



# I2C & USB 2.0

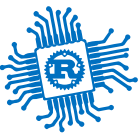
Lecture 6



# I2C & USB 2.0

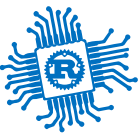
used by RP2040 & STM32U545RE

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



# I2C

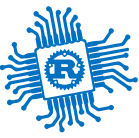
Inter-Integrated Circuit



# Bibliography

for this section

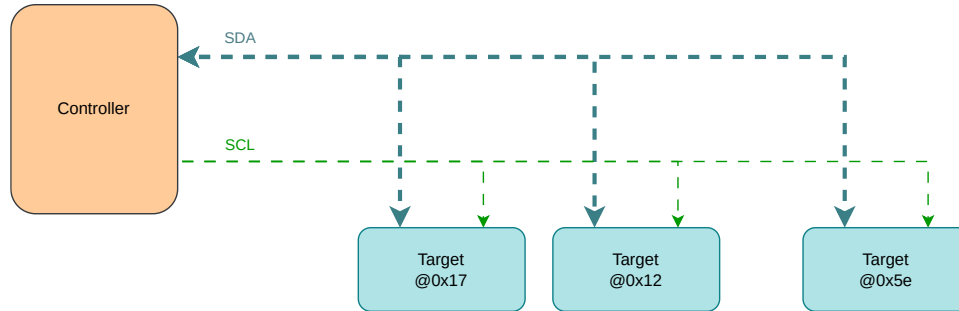
1. **Raspberry Pi Ltd**, *RP2350 Datasheet*
  - Chapter 12 - *Peripherals*
    - Chapter 12.2 - *I2C*
2. **STMicroelectronics**, *STM32U545RE Reference Manual*
  - Chapter 65 - *Inter integrated circuit interface*
3. **Paul Denisowski**, *Understanding I2C*



# I2C

a.k.a *I square C*

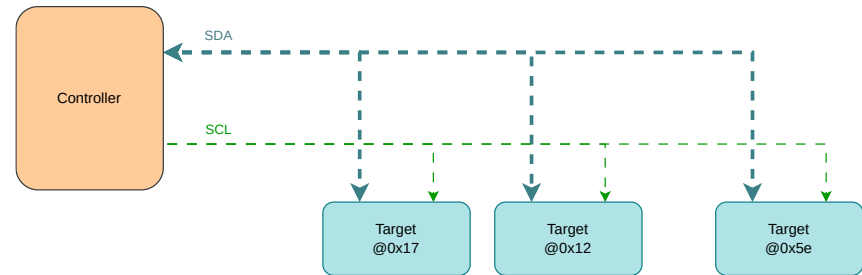
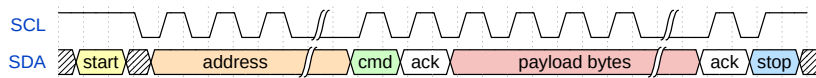
- Used for communication between integrated circuits
- Sensors usually expose an *SPI* and an *I2C* interface
- Two device types:
  - *controller* (master) - initiates the communication (usually MCU)
  - *target* (slave) - receive and transmit data when the *controller* requests (usually the sensor)





# Wires & Addresses

- **SDA** - Serial **D**Ata line - carries data from the **controller** to the **target** or from the **target** to the **controller**
- **SCL** - Serial **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**



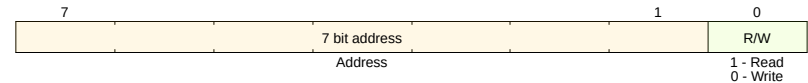


# Transmission Example

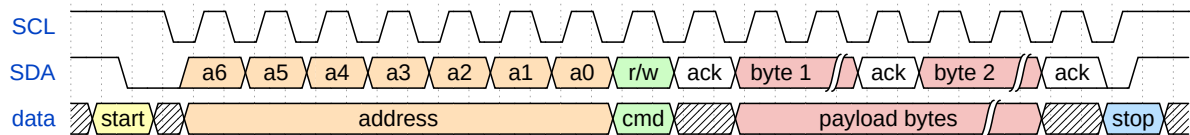
7 bit address

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



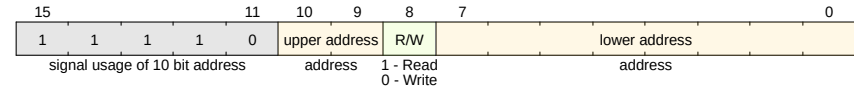


# Transmission Example

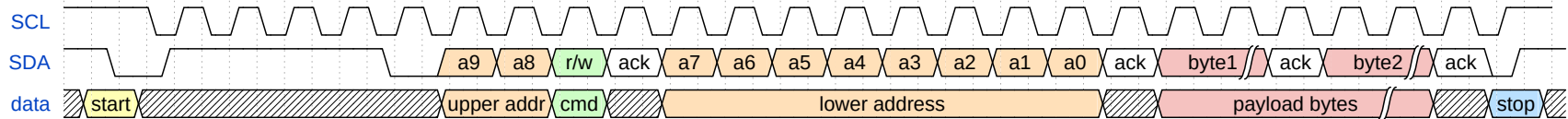
10 bit address

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

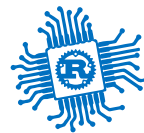
## Address Format



## 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

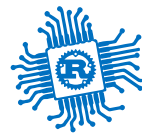


# Facts

---

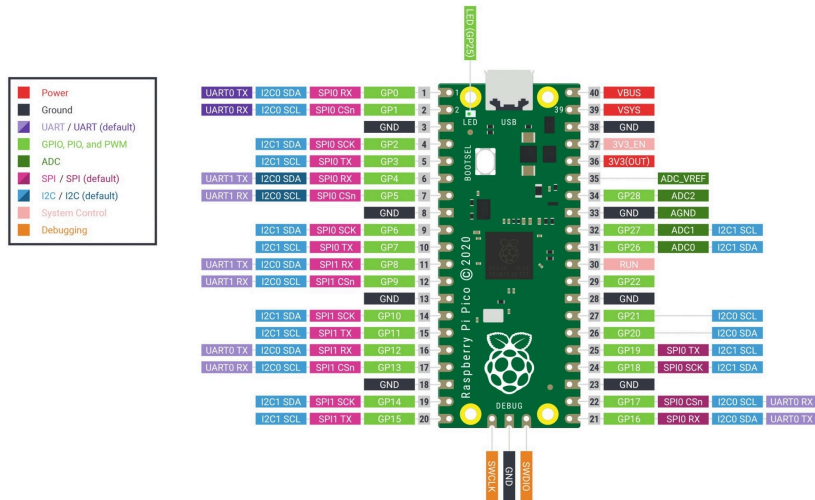
Transmission	<i>half duplex</i>	data must be sent in one direction at one time
Clock	<i>synchronized</i>	the <b>controller</b> and <b>target</b> use the same clock, there is no need for clock synchronization
Wires	<i>SDA / SCL</i>	the same read and write wire and a clock wire
Devices	<i>1 controller 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

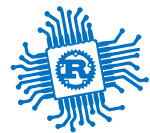
---



# Usage

- sensors
- small displays
- RP2350 has two I2C devices
- STM32U545RE has four I2C devices





# Embassy API

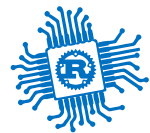
for RP2350, synchronous

```
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 let mut i2c = i2c::I2c::new_blocking(p.I2C1, scl, sda, I2cConfig::default());  
6  
7 let tx_buf = [0x90];  
8 i2c.write(0x5e, &tx_buf).unwrap();  
9  
10 let mut rx_buf = [0x00u8; 7];  
11 i2c.read(0x5e, &mut rx_buf).unwrap();  
12  
13 i2c.write_read(0x5e, &tx_buf, &mut rx_buf).unwrap();
```



# Embassy API

for STM32U545RE, synchronous

```
pub struct Config {  
    pub frequency: Hertz,  
    pub gpio_speed: Speed,  
    pub sda_pullup: bool,  
    pub scl_pullup: bool,  
    pub timeout: Duration,  
}
```

```
pub enum Error {  
    Bus, Arbitration,  
    Nack, Timeout,  
    Crc, Overrun,  
    ZeroLengthTransfer,  
}
```

```
1 use embassy_stm32::i2c::Config as I2cConfig;  
2  
3 let sda = p.PB7;  
4 let scl = p.PB6;  
5 let mut i2c = i2c::I2c::new_blocking(p.I2C1, scl, sda, I2cConfig::default());  
6  
7 let tx_buf = [0x90];  
8 i2c.blocking_write(0x5e, &tx_buf).unwrap();  
9  
10 let mut rx_buf = [0x00u8; 7];  
11 i2c.blocking_read(0x5e, &mut rx_buf).unwrap();  
12  
13 i2c.blocking_write_read(0x5e, &tx_buf, &mut rx_buf).unwrap();
```



# Embassy API

for RP2350, asynchronous

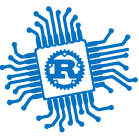
```
1 use embassy_rp::i2c::Config as I2cConfig;
2
3 bind_interrupts!(struct Irqs {
4     I2C1_IRQ => InterruptHandler<I2C1>;
5 });
6
7 let sda = p.PIN_14;
8 let scl = p.PIN_15;
9 let mut i2c = i2c::I2c::new_async(p.I2C1, scl, sda, Irqs, I2cConfig::default());
10
11 let tx_buf = [0x90];
12 i2c.write(0x5e,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



# Embassy API

for STM32U545RE, asynchronous

```
1 use embassy_rp::i2c::Config as I2cConfig;
2
3 bind_interrupts!(struct Irqs {
4     I2C1_EV => i2c::EventInterruptHandler<I2C1>;
5     I2C1_ER => i2c::ErrorInterruptHandler<I2C1>;
6 });
7
8 let sda = p.PB7;
9 let scl = p.PB6;
10 let mut i2c = I2c::new(p.I2C1, p.PB6, p.PB7, Irqs, p.GPDMA1_CH0, p.GPDMA1_CH1, Default::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();
17
18 i2c.write_read(0x5e, &tx_buf, &mut rx_buf).await.unwrap();
```



# Sensors

Analog and Digital Sensors

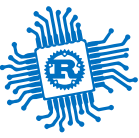


# Bibliography

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*

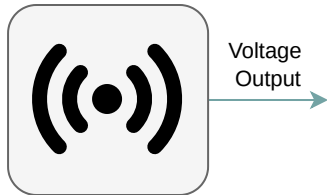


# Sensors

analog and digital

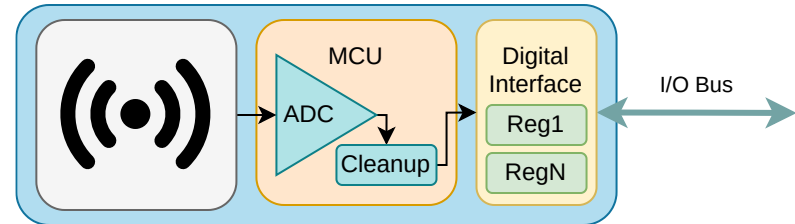
## Analog

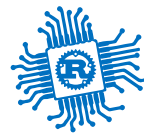
- only the transducer (the analog sensor)
- outputs (usually) voltage
- requires:
  - an ADC to be read
  - cleaning up the noise



## Digital

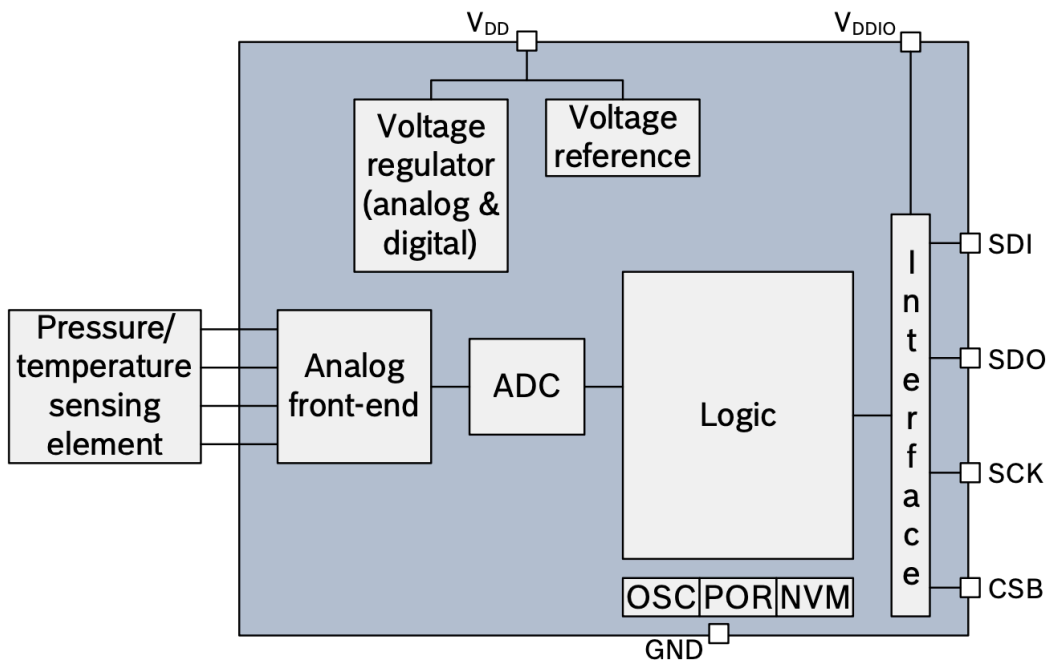
- consists of:
  - a transducer (the analog sensor)
  - an ADC
  - an MCU for cleaning up the noise
- outputs data using a digital bus



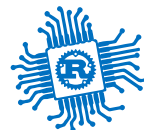


# BMP280 Digital Pressure Sensor

schematics



[Datasheet](#)



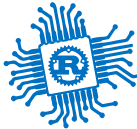
# BMP280 Digital Pressure Sensor

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

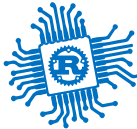


# Reading from a digital sensor

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

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

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

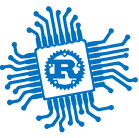


# Writing to a digital sensor

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

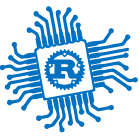
```
1  const DEVICE_ADDR: u8 = 0x77;
2  const REG_ADDR: u8 = 0xf4;
3
4  // see subchapters 3.3.2, 3.3.1 and 3.6
5  let value = 0b100_010_11;
6
7  let buf = [REG_ADDR, value];
8  i2c.write(DEVICE_ADDR, &buf).unwrap();
```

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



# USB 2.0

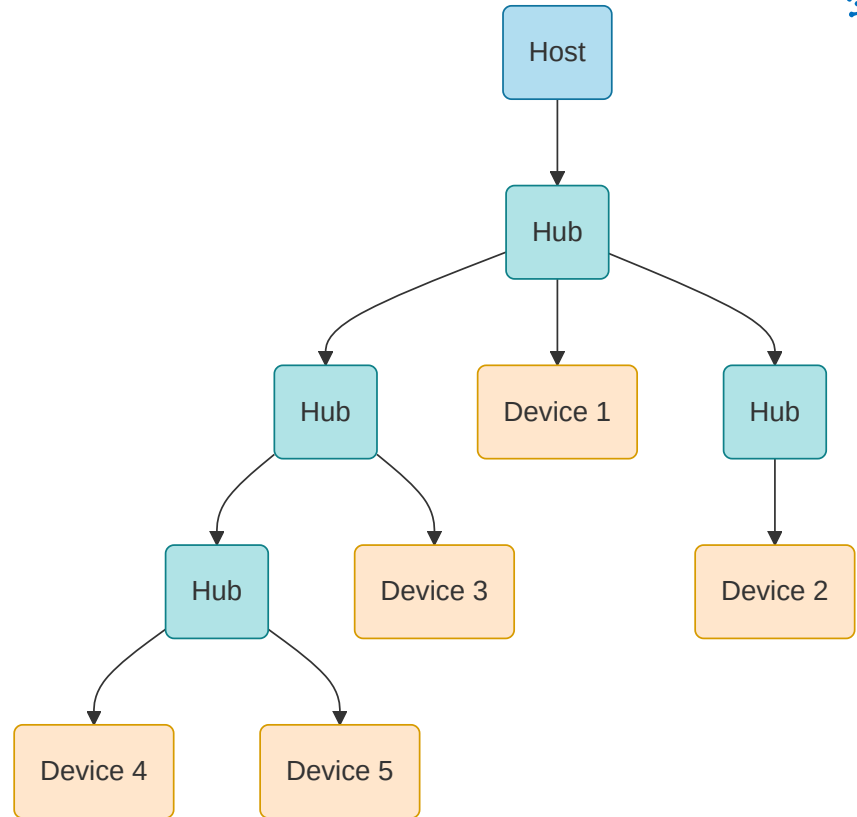
Universal Serial Bus

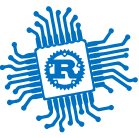


# Universal Serial Bus

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

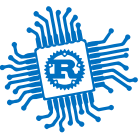




# Bibliography

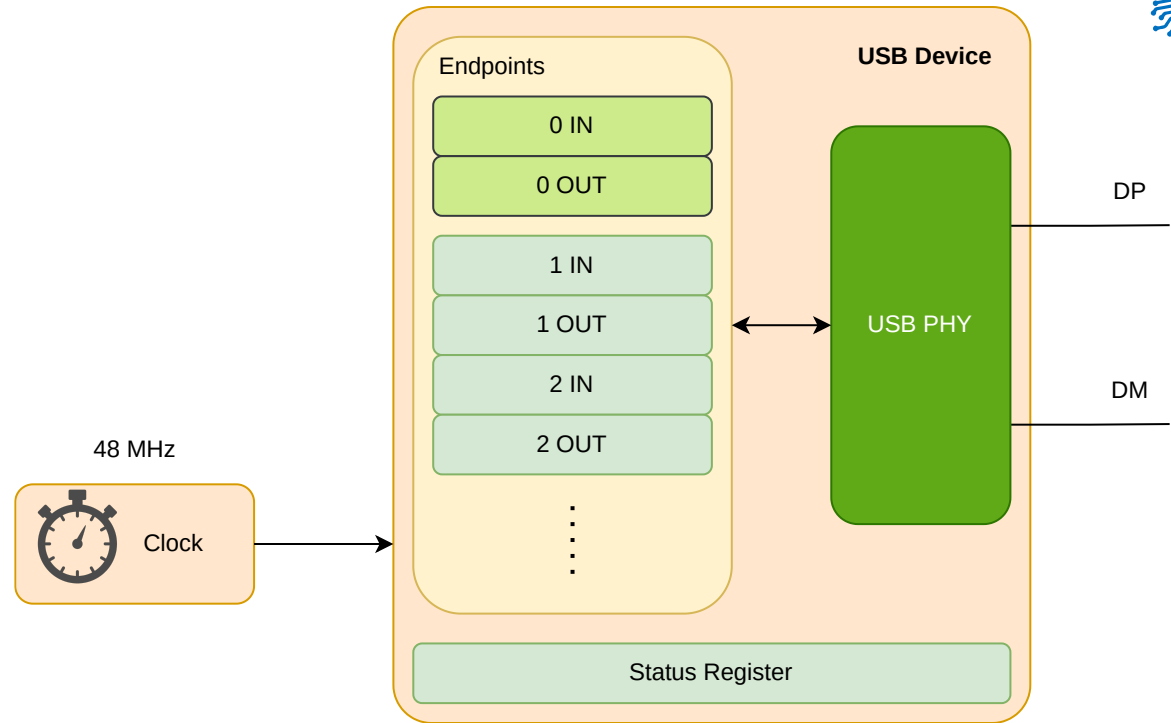
for this section

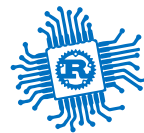
1. **Raspberry Pi Ltd**, *RP2350 Datasheet*
  - Chapter 12 - *Peripherals*
    - Chapter 12.7 - *USB*
2. **STMicroelectronics**, *STM32U545RE Reference Manual*
  - Chapter 71 - *Universal serial bus full-speed host/device interface (USB)*
3. **BeyondLogic**, *USB in a NutShell*
4. **Ben Eater**, *How does a USB keyboard work?*
5. **Ben Eater**, *How does USB device discovery work?*
6. *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

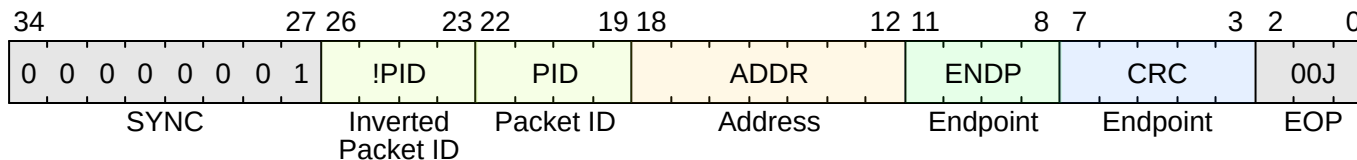




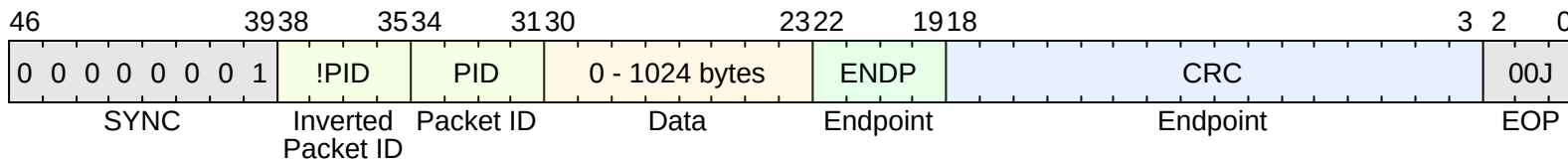
# USB Packet

the smallest element of data transmission

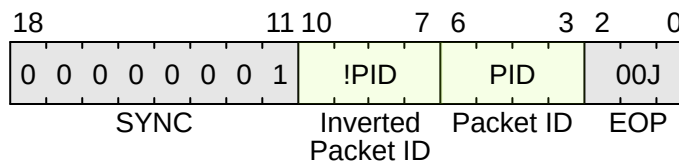
## Token



## Data



## Handshake



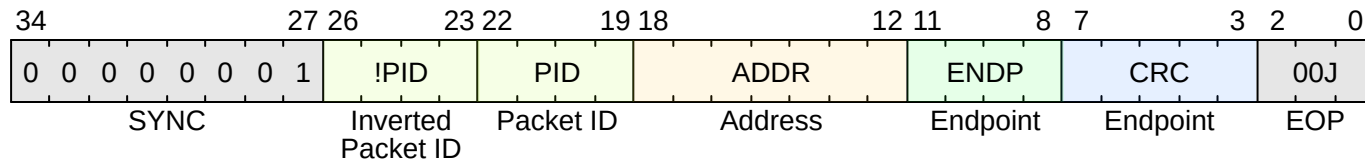


# Token Packet

usually asks for a data transmission

Type	PID	Description
<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



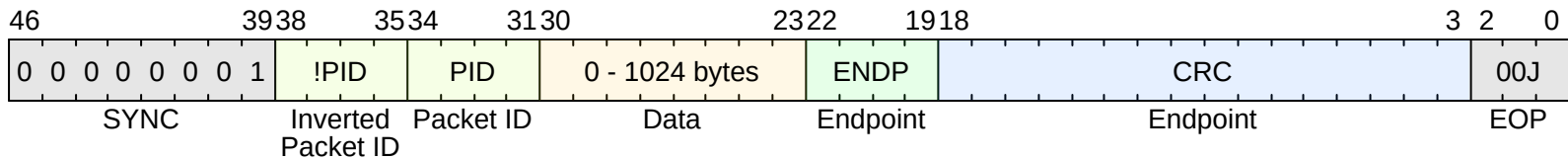


# Data Packet

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

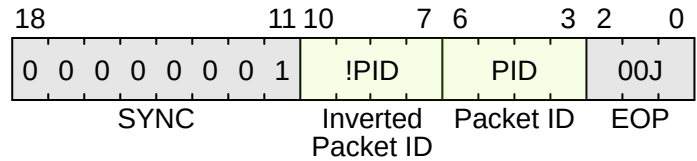


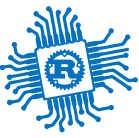


# Handshake Packet

acknowledges data

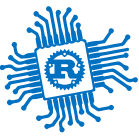
Type	PID	Description
<i>ACK</i>	0010	data has been <b>successfully received</b>
<i>NACK</i>	1010	data has <b>not</b> been <b>successfully received</b>
<i>STALL</i>	1110	the device has an <b>error</b>





# Transmission Modes

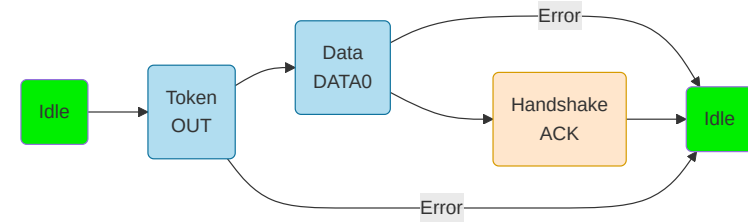
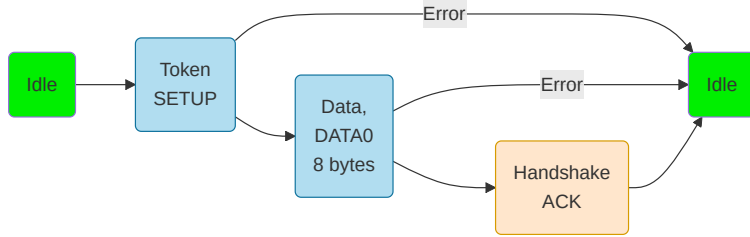
- *Control* - used for configuration
- *Isochronous* - used for high bandwidth, best effort
- *Bulk* - used for low bandwidth, stream
- *Interrupt* - used for low bandwidth, guaranteed latency



# Control

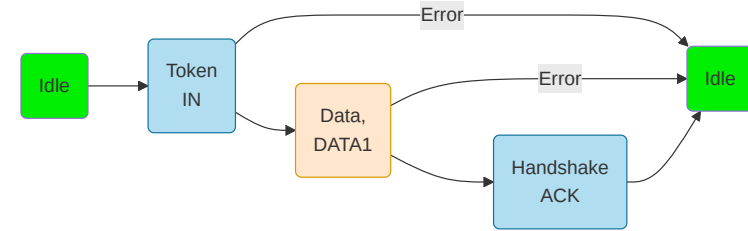
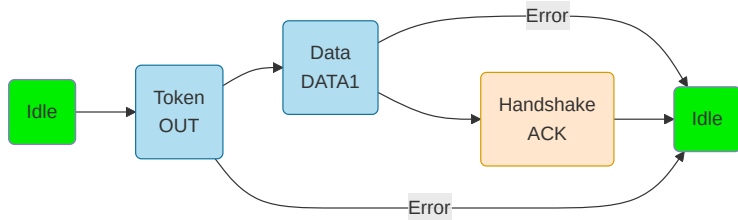
used to control a device - ask for data

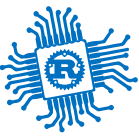
**Setup** - send a command (*GET\_DESCRIPTOR*, ...)



...

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

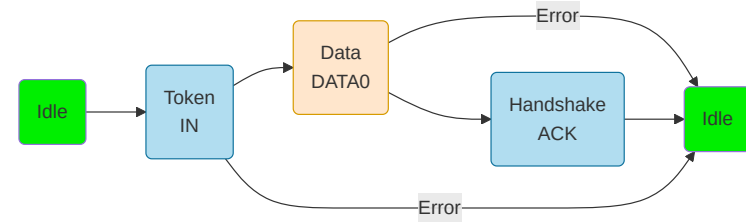
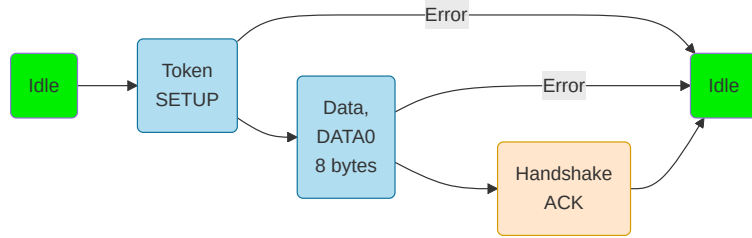




# Control

used to control a device - send data

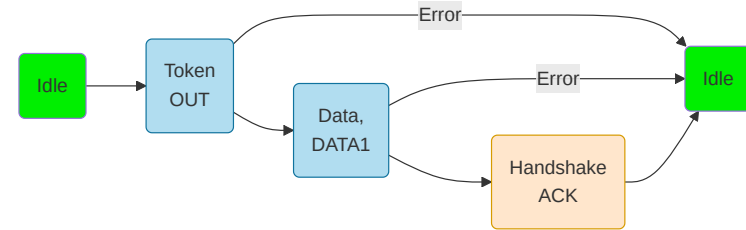
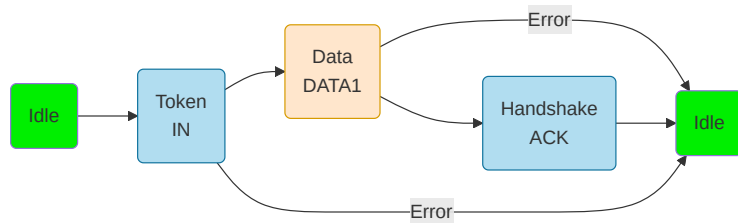
**Setup** - send a command (*SET\_ADDRESS*, ...)

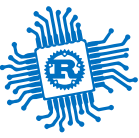


...

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

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



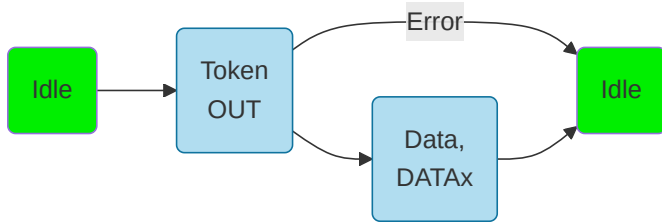


# Isochronous

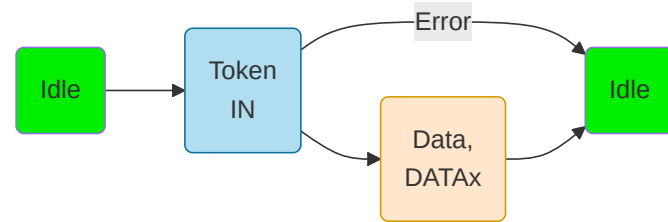
fast but not reliable transfer

- 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





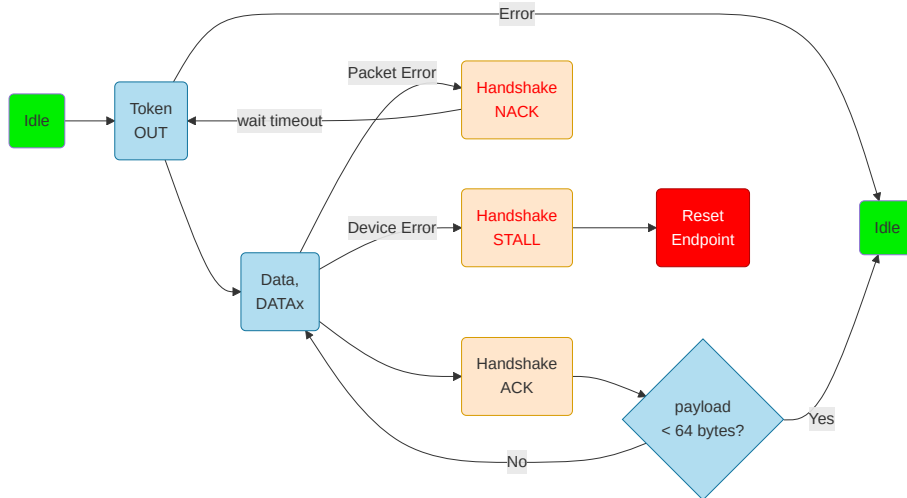
# Bulk

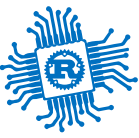
slow, but reliable transfer

- does not have a guaranteed bandwidth
- does not allow data loss
- 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



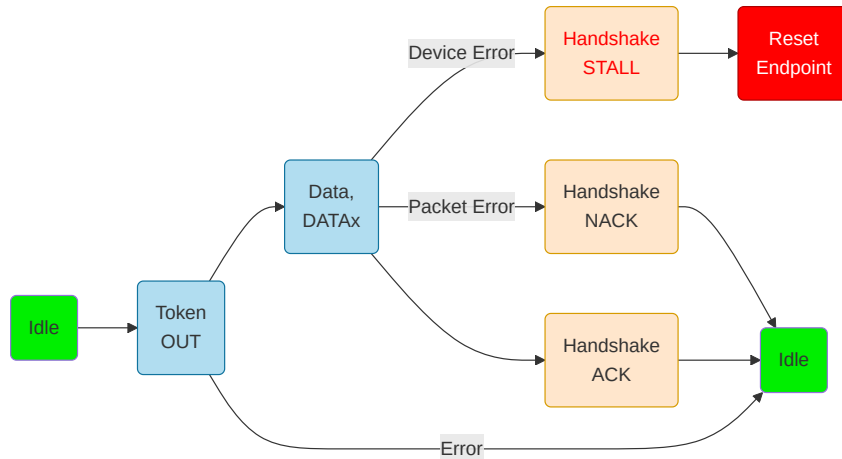


# Interrupt

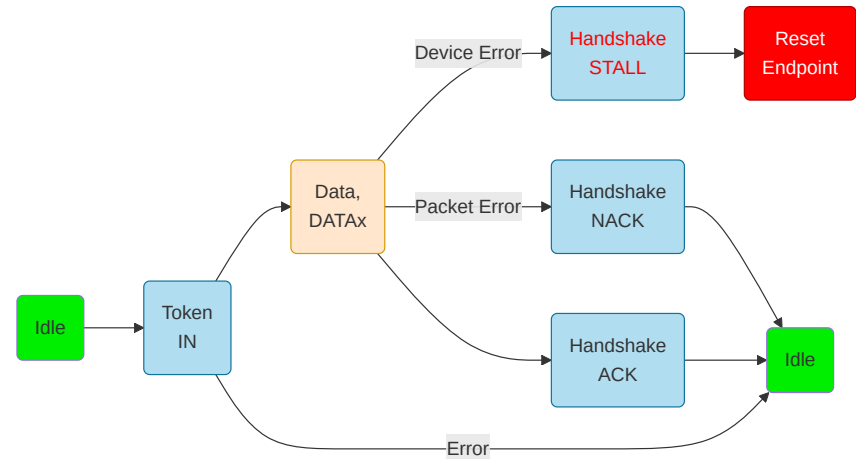
transfer data at a minimum time interval

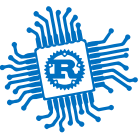
- the endpoint descriptor asks the host start an interrupt transfer at a time interval
- used for sending and receiving data at certain intervals

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



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

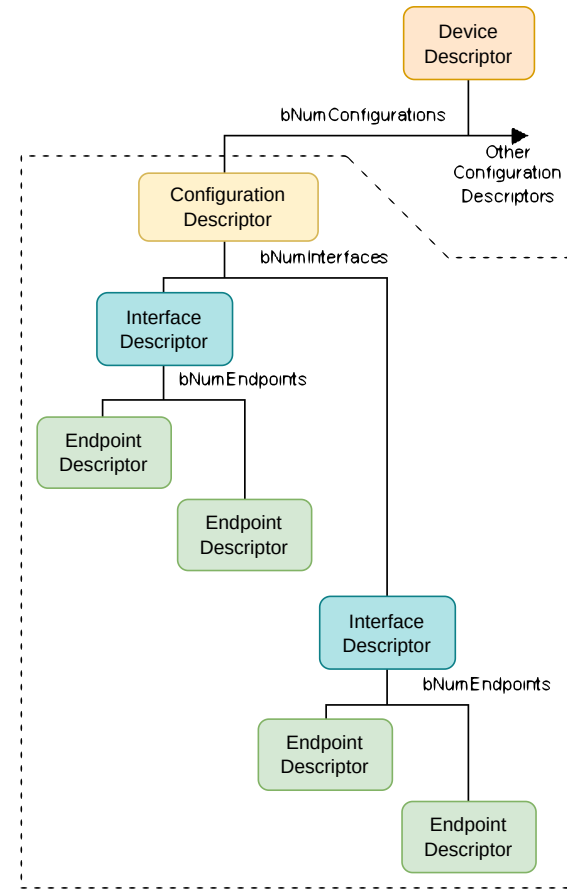




# Device Organization

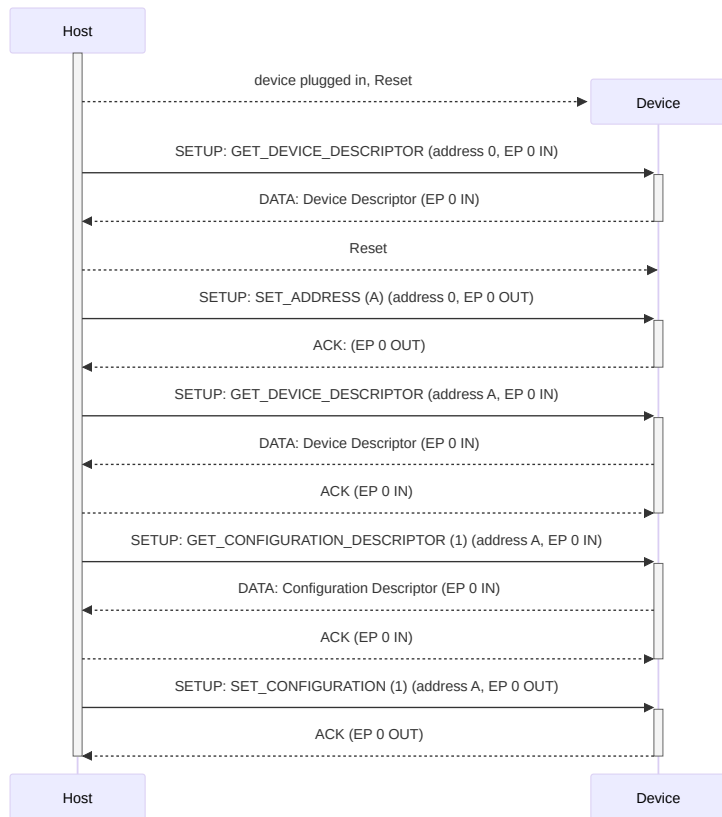
configuration, interfaces, endpoints

- 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 *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



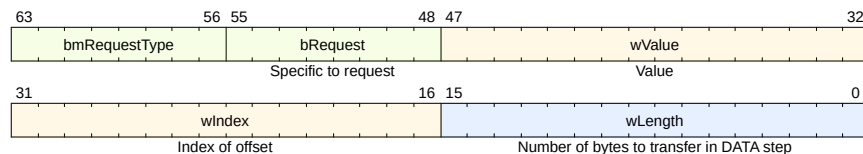


# Connection

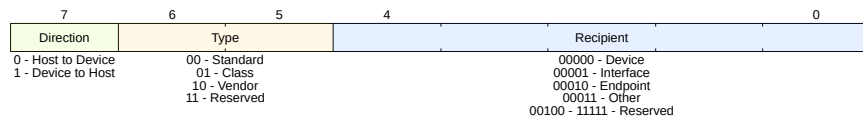


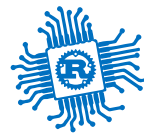
# Token SETUP Packet

The DATA packet of the SETUP Control Transfer



## bmRequestType field

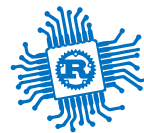




# Device Classes

predefined devices types

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



# Device Descriptor

describes the whole device

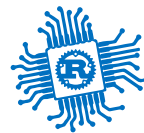
Field	Value	Description
<i>bLength</i>	18	Descriptor length in bytes.
<i>bDescriptorType</i>	1	Descriptor type (1 = Device Descriptor).
<i>bcdUSB</i>	0x0200	USB specification release number (2.0).
<i>bDeviceClass</i>	0xFF	Device class (0xFF = Vendor Specific).
<i>bDeviceSubClass</i>	0	Device subclass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



# Configuration Descriptor

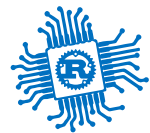
one of the configurations

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



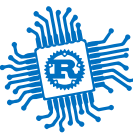
# Interface Descriptor

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



# Endpoint Descriptor

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

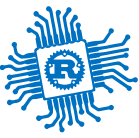


# Strings Descriptor

## String Descriptor for Configuration and Interface

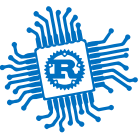
Field	Value	Description
<i>bLength</i>	4	Descriptor length in bytes (always 4 for string descriptor header).
<i>bDescriptorType</i>	3	Descriptor type (3 = String Descriptor).
<i>bString</i>	0x09 0x55 0x53 0x42 0x20 0x43 0x6F 0x6E 0x66 0x69 0x67 0x20 0x31	UTF-16LE string encoding: "USB Config 1" .

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



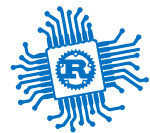
# USB 1.0 and 2.0 Modes

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	<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 several devices</i>	a receiver and a transmitter
Speed	<i>480 Mbit/s</i>	



# Embassy API

for RP2350 and STM32U545RE, setup the device

```
1 use embassy_stm32::usb::{Driver, InterruptHandler};
2 use embassy_usb::Config as UsbConfig;
3
4 bind_interrupts!(struct Irqs {
5     USB => InterruptHandler<USB>;
6 });
7
8 let mut config = UsbConfig::new(0xc0de, 0xcafe);
9 config.manufacturer = Some("Embassy");
10 config.product = Some("USB sender receiver");
11 config.serial_number = Some("12345678");
12 config.max_power = 100;
13 config.max_packet_size_0 = 64;
14
15 let driver = Driver::new(peripherals.USB, Irqs, peripherals.PA12, peripherals.PA11);
```

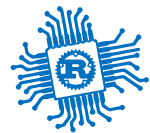
JP3 has to be moved to USB USER



# Embassy API

for RP2350 and STM32U545RE, setup the descriptors

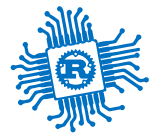
```
1 use embassy_usb::msos::{self, windows_version};
2 use embassy_usb::Builder;
3
4 // It needs some buffers for building the descriptors.
5 let mut config_descriptor = [0; 256];
6 let mut bos_descriptor = [0; 256];
7 let mut msos_descriptor = [0; 256];
8 let mut control_buf = [0; 64];
9
10 let mut builder = Builder::new(driver, config,
11     &mut config_descriptor, &mut bos_descriptor, &mut msos_descriptor, &mut control_buf,
12 );
13
14 // Required for Windows
15 const DEVICE_INTERFACE_GUIDS: &[&str] = &["{AFB9A6FB-30BA-44BC-9232-806CFC875321}"];
16 builder.msos_descriptor(windows_version::WIN8_1, 0);
17 builder.msos_feature(msos::CompatibleIdFeatureDescriptor::new("WINUSB", ""));
18 builder.msos_feature(msos::RegistryPropertyFeatureDescriptor::new(
19     "DeviceInterfaceGUIDs",
20     msos::PropertyData::RegMultiSz(DEVICE_INTERFACE_GUIDS),
21 ));
```



# Embassy API

for RP2350 and STM32U545RE, setup the device's function and start

```
1 // Add a vendor-specific function (class 0xFF), and corresponding interface,
2 // that uses our custom handler.
3 let mut function = builder.function(0xFF, 0, 0);
4 let mut interface = function.interface();
5 let mut alt = interface.alt_setting(0xFF, 0, 0, None);
6 let mut read_ep = alt.endpoint_bulk_out(64);
7 let mut write_ep = alt.endpoint_bulk_in(64);
8 drop(function);
9
10 // Build the builder.
11 let mut usb = builder.build();
12
13 // Create the USB device handler
14 let usb_run = usb.run();
```



# Embassy API

for RP2350 and STM32U545RE, use the USB device

```
1 let echo_run = async {
2   loop {
3     read_ep.wait_enabled().await;
4     info!("Connected");
5     loop {
6       let mut data = [0; 64];
7       match read_ep.read(&mut data).await {
8         Ok(n) => {
9           info!("Got bulk: {:a}", data[..n]);
10          // Echo back to the host:
11          write_ep.write(&data[..n]).await.ok();
12        }
13        Err(_) => break,
14      }
15    }
16    info!("Disconnected");
17  }
18 };
19
20 // Run everything concurrently.
21 // If we had made everything `static` above instead, we could do this using separate tasks instead.
22 join(usb_run, echo_run).await;
```



# Host API

using nusb

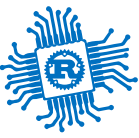
```
1 use nusb::transfer::RequestBuffer;
2
3 const BULK_OUT_EP: u8 = 0x01;
4 const BULK_IN_EP: u8 = 0x81;
5
6 async fn main() {
7     let di = nusb::list_devices()
8         .unwrap()
9         .find(|d| d.vendor_id() == 0xc0de && d.product_id() == 0xcafe)
10        .expect("no device found");
11
12    let device = di.open().expect("error opening device");
13    let interface = device.claim_interface(0).expect("error claiming interface");
14
15    let result = interface.bulk_out(BULK_OUT_EP, b"hello world".into()).await;
16    println!("{result:?}");
17
18    let result = interface.bulk_in(BULK_IN_EP, RequestBuffer::new(64)).await;
19    println!("{result:?}");
20 }
```



# Host API

using Python

```
1  import usb
2  import time
3
4  # Find the USB device
5  dev = usb.core.find(idVendor=0xc0de, idProduct=0xcafe)
6  if dev is None:
7      raise ValueError('Device not found')
8
9  dev.set_configuration() # Set the active configuration (this is usually required after device detection)
10
11  OUT_ENDPOINT = 0x01 # Usually 0x01 for OUT endpoint
12  IN_ENDPOINT = 0x81 # Usually 0x81 for IN endpoint (Endpoint 1, Direction IN)
13
14  data_to_send = b"Hello, USB Device!"
15
16  dev.write(OUT_ENDPOINT, data_to_send)
17  time.sleep(1) # Wait for a short time to ensure data is transferred
18
19  data_received = dev.read(IN_ENDPOINT, 64) # Read 64 bytes (adjust the size if needed)
20  print("Data received from device:", bytes(data_received))
21
22  usb.util.release_interface(dev, 0) # Release the device interface (optional, but good practice)
```



# Conclusion

we talked about

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