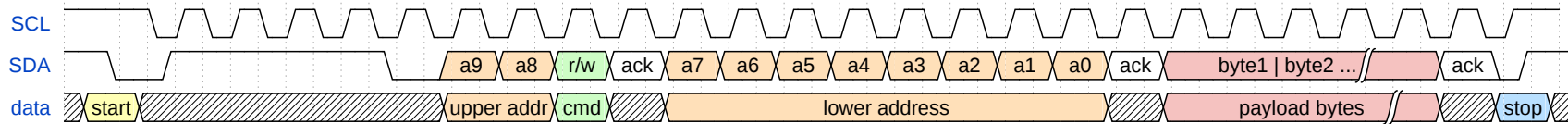
10 bit address

1. **controller** issues a **START** condition
2. **controller** sends **11110** followed by the *upper address* of the **target**
3. **controller** sends the command bit ( **R/W** )
4. **target** sends **ACK / NACK** to **controller**
5. **controller** sends the *lower address* of the **target**
6. **target** sends **ACK / NACK** to **controller**
7. **controller** or **target** sends data (depends on **R/W** )
  - receives **ACK / NACK** after every byte
8. **controller** issues a **STOP** condition

## Address Format



## Transmission



**controller** writes each bit when **CLK** is **LOW** , **target** samples every bit when **CLK** is **HIGH**





# I2C Modes

| Mode                  | Speed      | Capacity | Drive      | Direction      |
|-----------------------|------------|----------|------------|----------------|
| Standard mode (Sm)    | 100 kbit/s | 400 pF   | Open drain | Bidirectional  |
| Fast mode (Fm)        | 400 kbit/s | 400 pF   | Open drain | Bidirectional  |
| Fast mode plus (Fm+)  | 1 Mbit/s   | 550 pF   | Open drain | Bidirectional  |
| High-speed mode (Hs)  | 1.7 Mbit/s | 400 pF   | Open drain | Bidirectional  |
| High-speed mode (Hs)  | 3.4 Mbit/s | 100 pF   | Open drain | Bidirectional  |
| Ultra-fast mode (UFm) | 5 Mbit/s   | ?        | Push-pull  | Unidirectional |



# Facts

|              |   |  |
|--------------|---|--|
| Transmission | <i>half duplex</i>                          | data must be sent in one direction at one time   |
| Clock        | <i>synchronized</i>                         | the <b>controller</b> and <b>target</b> use the same clock, there is no need for clock synchronization |
| Wires        | <i>SDA / SCL</i>                            | the same read and write wire and a clock wire  |
| Devices      | <i>1 controller<br/>several<br/>targets</i> | a receiver and a transmitter   |
| Speed        | <i>5 Mbit/s</i>                             | usually 100 Kbit/s, 400 Kbit/s and 1 Mbit/s  |



- [illegible]



# Embassy API

for RP2040, synchronous

```
pub struct Config {  
    /// Frequency.  
    pub frequency: u32,  
}
```

```
pub enum ConfigError {  
    /// Max i2c speed is 1MHz  
    FrequencyTooHigh,  
    ClockTooSlow,  
    ClockTooFast,  
}
```

```
pub enum Error {  
    Abort(AbortReason),  
    InvalidReadBufferLength,  
    InvalidWriteBufferLength,  
    AddressOutOfRange(u16),  
    AddressReserved(u16),  
}
```

```
1  use embassy_rp::i2c::Config as I2cConfig;  
2  
3  let sda = p.PIN_14;  
4  let scl = p.PIN_15;  
5  
6  let mut i2c = i2c::I2c::new_blocking(p.I2C1, scl, sda, I2cConfig::default());  
7  
8  let tx_buf = [0x90];  
9  i2c.write(0x5e, &tx_buf).unwrap();  
10  
11 let mut rx_buf = [0x00u8; 7];  
12 i2c.read(0x5e, &mut rx_buf).unwrap();
```



# Embassy API

for RP2040, asynchronous

```
1  use embassy_rp::i2c::Config as I2cConfig;
2
3  bind_interrupts!(struct Irqs {
4      I2C1_IRQ => InterruptHandler<I2C1>;
5  });
6
7  let sda = p.PIN_14;
8  let scl = p.PIN_15;
9
10 let mut i2c = i2c::I2c::new_async(p.I2C1, scl, sda, Irqs, I2cConfig::default());
11
12 let tx_buf = [0x90];
13 i2c.write(0x5e, &tx_buf).await.unwrap();
14
15 let mut rx_buf = [0x00u8; 7];
16 i2c.read(0x5e, &mut rx_buf).await.unwrap();
```



# USB 2.0

Universal Serial Bus



# Universal Serial Bus

2.0

- Used for communication between a host and several devices that each provide functions
- Two modes:
  - *host* - initiates the communication (usually a computer)
  - *device* - receives and transmits data when the *host* requests it
- each device has a 7 bit address assigned upon connect
  - maximum 127 devices connected to a USB host
- devices are interconnected using *hubs*
- USB devices tree





# Bibliography

for this section

## 1. **Raspberry Pi Ltd**, *RP2040 Datasheet*

- Chapter 4 - *Peripherals*
  - Chapter 4.1 - *USB*

## 2. *USB Made Simple*





# USB Device

- can work as **host** or **device**, but not at the same time
- uses a differential line for transmission
- uses a 48 MHz clock
- maximum 16 endpoints (buffers)
  - *IN* - from **device** to **host**
  - *OUT* - from **host** to **device**
- endpoints 0 IN and OUT are used for control





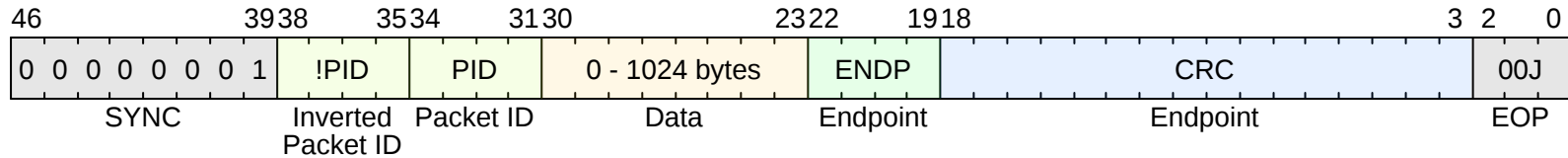
# USB Packet

the smallest element of data transmission

## Token



## Data



## Handshake





# Token Packet

usually asks for a data transmission

| Type         | PID  | Description  |
|--------------|------|--|
| <i>OUT</i>   | 0001 | <b>host</b> wants to transmit data to the <b>device</b>  |
| <i>IN</i>    | 1001 | <b>host</b> wants to receive data from the <b>device</b> |
| <i>SETUP</i> | 1101 | <b>host</b> wants to setup the <b>device</b>             |

Address: ADDR : ENDP





# Data Packet

transmits data

| Type         | PID  | Description   |
|--------------|------|---|
| <i>DATA0</i> | 0011 | the data packet is the first one or follows after a <i>DATA1</i> packet |
| <i>DATA1</i> | 1011 | the data packet follows after a <i>DATA0</i> packet                     |

Data can be between 0 and 1024 bytes





# Handshake Packet

acknowledges data

| Type         | PID  | Description   |
|--------------|------|---|
| <i>ACK</i>   | 0010 | data has been <b>successfully received</b>            |
| <i>NACK</i>  | 1010 | data has <b>not</b> been <b>successfully received</b> |
| <i>STALL</i> | 1110 | the device has an <b>error</b>                        |





# Transmission Modes

- *Control* - used for configuration
- *Isochronous* - used for high bandwidth, best effort
- *Bulk* - used for low bandwidth, stream
- *Interrupt* - used for low bandwidth, guaranteed latency



# Control

used to control a device - ask for data

**Setup** - send a command (*GET\_DESCRIPTOR*,...)



...



**Status** - report the status to the host

**Data** - optional several transfers, host transfers data

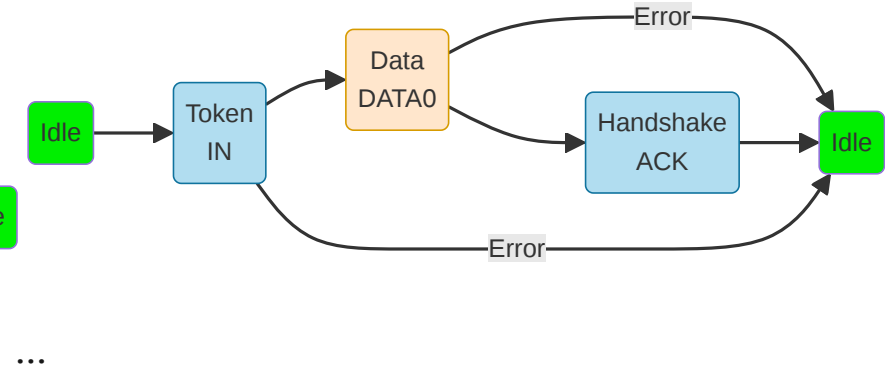
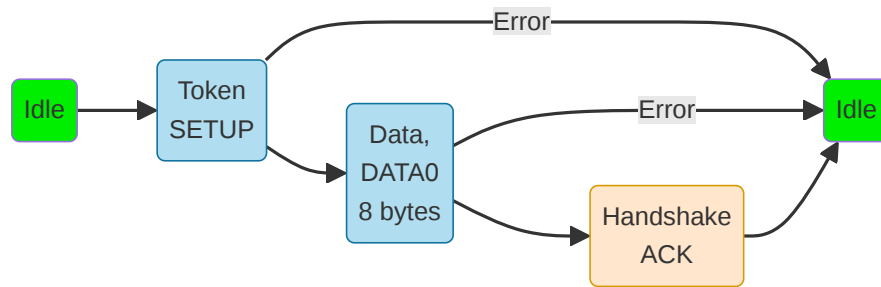




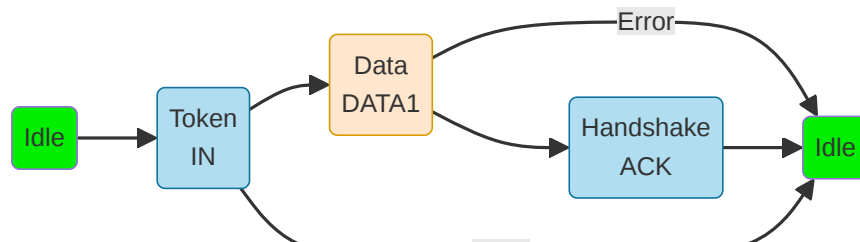
# Control

used to control a device - send data

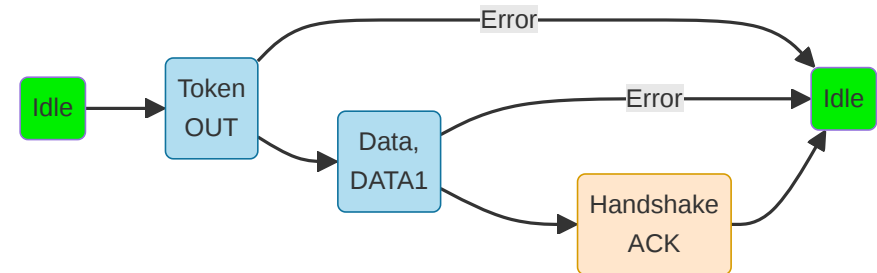
**Setup** - send a command (*SET\_ADDRESS*,...)



**Data** - *optional* several transfers, device transfers the requested data



**Status** - report the status to the device







# Isochronous

fast but not reliable transfer

- has a guaranteed bandwidth
- allows data loss
- used for functions like streaming where losing a packet has a minimal impact

**OUT** - transfer data from the host to the device



**IN** - transfer data from the device to the host





# Bulk

slow, but reliable transfer

- does not have a guaranteed bandwidth
- secure transfer
- used for large data transfers where losing packets is not permitted

**OUT** - transfer data from the host to the device



**IN** - transfer data from the device to the host





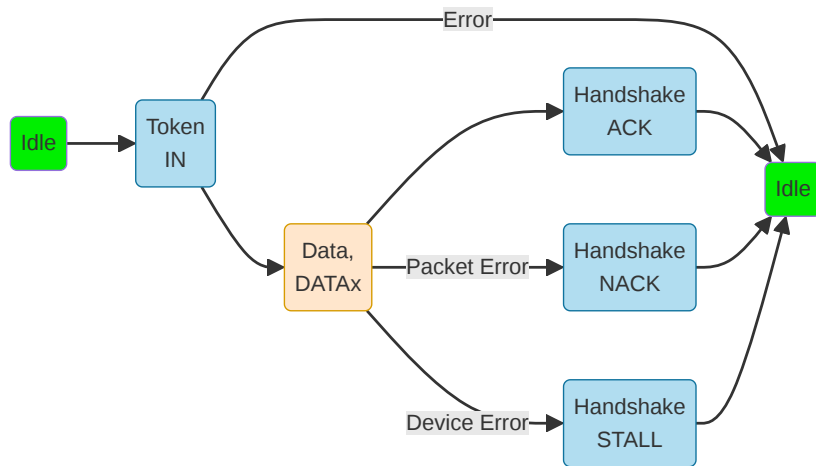
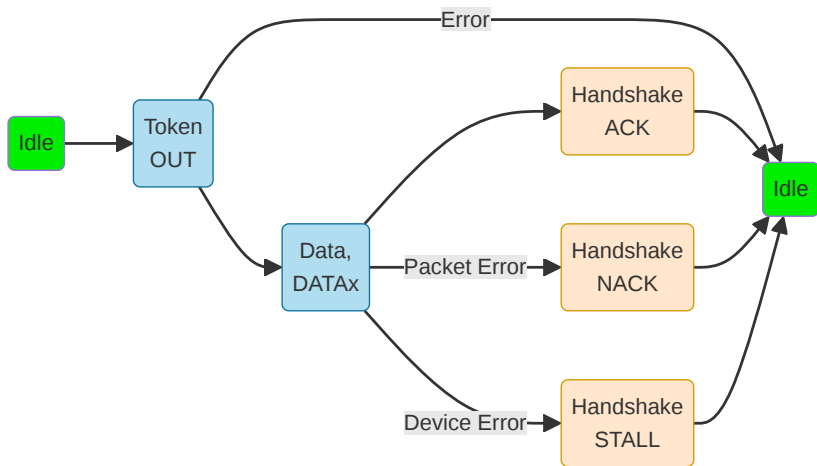
# Interrupt

transfer data at a minimum time interval

- the endpoint descriptor asks the host start an interrupt transfer at a time interval
- used for sending and receiving data at certain intervals

**OUT** - transfer data from the host to the device

**IN** - transfer data from the device to the host





# Device Organization

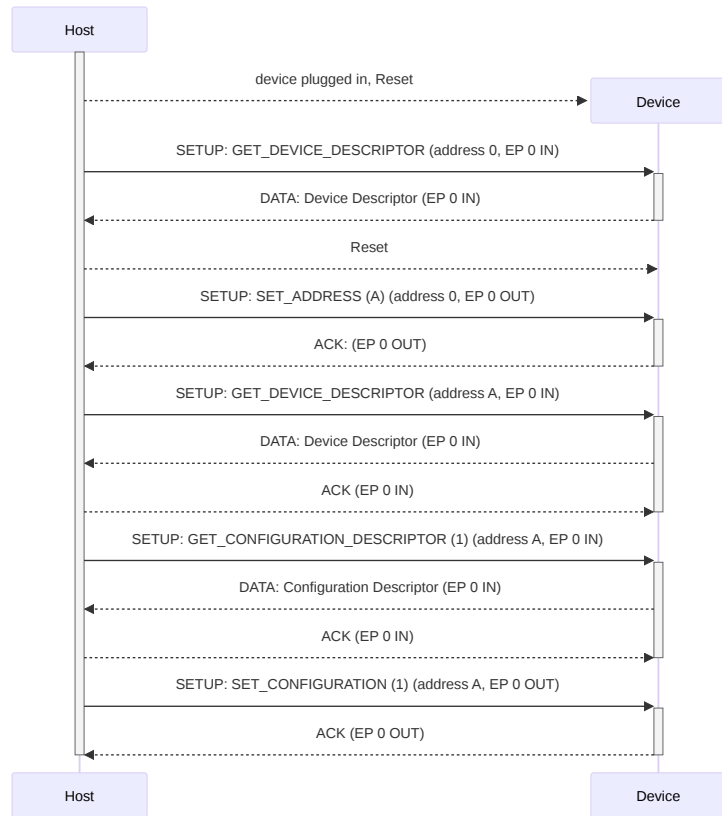
configuration, interfaces, endpoints

- a device can have multiple configurations
  - for instance different functionality based on power consumption
- a configuration has multiple interfaces
  - a device can perform multiple functions
  - Debugger
  - Serial Port
- each interface has multiple endpoints attached
  - endpoints are used for data transfer
  - maximum 16 endpoints, can be configured IN and OUT
- the device reports the descriptors in this order





# Connection



# Token SETUP Packet

The DATA packet of the SETUP Control Transfer



## *bmRequestType* field





# USB 1.0 and 2.0 Modes

| Mode       | Speed      | Version |
|------------|------------|---------|
| Low Speed  | 1.5 Mbit/s | 1.0     |
| Full Speed | 12 Mbit/s  | 1.0     |
| High Speed | 480 Mbit/s | 2.0     |



# Facts

|              |                                   |   |
|--------------|-----------------------------------|---|
| Transmission | <i>half duplex</i>                | data must be sent in one direction at one time                      |
| Clock        | <i>independent</i>                | the <b>host</b> and the <b>device</b> must synchronize their clocks |
| Wires        | <i>DP / DM</i>                    | data is sent in a differential way                                  |
| Devices      | <i>1 host<br/>several devices</i> | a receiver and a transmitter  |
| Speed        | <i>480 MBit/s</i>                 |   |



# Embassy API

for RP2040, setup the device

```
use embassy_rp::usb::{Driver, Instance, InterruptHandler};
use embassy_usb::class::cdc_acm::{CdcAcmClass, State};

bind_interrupts!(struct Irqs {
    USBCTRL_IRQ => InterruptHandler<USB>;
});

let driver = Driver::new(p.USB, Irqs);

let mut config = Config::new(0xc0de, 0xcafe);
config.manufacturer = Some("Embassy");
config.product = Some("USB-serial example");
config.serial_number = Some("12345678");
config.max_power = 100;
config.max_packet_size_0 = 64;

// Required for windows compatibility.
config.device_class = 0xEF;
config.device_sub_class = 0x02;
config.device_protocol = 0x01;
config.composite_with_iads = true;
```

```
// It needs some buffers for building the descriptors.
let mut config_descriptor = [0; 256];
let mut bos_descriptor = [0; 256];
let mut control_buf = [0; 64];

let mut state = State::new();

let mut builder = Builder::new(
    driver,
    config,
    &mut config_descriptor,
    &mut bos_descriptor,
    &mut [], // no msos descriptors
    &mut control_buf,
);

// Create classes on the builder.
let mut class = CdcAcmClass::new(&mut builder, &mut state, 64);

// Build the builder.
let mut usb = builder.build();

// Run the USB device.
let usb_driver = usb.run();
```





# Embassy API

for RP2040, use the USB device

```
1  let echo_loop = async {  
2      loop {  
3          class.wait_connection().await;  
4          info!("Connected");  
5          let _ = echo(&mut class).await;  
6          info!("Disconnected");  
7      }  
8  };  
9  
10 // Run everything concurrently.  
11 join(usb_driver, echo_loop).await;
```

```
1  async fn echo<'d, T: Instance + 'd>(class: &mut CdcAcmClass<'d, Driver<'d, T>>) -> Result<(), EndpointError> {  
2      let mut buf = [0; 64];  
3      loop {  
4          let n = class.read_packet(&mut buf).await?;  
5          let data = &buf[..n];  
6          info!("data: {:x}", data);  
7          class.write_packet(data).await?;  
8      }  
9  }
```



# 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 5.2 - *I2C 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      |
|            | do not write       | read only        | read / write      | read only      | read only        | read only | write only |

Datasheet



# Reading from a digital sensor

using synchronous/asynchronous I2C to read the `press_lsb` register of BMP280

```
1  const DEVICE_ADDR: u8 = 0x77;
2  const REG_ADDR: u8 = 0xf8;
3
4  i2c.write(DEVICE_ADDR, &[REG_ADDR]).unwrap();
5
6  let mut buf = [0x00u8];
7  i2c.read(DEVICE_ADDR, &mut buf).unwrap();
8
9  // use the value
10 let pressure_lsb = buf[1];
```

```
1  const DEVICE_ADDR: u8 = 0x77;
2  const REG_ADDR: u8 = 0xf8;
3
4  i2c.write(DEVICE_ADDR, &[REG_ADDR]).await.unwrap();
5
6  let mut buf = [0x00u8];
7  i2c.read(DEVICE_ADDR, &mut buf).await.unwrap();
8
9  // use the value
10 let pressure_lsb = buf[1];
```



# Writing to a digital sensor

using synchronous/asynchronous I2C to set up the `ctrl_meas` register of the BMP280 sensor

```
1  const DEVICE_ADDR: u8 = 0x77;
2  const REG_ADDR: u8 = 0xf4;
3
4  // see subchapters 3.3.2, 3.3.1 and 3.6
5  let value = 0b100_010_11;
6
7  i2c.write(DEVICE_ADDR, &[REG_ADDR]);
8
9  let buf = [REG_ADDR, value];
10 i2c.write(DEVICE_ADDR, &buf).unwrap();
```

```
1  const DEVICE_ADDR: u8 = 0x77;
2  const REG_ADDR: u8 = 0xf4;
3
4  // see subchapters 3.3.2, 3.3.1 and 3.6
5  let value = 0b100_010_11;
6
7  i2c.write(DEVICE_ADDR, &[REG_ADDR]);
8
9  let buf = [REG_ADDR, value];
10 i2c.write(DEVICE_ADDR, &buf).await.unwrap();
```





# Conclusion

we discussed about

- Buses
  - Inter-Integrated Circuit
  - Universal Serial Bus v2.0