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

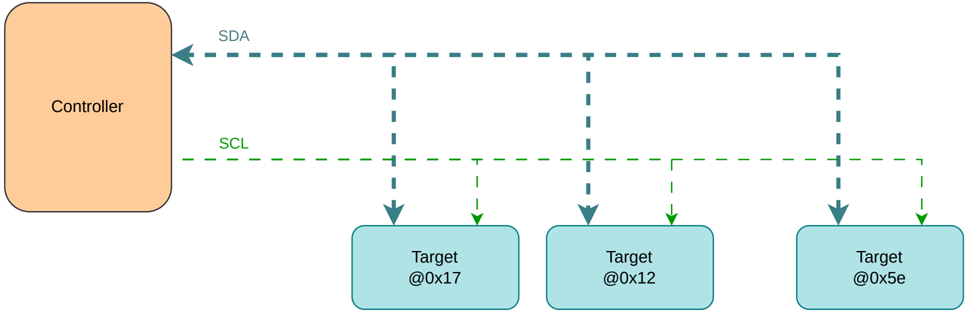
for this section

1. **Raspberry Pi Ltd**, *RP2350 Datasheet*
    - Chapter 12 - *Peripherals*
        - Chapter 12.2 - *I2C*
2. **Paul Denisowski**, *Understanding I2C*

I2C

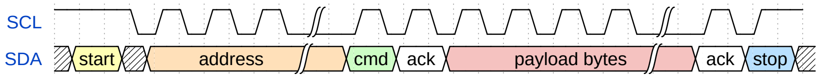
a.k.a *I square C*

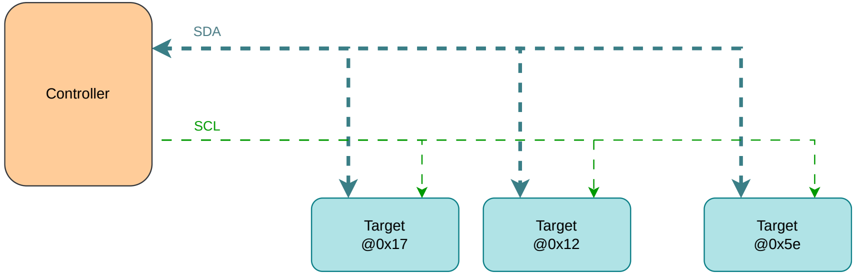
- Used for communication between integrated circuits
- Sensors usually expose an *SPI* and an *I2C* interface
- Two device types:
  - *controller* (master) - initiates the communication (usually MCU)
  - *target* (slave) - receive and transmit data when the *controller* requests (usually the sensor)



Wires & Addresses

- *SDA* - **S**erial **DA**ta line - carries data from the **controller** to the **target** or from the **target** to the **controller**
- *SCL* - **S**erial **CL**ock 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`



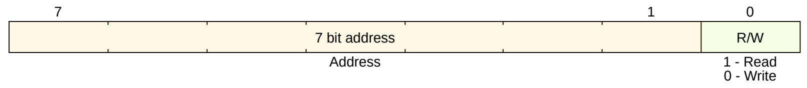


Transmission Example

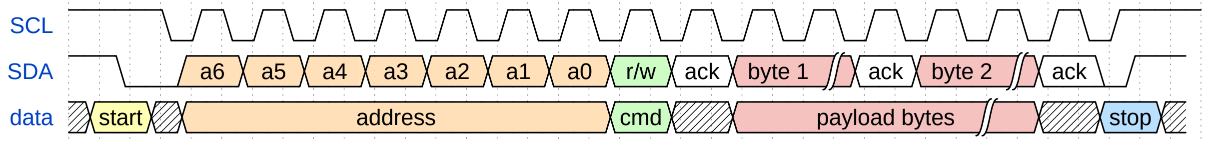
7 bit address

1. **controller** issues a `START` condition
   - pulls the `SDA` line `LOW`
   - waits for ~ 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

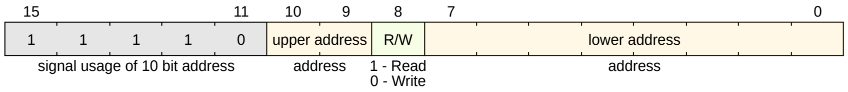


Transmission Example

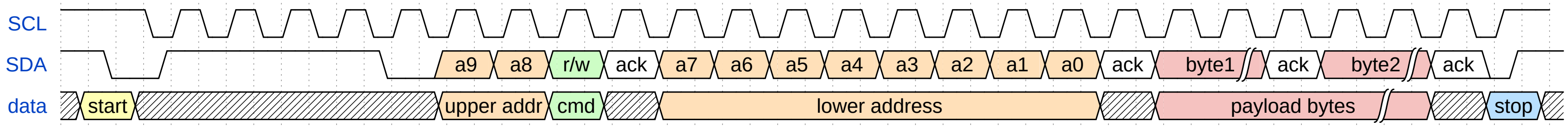
10 bit address

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

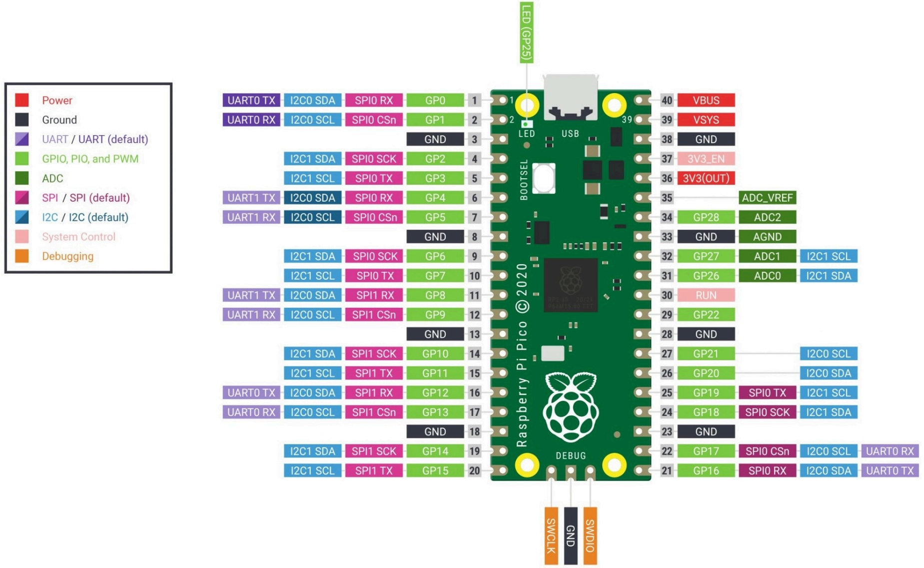
Mode	Speed	Capacity	Drive	Direction
Standard mode (Sm)	100 kbit/s	400 pF	Open drain	Bidirectional
Fast mode (Fm)	400 kbit/s	400 pF	Open drain	Bidirectional
Fast mode plus (Fm+)	1 Mbit/s	550 pF	Open drain	Bidirectional
High-speed mode (Hs)	1.7 Mbit/s	400 pF	Open drain	Bidirectional
High-speed mode (Hs)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

Facts

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

Usage

- sensors
- small displays
- RP2350 has two I2C devices



Embassy API

for RP2350, synchronous

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

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

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

```rust
use embassy_rp::i2c::Config as I2cConfig;

let sda = p.PIN_14;
let scl = p.PIN_15;
let mut i2c = i2c::I2c::new_blocking(p.I2C1, scl, sda, I2cConfig::default());

let tx_buf = [0x90];
i2c.write(0x5e, &tx_buf).unwrap();

let mut rx_buf = [0x00u8; 7];
i2c.read(0x5e, &mut rx_buf).unwrap();

i2c.write_read(0x5e, &tx_buf, &mut rx_buf).unwrap();
```

Embassy API

for RP2350, asynchronous

```rust
use embassy_rp::i2c::Config as I2cConfig;

bind_interrupts!(struct Irqs {
    I2C1_IRQ => InterruptHandler<I2C1>;
});

let sda = p.PIN_14;
let scl = p.PIN_15;
let mut i2c = i2c::I2c::new_async(p.I2C1, scl, sda, Irqs, I2cConfig::default());

let tx_buf = [0x90];
i2c.write(0x5e, &tx_buf).await.unwrap();

let mut rx_buf = [0x00u8; 7];
i2c.read(0x5e, &mut rx_buf).await.unwrap();

i2c.write_read(0x5e, &tx_buf, &mut rx_buf).await.unwrap();
```

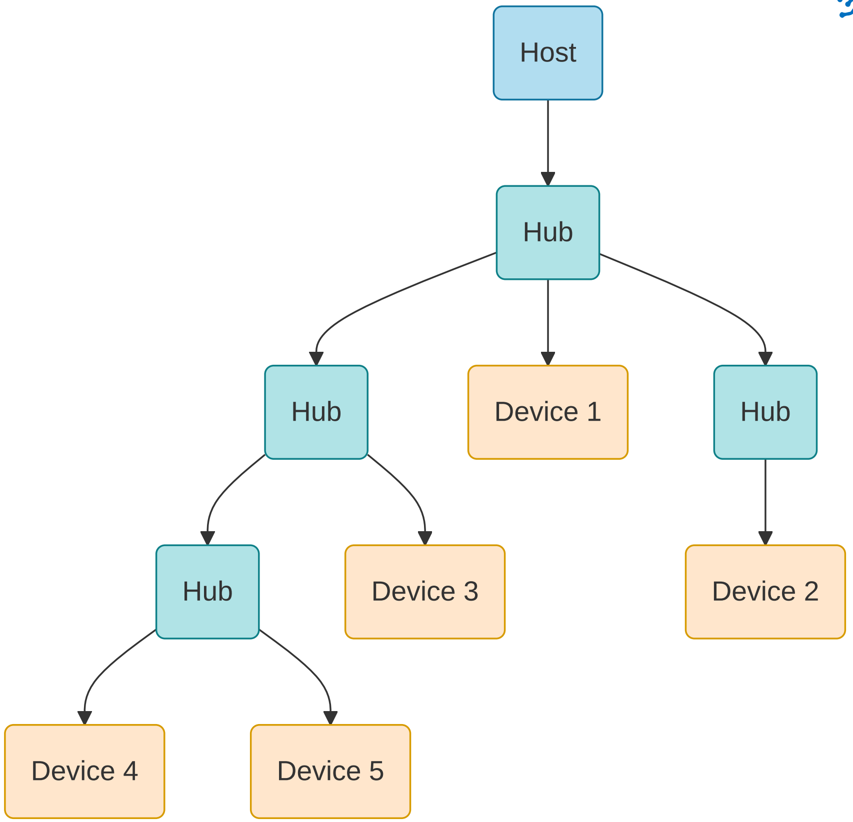
USB 2.0

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



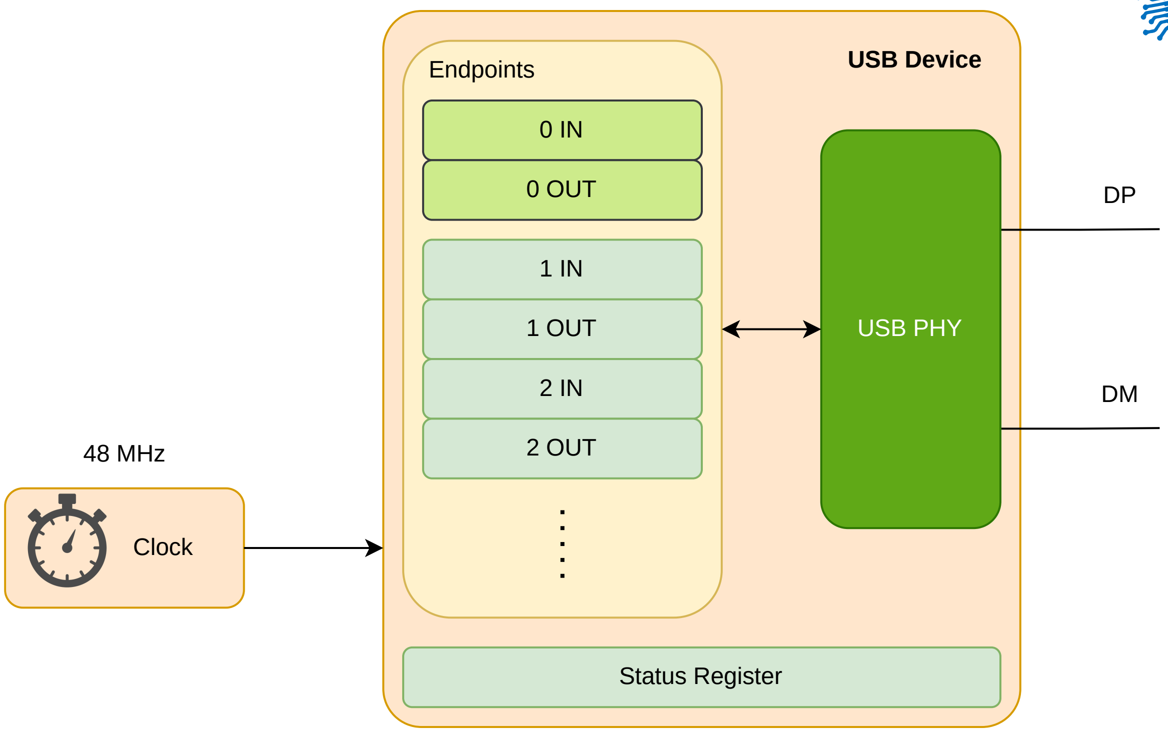
Bibliography

for this section

1. **Raspberry Pi Ltd**, *RP2350 Datasheet*
   - Chapter 12 - *Peripherals*
     - Chapter 12.7 - *USB*
2. *USB Made Simple*

USB

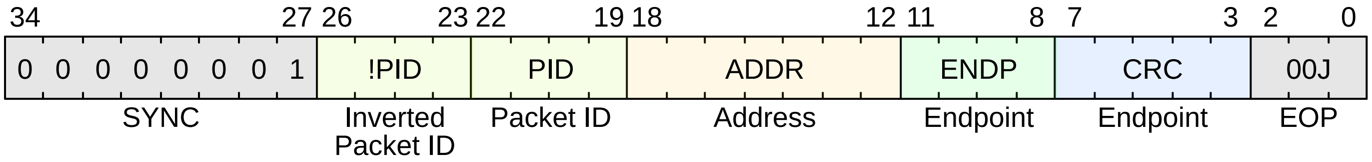
- can work as **host** or **device**, but not at the same time
- uses a differential line for transmission
- uses a 48 MHz clock
- maximum 16 endpoints (buffers)
  - *IN* - from **device** to **host**
  - *OUT* - from **host** to **device**
- endpoints 0 IN and OUT are used for control



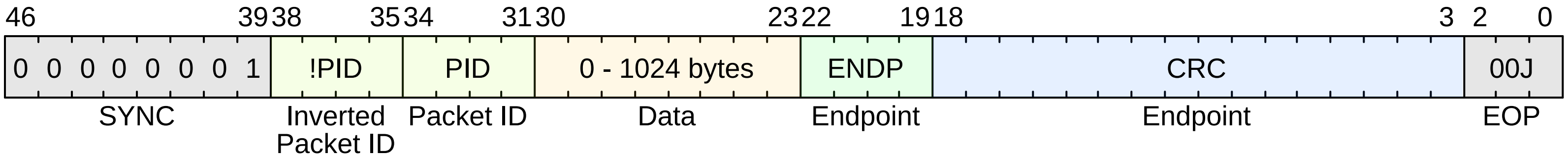
USB Packet

the smallest element of data transmission

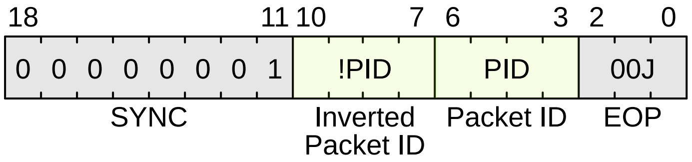
Token



Data



Handshake

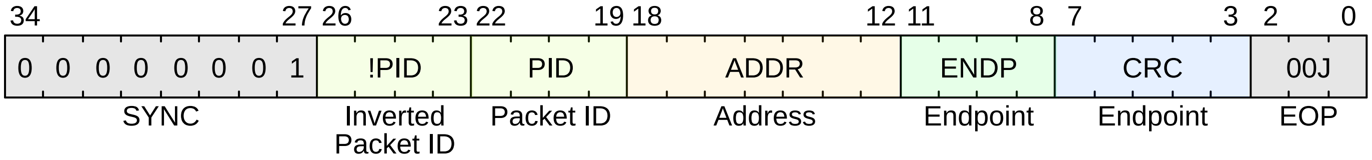


Token Packet

usually asks for a data transmission

Type	PID	Description
OUT	`0001`	**host** wants to transmit data to the **device**
IN	`1001`	**host** wants to receive data from the **device**
SETUP	`1101`	**host** wants to setup the **device**

Address: `ADDR` : `ENDP`

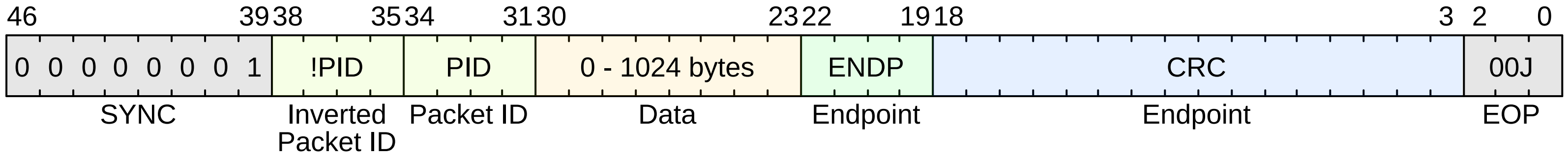


Data Packet

transmits data

Type	PID	Description
DATA0	`0011`	the data packet is the first one or follows after a `DATA1` packet
DATA1	`1011`	the data packet follows after a `DATA0` packet

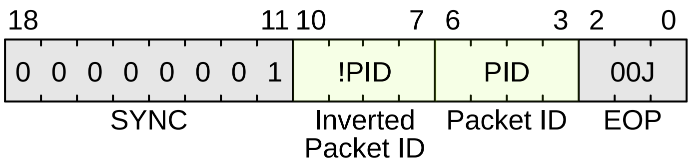
Data can be between 0 and 1024 bytes



Handshake Packet

acknowledges data

Type	PID	Description
ACK	`0010`	data has been **successfully received**
NACK	`1010`	data has **not** been **successfully received**
STALL	`1110`	the device has an **error**



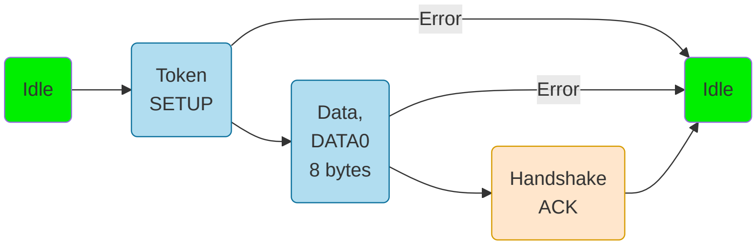
Transmission Modes

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

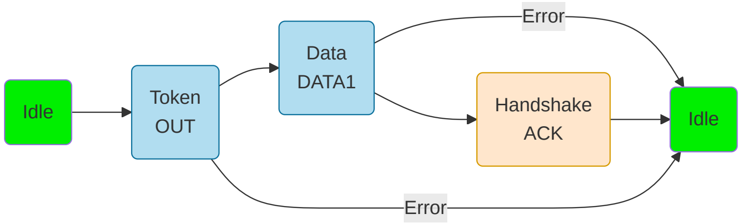
Control

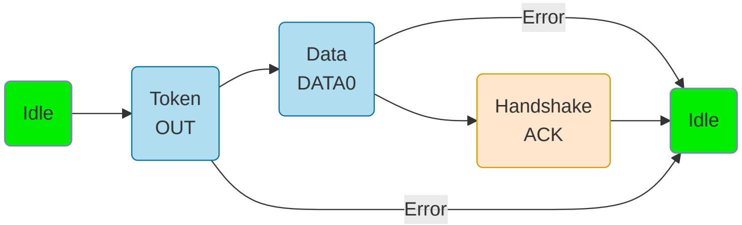
used to control a device - ask for data

Setup - send a command (*GET_DESCRIPTOR, ...*)



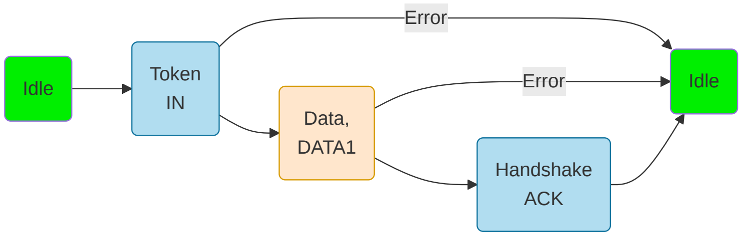
Data - *optional* several transfers, host transfers data





...

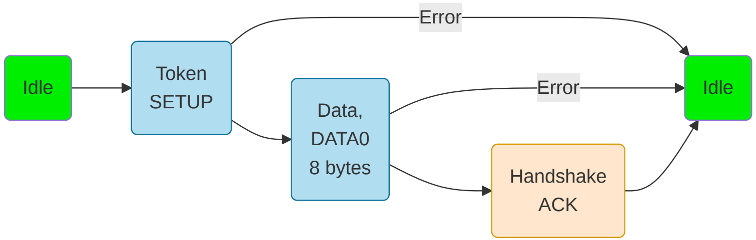
Status - report the status to the host



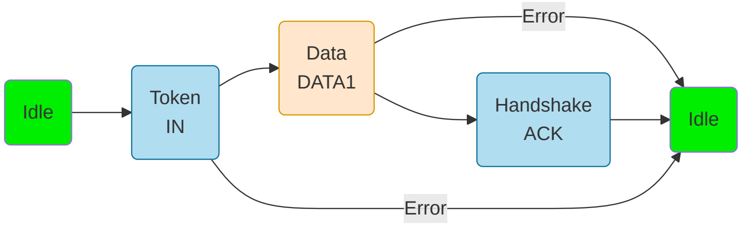
Control

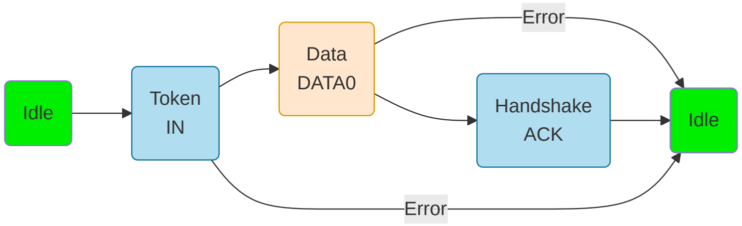
used to control a device - send data

Setup - send a command (*SET_ADDRESS*, ...)



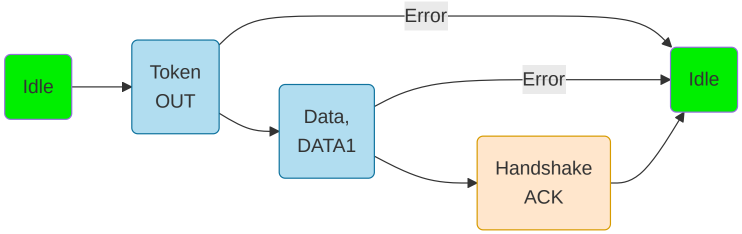
Data - *optional* several transfers, device transfers the requested data





...

Status - report the status to the device

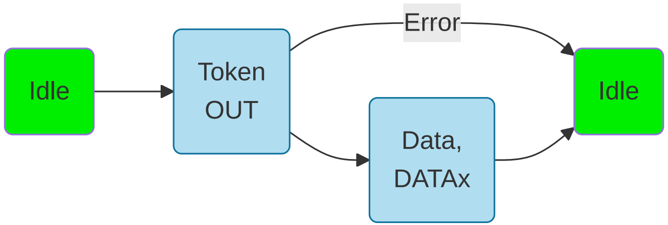


Isochronous

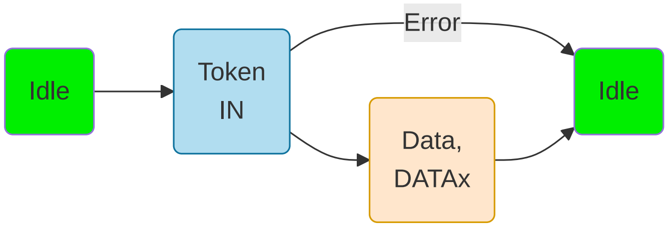
fast but not reliable transfer

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

OUT - transfer data from the host to the device



IN - transfer data from the device to the host



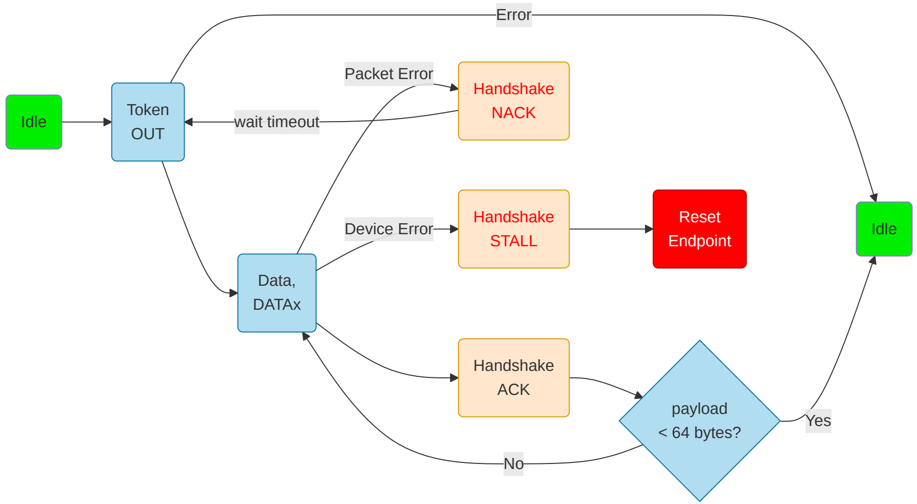
Bulk

slow, but reliable transfer

- does not have a guaranteed bandwidth
- does not allow data loss
- used for large data transfers where loosing packets is not permitted

OUT - transfer data from the host to the device

IN - transfer data from the device to the host

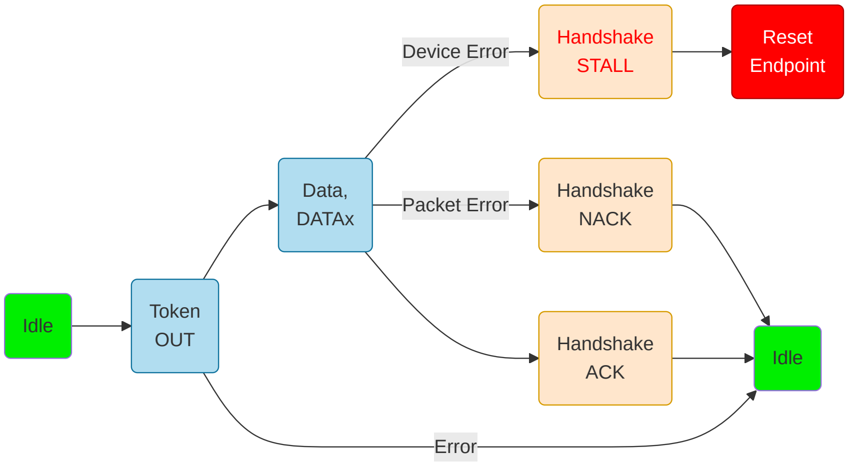


Interrupt

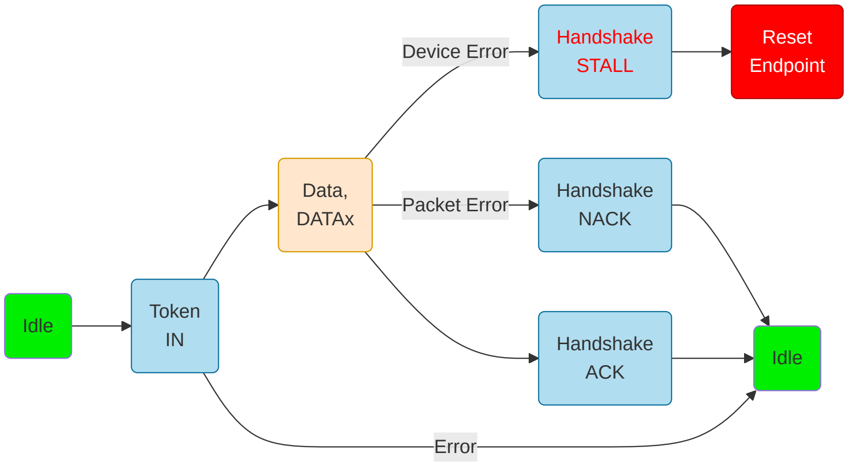
transfer data at a minimum time interval

- 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



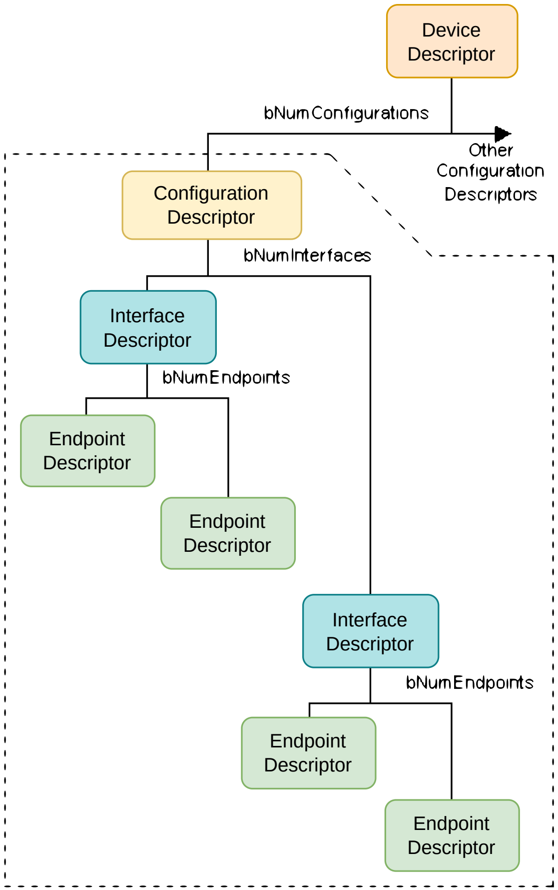
IN - transfer data from the device to the host



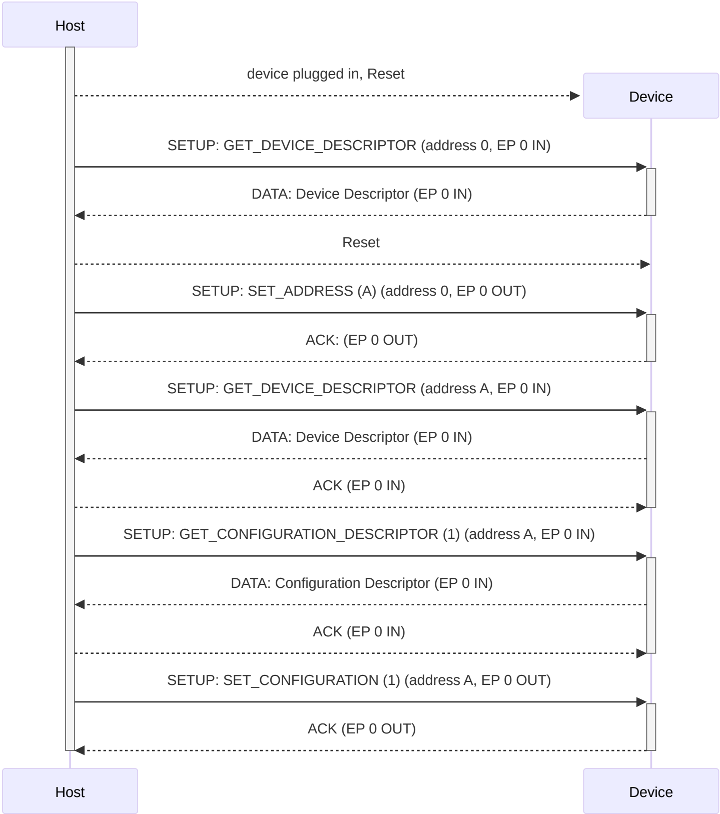
Device Organization

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 *alternate settings* with 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

63 … 56	55 … 48	47 … 32
bmRequestType	bRequest	wValue
	Specific to request	Value

31 … 16	15 … 0
wIndex	wLength
Index of offset	Number of bytes to transfer in DATA step

bmRequestType field

7	6 … 5	4 … 0
Direction	Type	Recipient
0 - Host to Device 1 - Device to Host	00 - Standard 01 - Class 10 - Vendor 11 - Reserved	00000 - Device 00001 - Interface 00010 - Endpoint 00011 - Other 00100 - 11111 - Reserved

Device Classes

predefined devices types

Device Class Code	Class Name	Description
`0x00`	**Device Class**	Device class-specific; the class code is assigned by the device.
`0x01`	**Audio**	Audio devices (e.g., audio interfaces, speakers, microphones).
`0x02`	**Communications and CDC Control**	Devices related to communication (e.g., modems, network adapters).
`0x03`	**HID (Human Interface Device)**	Devices like keyboards, mice, and other human interface devices.
`0x05`	**Physical Interface Device (PID)**	Devices that require physical input/output (e.g., game controllers).
`0x06`	**Image**	Image devices such as digital cameras and scanners.
`0x07`	**Printer**	Devices for printing (e.g., printers).
`0x08`	**Mass Storage**	Mass storage devices (e.g., USB flash drives, external hard drives).
`0x0A`	**Still Image Capture Device**	Devices for still image capture (e.g., digital cameras).
`0x0B`	**Smart Card**	Smart card readers and related devices.
`0x0D`	**Content Security**	Devices for content protection (e.g., video players).
`0x0E`	**Video**	Video devices (e.g., webcams, video capture devices).
`0x0F`	**Personal Healthcare**	Healthcare devices (e.g., thermometers, blood pressure monitors).
`0x10`	**Audio/Video**	Devices with combined audio/video functions.
`0x11`	**Health Device**	Devices used in health-related monitoring.
`0x12`	**Diagnostic Device**	Devices for diagnostics or test instruments.
`0xFF`	**Vendor Specific**	Vendor-specific devices (class code not assigned by USB standard).

Device Descriptor

describes the whole device

Field	Value	Description
bLength	`18`	Descriptor length in bytes.
bDescriptorType	`1`	Descriptor type (1 = Device Descriptor).
bcdUSB	`0x0200`	USB specification release number (2.0).
bDeviceClass	`0xFF`	Device class (0xFF = Vendor Specific).
bDeviceSubClass	`0`	Device subclass (0 = defined by the interface).
bDeviceProtocol	`0`	Device protocol (0 = defined by the interface).
bMaxPacketSize0	`64`	Maximum packet size for endpoint 0 (64 bytes).
idVendor	`0xC0DE`	Vendor ID (example: `0xC0DE`).
idProduct	`0xCAFE`	Product ID (example: `0xCAFE`).
bcdDevice	`0x0100`	Device release number (example: `1.0`).
iManufacturer	`1`	Index of the string descriptor for the manufacturer.
iProduct	`2`	Index of the string descriptor for the product.
iSerialNumber	`3`	Index of the string descriptor for the serial number.
bNumConfigurations	`1`	Number of configurations supported by the device.

Configuration Descriptor

one of the configurations

Field	Value	Description
bLength	`9`	Descriptor length in bytes (always 9 for configuration descriptor).
bDescriptorType	`2`	Descriptor type (2 = Configuration Descriptor).
wTotalLength	`0x0022`	Total length of data returned for this configuration (including all descriptors).
bNumInterfaces	`1`	Number of interfaces supported by this configuration.
bConfigurationValue	`1`	Value to select this configuration.
iConfiguration	`4`	Index of the string descriptor describing the configuration.
bmAttributes	`0x80`	Configuration characteristics (bus-powered, no remote wake-up).
bMaxPower	`50`	Maximum power consumption (in 2mA units, so `50` means 100mA).

Interface Descriptor

Field	**Value**	**Description**
bLength	`9`	Descriptor length in bytes (always 9 for interface descriptor).
bDescriptorType	`4`	Descriptor type (4 = Interface Descriptor).
bInterfaceNumber	`0`	Number of this interface (starting from 0).
bAlternateSetting	`0`	Alternate setting (0 = default setting).
bNumEndpoints	`1`	Number of endpoints used by this interface.
bInterfaceClass	`0xFF`	Interface class (0xFF = Vendor Specific).
bInterfaceSubClass	`0`	Interface subclass (0 = vendor specific).
bInterfaceProtocol	`0`	Interface protocol (0 = vendor specific).
iInterface	`5`	Index of the string descriptor describing this interface.

Endpoint Descriptor

Field	Value	Description
bLength	`7`	Descriptor length in bytes (always 7 for endpoint descriptor).
bDescriptorType	`5`	Descriptor type (5 = Endpoint Descriptor).
bEndpointAddress	`0xb1_0000_001`	Endpoint address (`0x81`): **Bit 7** indicates **IN** direction (device to host), and **Bits 0-3** indicate the endpoint number (`1` in this case).
bmAttributes	`0x02`	Endpoint attributes (`0x02` = Bulk endpoint).
wMaxPacketSize	`64`	Maximum packet size the endpoint can handle (64 bytes).
bInterval	`0`	Interval for polling (relevant for interrupt endpoints; `0` for others).

Strings Descriptor

String Descriptor for Configuration and Interface

Field	Value	Description
bLength	`4`	Descriptor length in bytes (always 4 for string descriptor header).
bDescriptorType	`3`	Descriptor type (3 = String Descriptor).
bString	`0x09 0x55 0x53 0x42 0x20 0x43 0x6F 0x6E 0x66 0x69 0x67 0x20 0x31`	UTF-16LE string encoding: `"USB Config 1"`.

Explanation: This string descriptor corresponds to **Configuration 1**. The string is encoded in **UTF-16LE** (little-endian). Each character is represented by two bytes.

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	*half duplex*	data must be sent in one direction at one time
Clock	*independent*	the **host** and the **device** must synchronize their clocks
Wires	*DP / DM*	data is sent in a differential way
Devices	*1 host* *several devices*	a receiver and a transmitter
Speed	*480 MBbit/s*	

Embassy API

for RP2350, setup the device

```rust
use embassy_rp::usb::{Driver, InterruptHandler};
use embassy_usb::Config;

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

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

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

Embassy API

for RP2350, setup the descriptors

```rust
use embassy_usb::msos::{self, windows_version};
use embassy_usb::Builder;

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

let mut builder = Builder::new(driver, config,
  &mut config_descriptor, &mut bos_descriptor, &mut msos_descriptor, &mut control_buf,
);

// Required for Windows
const DEVICE_INTERFACE_GUIDS: &[&str] = &["{AFB9A6FB-30BA-44BC-9232-806CFC875321}"];
builder.msos_descriptor(windows_version::WIN8_1, 0);
builder.msos_feature(msos::CompatibleIdFeatureDescriptor::new("WINUSB", ""));
builder.msos_feature(msos::RegistryPropertyFeatureDescriptor::new(
  "DeviceInterfaceGUIDs",
  msos::PropertyData::RegMultiSz(DEVICE_INTERFACE_GUIDS),
));
```

Embassy API

for RP2350, setup the device's function and start

```rust
// Add a vendor-specific function (class 0xFF), and corresponding interface,
// that uses our custom handler.
let mut function = builder.function(0xFF, 0, 0);
let mut interface = function.interface();
let mut alt = interface.alt_setting(0xFF, 0, 0, None);
let mut read_ep = alt.endpoint_bulk_out(64);
let mut write_ep = alt.endpoint_bulk_in(64);
drop(function);

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

// Create the USB device handler
let usb_run = usb.run();
```

Embassy API

for RP2350, use the USB device

```rust
let echo_run = async {
  loop {
    read_ep.wait_enabled().await;
    info!("Connected");
    loop {
      let mut data = [0; 64];
      match read_ep.read(&mut data).await {
        Ok(n) => {
          info!("Got bulk: {:a}", data[..n]);
          // Echo back to the host:
          write_ep.write(&data[..n]).await.ok();
        }
        Err(_) => break,
      }
    }
    info!("Disconnected");
  }
};

// Run everything concurrently.
// If we had made everything `'static` above instead, we could do this using separate tasks instead.
join(usb_run, echo_run).await;
```

Host API

using `nusb`

```rust
use nusb::transfer::RequestBuffer;

const BULK_OUT_EP: u8 = 0x01;
const BULK_IN_EP: u8 = 0x81;

async fn main() {
    let di = nusb::list_devices()
        .unwrap()
        .find(|d| d.vendor_id() == 0xc0de && d.product_id() == 0xcafe)
        .expect("no device found");

    let device = di.open().expect("error opening device");
    let interface = device.claim_interface(0).expect("error claiming interface");

    let result = interface.bulk_out(BULK_OUT_EP, b"hello world".into()).await;
    println!("{result:?}");

    let result = interface.bulk_in(BULK_IN_EP, RequestBuffer::new(64)).await;
    println!("{result:?}");
}
```

Host API

using Python

```python
import usb
import time

# Find the USB device
dev = usb.core.find(idVendor=0xc0de, idProduct=0xcafe)
if dev is None:
    raise ValueError('Device not found')

dev.set_configuration() # Set the active configuration (this is usually required after device detection)

OUT_ENDPOINT = 0x01  # Usually 0x01 for OUT endpoint
IN_ENDPOINT = 0x81  # Usually 0x81 for IN endpoint (Endpoint 1, Direction IN)

data_to_send = b"Hello, USB Device!"

dev.write(OUT_ENDPOINT, data_to_send)
time.sleep(1) # Wait for a short time to ensure data is transferred

data_received = dev.read(IN_ENDPOINT, 64)  # Read 64 bytes (adjust the size if needed)
print("Data received from device:", bytes(data_received))

usb.util.release_interface(dev, 0) # Release the device interface (optional, but good practice)
```

Sensors

Analog and Digital Sensors

Bibliography

for this section

BOSCH, *BMP280 Digital Pressure Sensor*

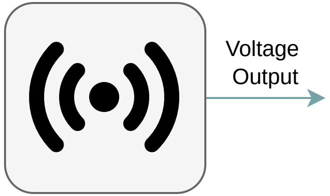
- Chapter 3 - *Functional Description*
- Chapter 4 - *Global memory map and register description*
- Chapter 5 - *Digital Interfaces*
    - Subchapter 5.2 - *I2C Interface*

Sensors

analog and digital

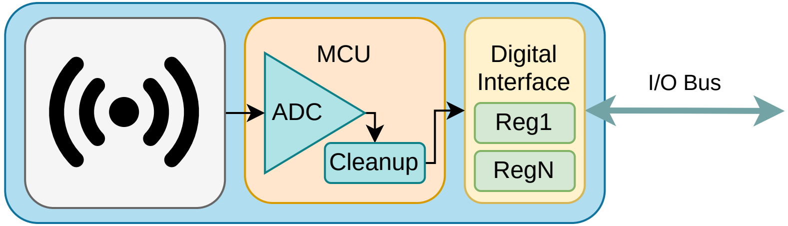
Analog

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



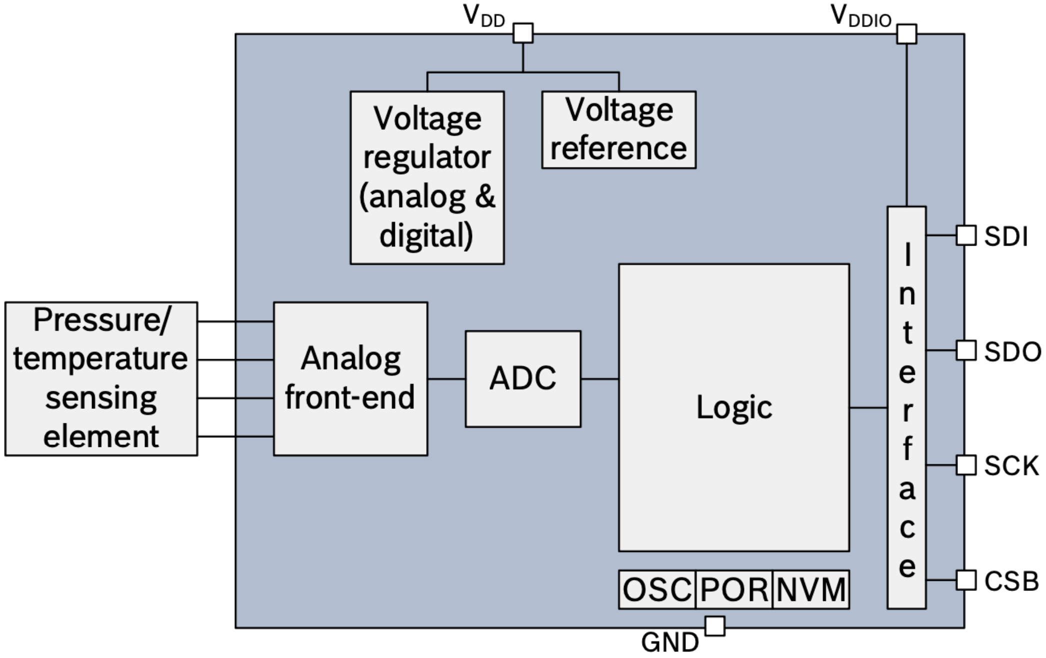
Digital

- 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
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 I2C to read the `press_lsb` register of BMP280

```rust
const DEVICE_ADDR: u8 =  0x77;
const REG_ADDR: u8 = 0xf8;

let mut buf = [0x00u8];

i2c.write_read(
  DEVICE_ADDR, &[REG_ADDR], &mut buf
).unwrap();

// use the value
let pressure_lsb = buf[1];
```

```rust
const DEVICE_ADDR: u8 =  0x77;
const REG_ADDR: u8 = 0xf8;

let mut buf = [0x00u8];

i2c.write_read(
  DEVICE_ADDR, &[REG_ADDR], &mut buf
).await.unwrap();

// use the value
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

```rust
const DEVICE_ADDR: u8 =  0x77;
const REG_ADDR: u8 = 0xf4;

// see subchapters 3.3.2, 3.3.1 and 3.6
let value = 0b100_010_11;

let buf = [REG_ADDR, value];
i2c.write(DEVICE_ADDR, &buf).unwrap();
```

```rust
const DEVICE_ADDR: u8 =  0x77;
const REG_ADDR: u8 = 0xf4;

// see subchapters 3.3.2, 3.3.1 and 3.6
let value = 0b100_010_11;

let buf = [REG_ADDR, value];
i2c.write(DEVICE_ADDR, &buf).await.unwrap();
```

Conclusion

we talked about

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