



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*

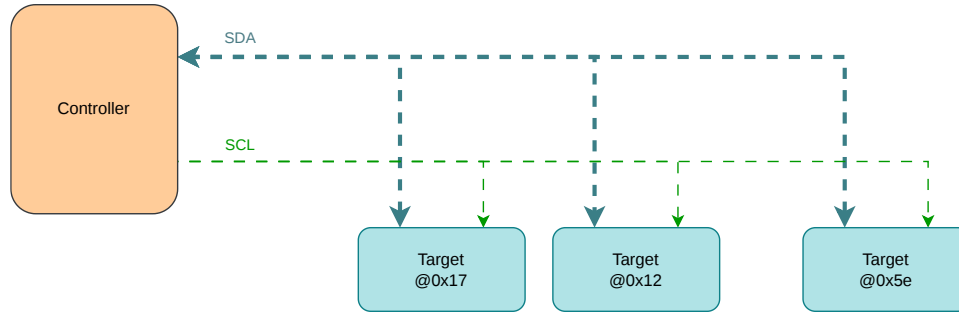
2. **Paul Denisowski**, *Understanding I2C*



I2C

a.k.a *I square C*

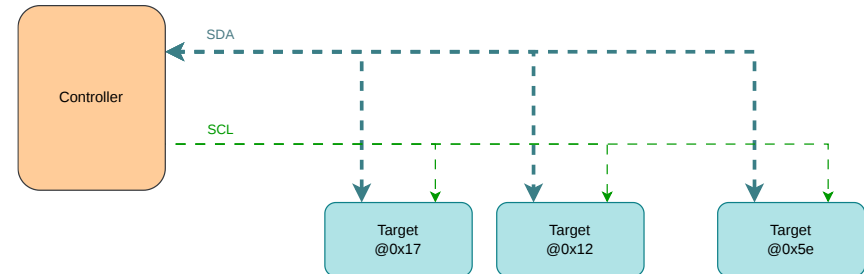
- Used for communication between integrated circuits
- Sensors usually expose an *SPI* and an *I2C* interface
- Two device types:
 - *controller* (master) - initiates the communication (usually MCU)
 - *target* (slave) - receive and transmit data when the *controller* requests (usually the sensor)





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



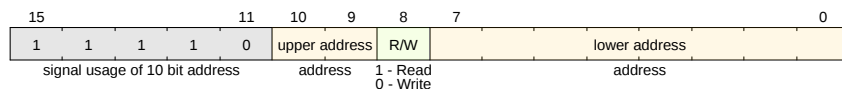


Transmission Example

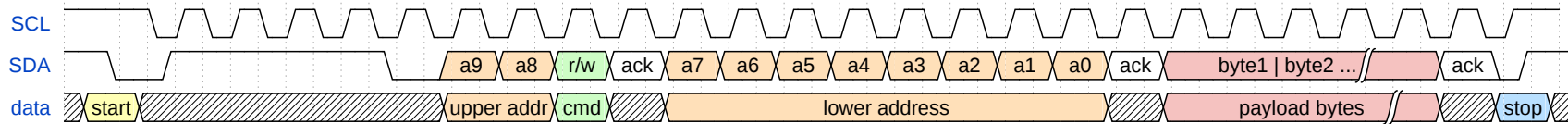
10 bit address

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 controller and target 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 several targets</i>	a receiver and a transmitter
Speed	<i>5 Mbit/s</i>	usually 100 Kbit/s, 400 Kbit/s and 1 Mbit/s



-
- Legend:**
- Power
 - Ground
 - UART / UART (default)
 - GPIO, PIO, and PWM
 - ADC
 - SPI / SPI (default)
 - I2C / I2C (default)
 - System Control
 - Debugging
- Pin Connections:**
- GPIO 0: 3.3V
 - GPIO 1: GND
 - GPIO 2: GND
 - GPIO 3: GND
 - GPIO 4: 3.3V
 - GPIO 5: GND
 - GPIO 6: 3.3V
 - GPIO 7: GND
 - GPIO 8: 3.3V
 - GPIO 9: GND
 - GPIO 10: 3.3V
 - GPIO 11: GND
 - GPIO 12: 3.3V
 - GPIO 13: GND
 - GPIO 14: 3.3V
 - GPIO 15: GND
 - GPIO 16: 3.3V
 - GPIO 17: GND
 - GPIO 18: 3.3V
 - GPIO 19: GND
 - GPIO 20: 3.3V
 - GPIO 21: GND
 - GPIO 22: 3.3V
 - GPIO 23: GND
 - GPIO 24: 3.3V
 - GPIO 25: GND
 - GPIO 26: 3.3V
 - GPIO 27: GND
 - GPIO 28: 3.3V
 - GPIO 29: GND
 - GPIO 30: 3.3V
 - GPIO 31: GND
 - GPIO 32: 3.3V
 - GPIO 33: GND
 - GPIO 34: 3.3V
 - GPIO 35: GND
 - GPIO 36: 3.3V
 - GPIO 37: GND
 - GPIO 38: 3.3V
 - GPIO 39: GND
 - GPIO 40: 3.3V
 - GPIO 41: GND
 - GPIO 42: 3.3V
 - GPIO 43: GND
 - GPIO 44: 3.3V
 - GPIO 45: GND
 - GPIO 46: 3.3V
 - GPIO 47: GND
 - GPIO 48: 3.3V
 - GPIO 49: GND
 - GPIO 50: 3.3V
 - GPIO 51: GND
 - GPIO 52: 3.3V
 - GPIO 53: GND
 - GPIO 54: 3.3V
 - GPIO 55: GND
 - GPIO 56: 3.3V
 - GPIO 57: GND
 - GPIO 58: 3.3V
 - GPIO 59: GND
 - GPIO 60: 3.3V
 - GPIO 61: GND
 - GPIO 62: 3.3V
 - GPIO 63: GND
 - GPIO 64: 3.3V
 - GPIO 65: GND
 - GPIO 66: 3.3V
 - GPIO 67: GND
 - GPIO 68: 3.3V
 - GPIO 69: GND
 - GPIO 70: 3.3V
 - GPIO 71: GND
 - GPIO 72: 3.3V
 - GPIO 73: GND
 - GPIO 74: 3.3V
 - GPIO 75: GND
 - GPIO 76: 3.3V
 - GPIO 77: GND
 - GPIO 78: 3.3V
 - GPIO 79: GND
 - GPIO 80: 3.3V
 - GPIO 81: GND
 - GPIO 82: 3.3V
 - GPIO 83: GND
 - GPIO 84: 3.3V
 - GPIO 85: GND
 - GPIO 86: 3.3V
 - GPIO 87: GND
 - GPIO 88: 3.3V
 - GPIO 89: GND
 - GPIO 90: 3.3V
 - GPIO 91: GND
 - GPIO 92: 3.3V
 - GPIO 93: GND
 - GPIO 94: 3.3V
 - GPIO 95: GND
 - GPIO 96: 3.3V
 - GPIO 97: GND
 - GPIO 98: 3.3V
 - GPIO 99: GND
 - GPIO 100: 3.3V
 - GPIO 101: GND
 - GPIO 102: 3.3V
 - GPIO 103: GND
 - GPIO 104: 3.3V
 - GPIO 105: GND
 - GPIO 106: 3.3V
 - GPIO 107: GND
 - GPIO 108: 3.3V
 - GPIO 109: GND
 - GPIO 110: 3.3V
 - GPIO 111: GND
 - GPIO 112: 3.3V
 - GPIO 113: GND
 - GPIO 114: 3.3V
 - GPIO 115: GND
 - GPIO 116: 3.3V
 - GPIO 117: GND
 - GPIO 118: 3.3V
 - GPIO 119: GND
 - GPIO 120: 3.3V
 - GPIO 121: GND
 - GPIO 122: 3.3V
 - GPIO 123: GND
 - GPIO 124: 3.3V
 - GPIO 125: GND
 - GPIO 126: 3.3V
 - GPIO 127: GND
 - GPIO 128: 3.3V
 - GPIO 129: GND
 - GPIO 130: 3.3V
 - GPIO 131: GND
 - GPIO 132: 3.3V
 - GPIO 133: GND
 - GPIO 134: 3.3V
 - GPIO 135: GND
 - GPIO 136: 3.3V
 - GPIO 137: GND
 - GPIO 138: 3.3V
 - GPIO 139: GND
 - GPIO 140: 3.3V
 - GPIO 141: GND
 - GPIO 142: 3.3V
 - GPIO 143: GND
 - GPIO 144: 3.3V
 - GPIO 145: GND
 - GPIO 146: 3.3V
 - GPIO 147: GND
 - GPIO 148: 3.3V
 - GPIO 149: GND
 - GPIO 150: 3.3V
 - GPIO 151: GND
 - GPIO 152: 3.3V
 - GPIO 153: GND
 - GPIO 154: 3.3V
 - GPIO 155: GND
 - GPIO 156: 3.3V
 - GPIO 157: GND
 - GPIO 158: 3.3V
 - GPIO 159: GND
 - GPIO 160: 3.3V
 - GPIO 161: GND
 - GPIO 162: 3.3V
 - GPIO 163: GND
 - GPIO 164: 3.3V
 - GPIO 165: GND
 - GPIO 166: 3.3V
 - GPIO 167: GND
 - GPIO 168: 3.3V
 - GPIO 169: GND
 - GPIO 170: 3.3V
 - GPIO 171: GND
 - GPIO 172: 3.3V
 - GPIO 173: GND
 - GPIO 174: 3.3V
 - GPIO 175: GND
 - GPIO 176: 3.3V
 - GPIO 177: GND
 - GPIO 178: 3.3V
 - GPIO 179: GND
 - GPIO 180: 3.3V
 - GPIO 181: GND
 - GPIO 182: 3.3V
 - GPIO 183: GND
 - GPIO 184: 3.3V
 - GPIO 185: GND
 - GPIO 186: 3.3V
 - GPIO 187: GND
 - GPIO 188: 3.3V
 - GPIO 189: GND
 - GPIO 190: 3.3V
 - GPIO 191: GND
 - GPIO 192: 3.3V
 - GPIO 193: GND
 - GPIO 194: 3.3V
 - GPIO 195: GND
 - GPIO 196: 3.3V
 - GPIO 197: GND
 - GPIO 198: 3.3V
 - GPIO 199: GND
 - GPIO 200: 3.3V
 - GPIO 201: GND
 - GPIO 202: 3.3V
 - GPIO 203: GND
 - GPIO 204: 3.3V
 - GPIO 205: GND
 - GPIO 206: 3.3V
 - GPIO 207: GND
 - GPIO 208: 3.3V
 - GPIO 209: GND
 - GPIO 210: 3.3V
 - GPIO 211: GND
 - GPIO 212: 3.3V
 - GPIO 213: GND
 - GPIO 214: 3.3V
 - GPIO 215: GND
 - GPIO 216: 3.3V
 - GPIO 217: GND
 - GPIO 218: 3.3V
 - GPIO 219: GND
 - GPIO 220: 3.3V
 - GPIO 221: GND
 - GPIO 222: 3.3V
 - GPIO 223: GND
 - GPIO 224: 3.3V
 - GPIO 225: GND
 - GPIO 226: 3.3V
 - GPIO 227: GND
 - GPIO 228: 3.3V
 - GPIO 229: GND
 - GPIO 230: 3.3V
 - GPIO 231: GND
 - GPIO 232: 3.3V
 - GPIO 233: GND
 - GPIO 234: 3.3V
 - GPIO 235: GND
 - GPIO 236: 3.3V
 - GPIO 237: GND
 - GPIO 238: 3.3V
 - GPIO 239: GND
 - GPIO 240: 3.3V
 - GPIO 241: GND
 - GPIO 242: 3.3V
 - GPIO 243: GND
 - GPIO 244: 3.3V
 - GPIO 245: GND
 - GPIO 246: 3.3V
 - GPIO 247: GND
 - GPIO 248: 3.3V
 - GPIO 249: GND
 - GPIO 250: 3.3V
 - GPIO 251: GND
 -



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





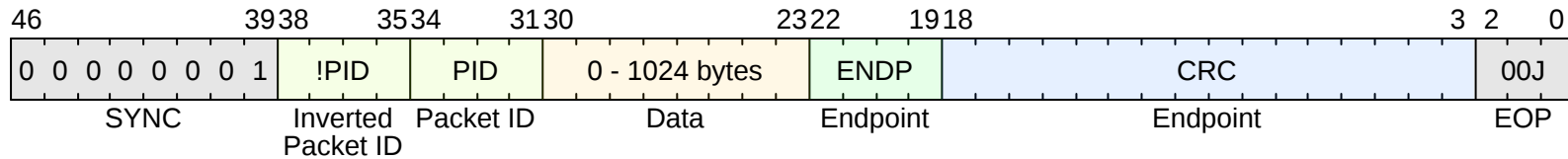
USB Packet

the smallest element of data transmission

Token



Data



Handshake



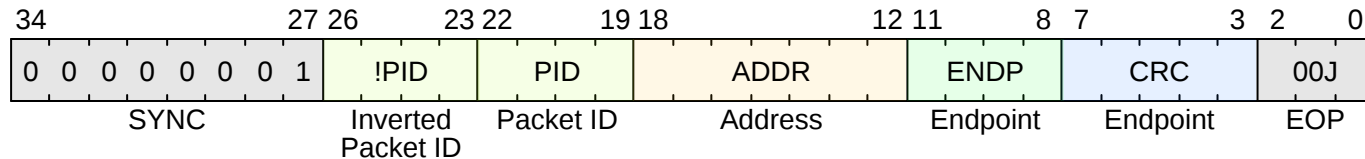


Token Packet

usually asks for a data transmission

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

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 successfully received
<i>NACK</i>	1010	data has not been successfully received
<i>STALL</i>	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*,...)



...



Status - report the status to the host

Data - optional several transfers, host transfers data

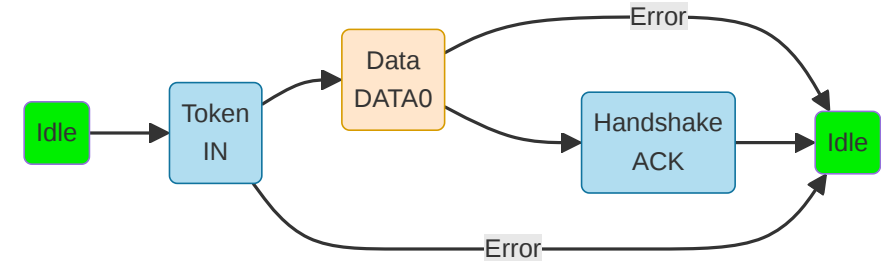
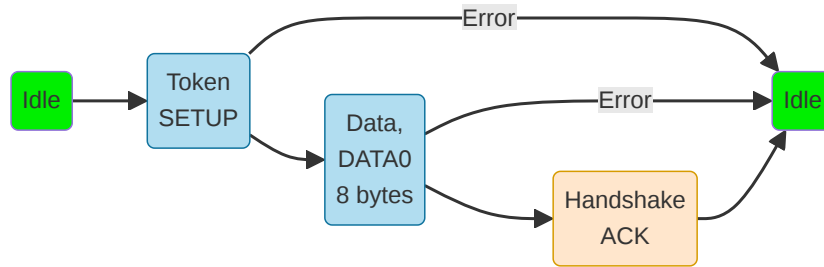




Control

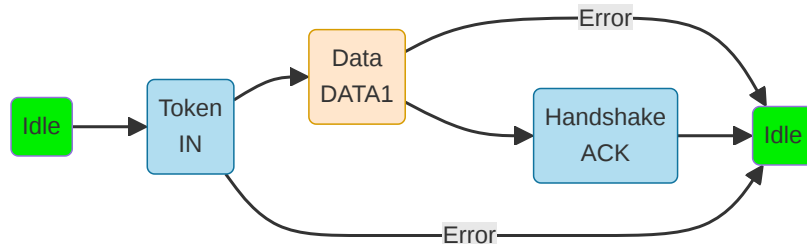
used to control a device - send data

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

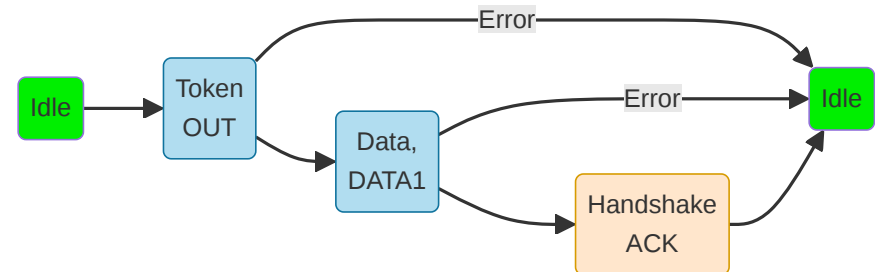


...

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



Status - report the status to the device



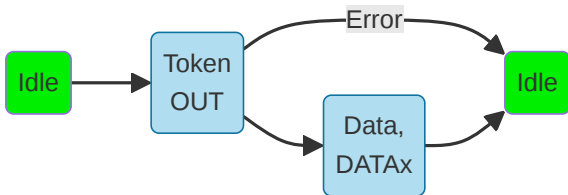


Isochronous

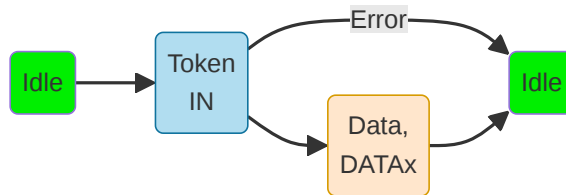
fast but not reliable transfer

- 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

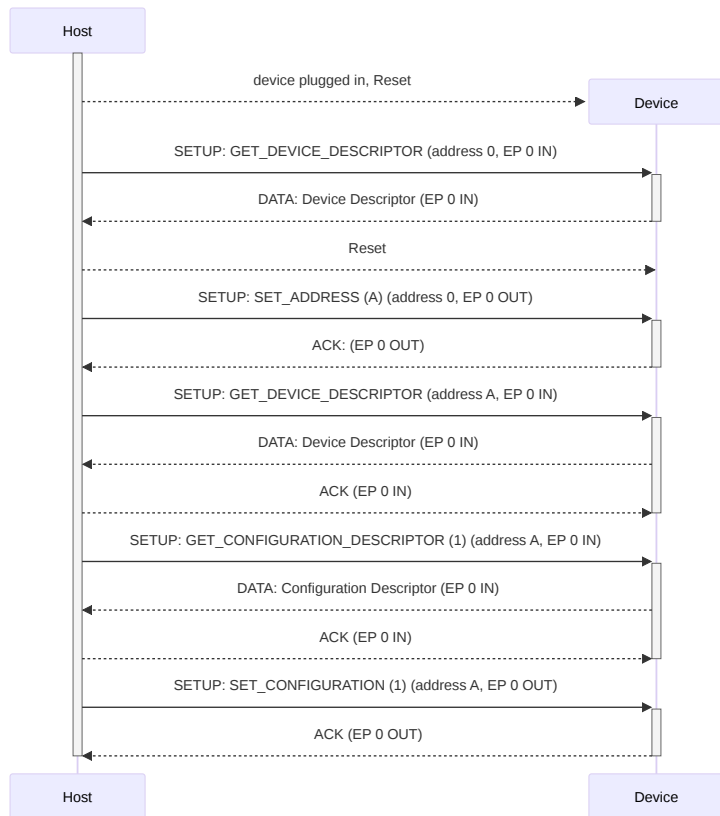
configuration, interfaces, endpoints

- 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



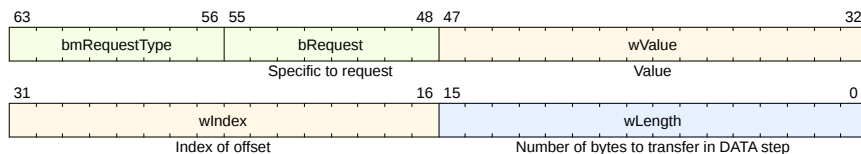


Connection



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 host and the device must synchronize their clocks
Wires	<i>DP / DM</i>	data is sent in a differential way
Devices	<i>1 host 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 talked about

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