Embedded Systems

Lecture 3
Hardware-Software Interfaces – Serial Interfaces

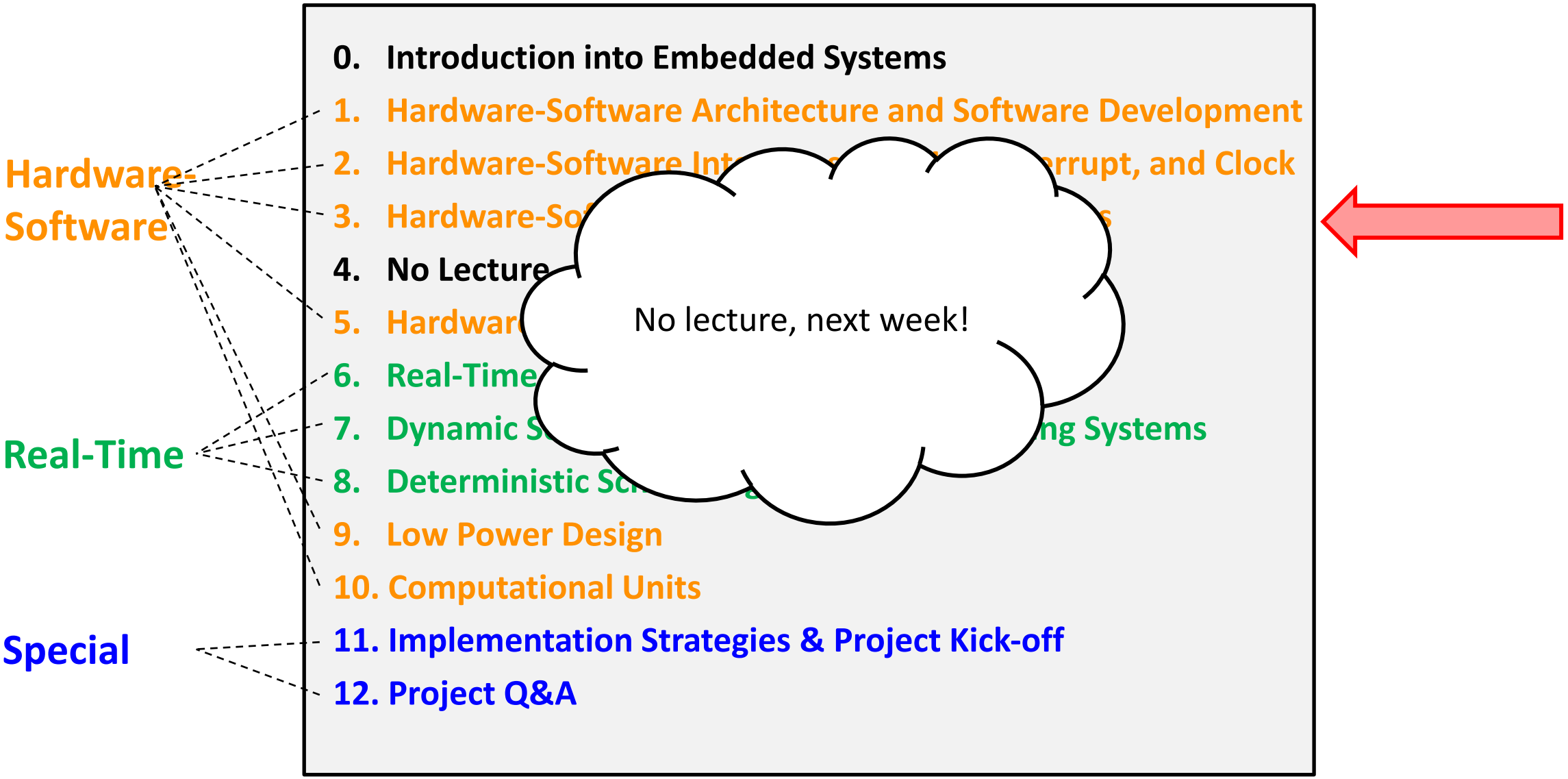
Michele Magno

D-ITET Center for Project-Based Learning

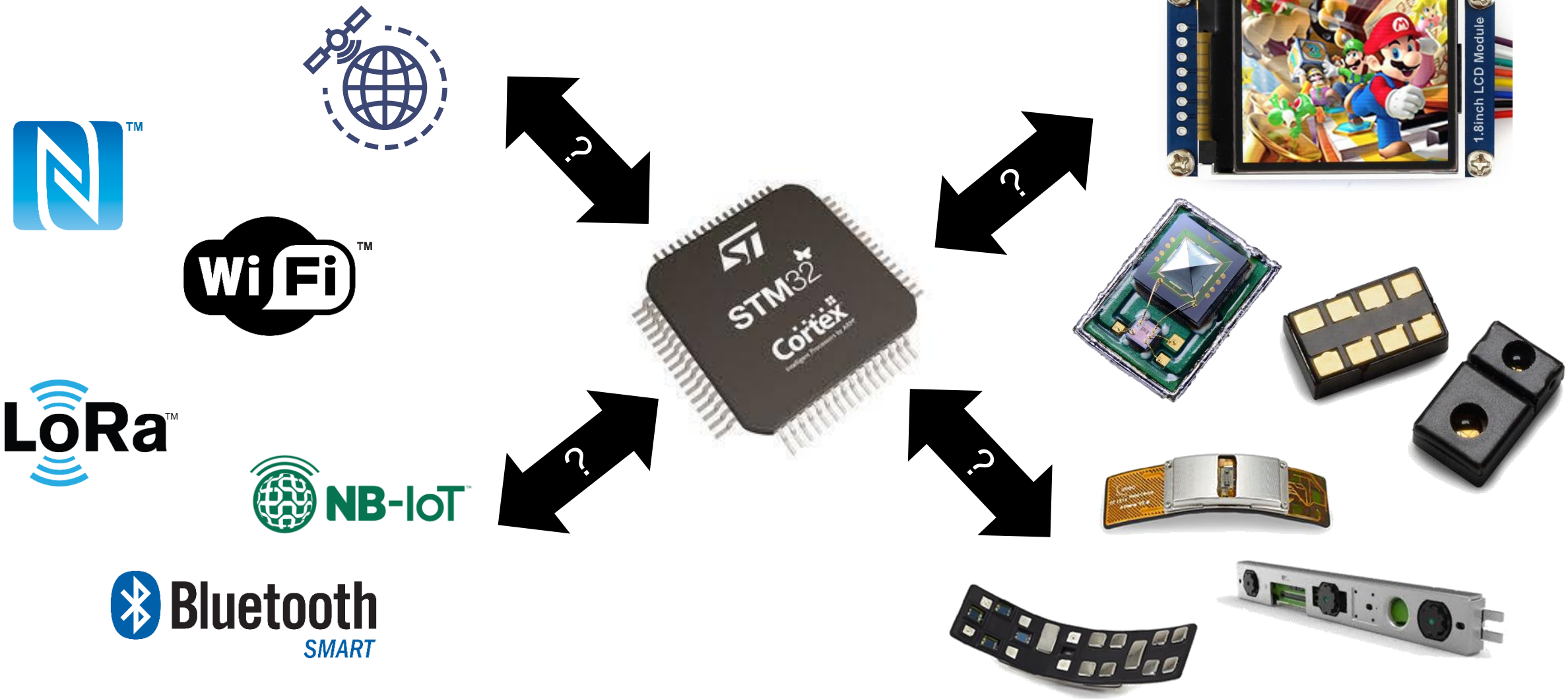
ETH zürich

Slides partially adopted from Prof. Thiele

Where We are


Hardware-Software
Real-Time
Special
0. Introduction into Embedded Systems
1. Hardware-Software Architecture and Software Development
2. Hardware-Software Int
errupt, and Clock
3. Hardware-So
s
4. No Lecture
5. Hardwar
6. Real-Time
7. Dynamic S
ng Systems
8. Deterministic Sc
9. Low Power Design
10. Computational Units
11. Implementation Strategies & Project Kick-off
12. Project Q&A
No lecture, next week!

Serial Interfaces: Sensors, Display, Radio etc.



Communication Interfaces

In embedded systems, the microcontroller communicates with sensors, actuators or other microcontrollers to *exchange information* or data.

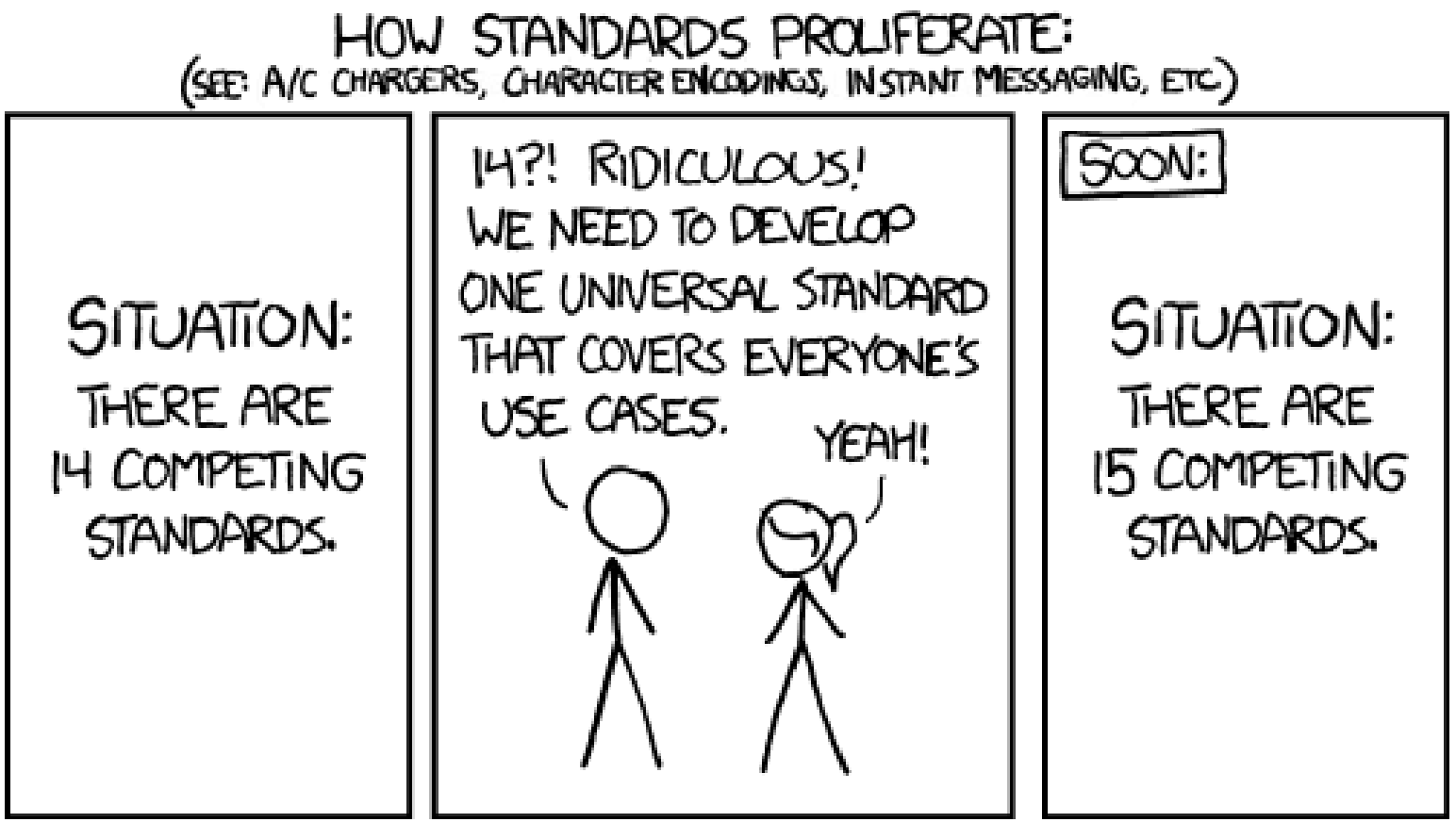
To satisfy various needs, there exists different communication protocols, such as:

- Wireless
  - Bluetooth Low Energy (BLE)
  - Near Field Communication (NFC)
  - Long Range (LoRa)
- Wired
  - Inter-Integrated Circuit (I^2C)
  - Serial Peripheral Interface (SPI)
  - Universal asynchronous receiver / transmitter (UART)

In an Embedded System often stand-alone module connected with the wired interface

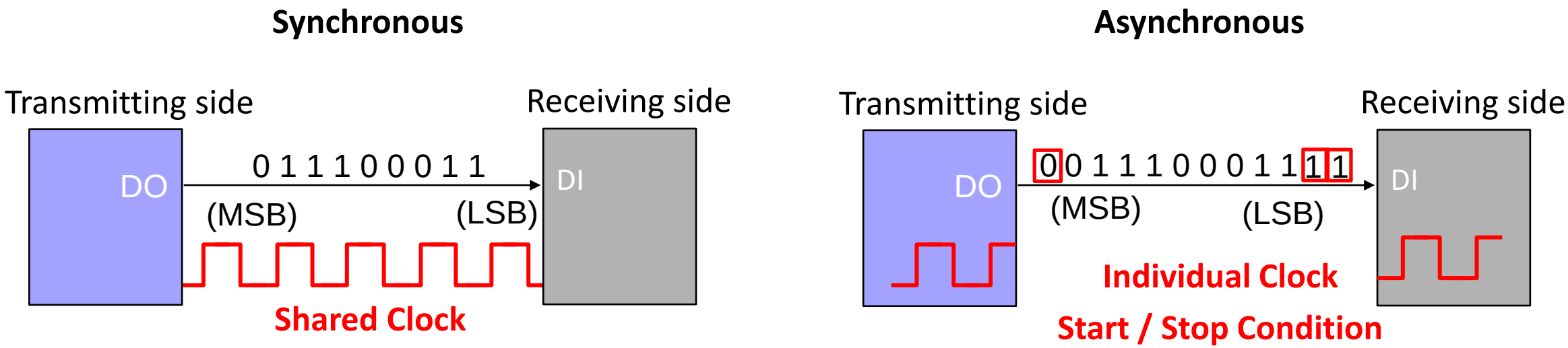
In this course!

Serial Interface Standards



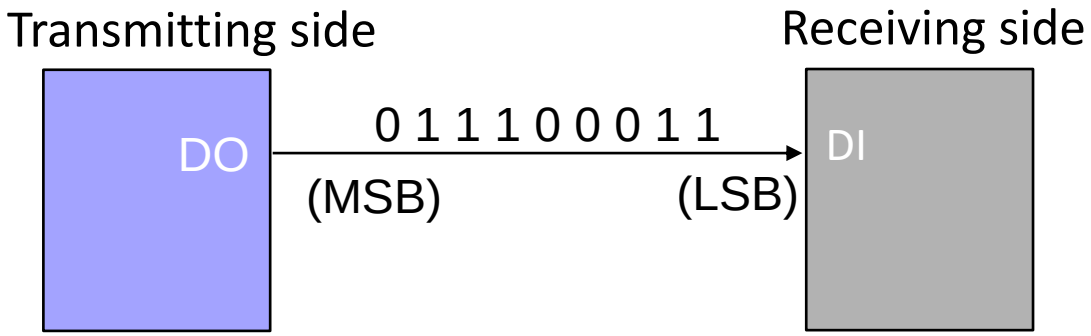
Asynchronous and Synchronous Communication

How are individual bits separated?

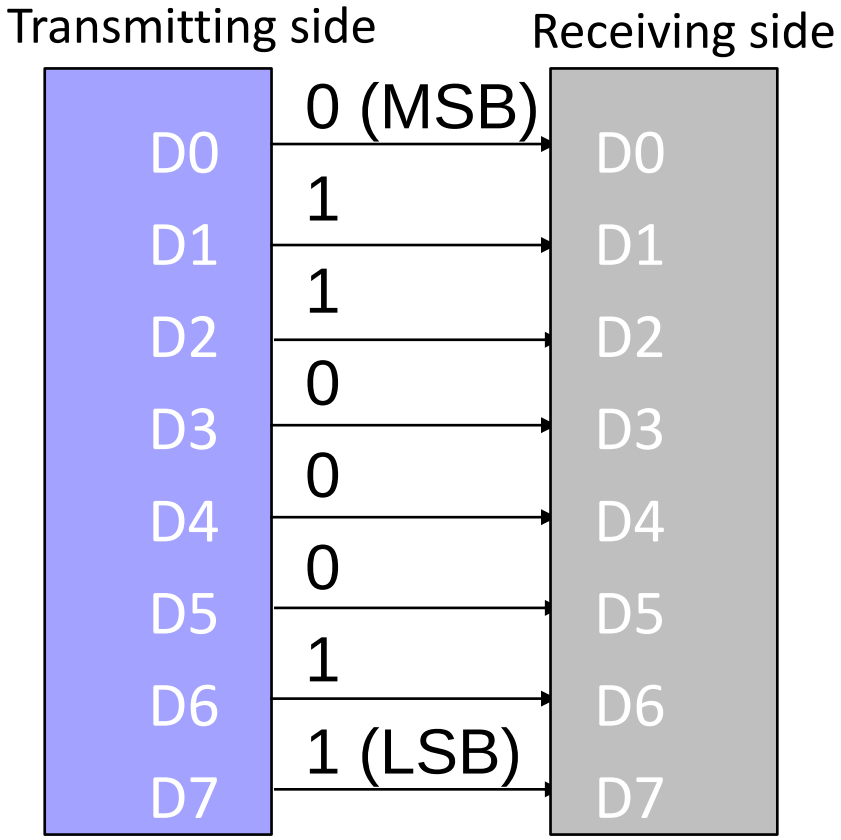


Parallel and Serial Communication

Serial interface example



Parallel interface example

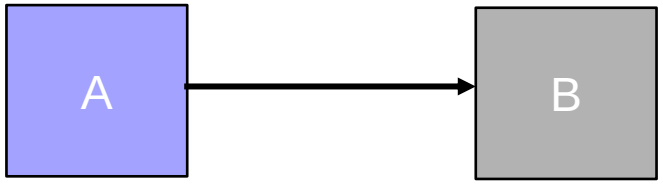


Typical Applications

Characteristic	Parallel communication	Serial communication
Data lines	One line per bit	One line
Sequence	All bits of one word simultaneously	Sequence of bits
Transmission rate at same clock speed	High	Low
Cost	High	Low
Critical characteristics	Synchronization between the different bits is demanding	Asynchronous transmission needs start and stop bits Synchronous transmission needs some other synchronization

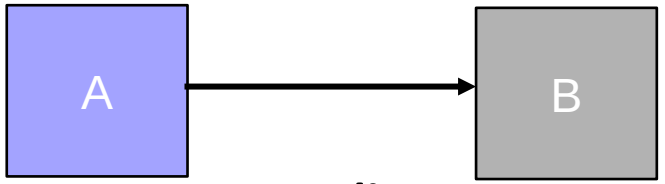
Simplex/Duplex Communication

Simplex



One direction only

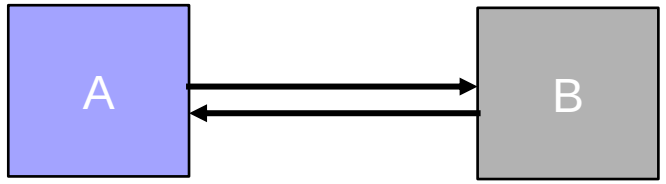
Half-Duplex



A sending, B receiving

B sending, A receiving

Full-Duplex



Bidirectional Simultaneously

UART - Universal Asynchronous Receiver-Transmitter

UART and USART

The *Universal Asynchronous Receiver-Transmitter (UART)* peripheral is used to interface MCUs with other computing devices and is based on an asynchronous communication protocol.

The *Universal Synchronous-Asynchronous Receiver Transmitter (USART)* support both asynchronous and synchronous communication protocols. Therefore, the USART includes the UART functionality.

Both UART and USART are used to *interface MCUs with other computing devices*:

- Communication with other processors, or PC (e.g. a serial terminal) for logging or debugging
- Used to connect MCUs with modems and transceivers as telephone modems, Bluetooth, Wi-Fi, GSM/GPRS

UART - Topology

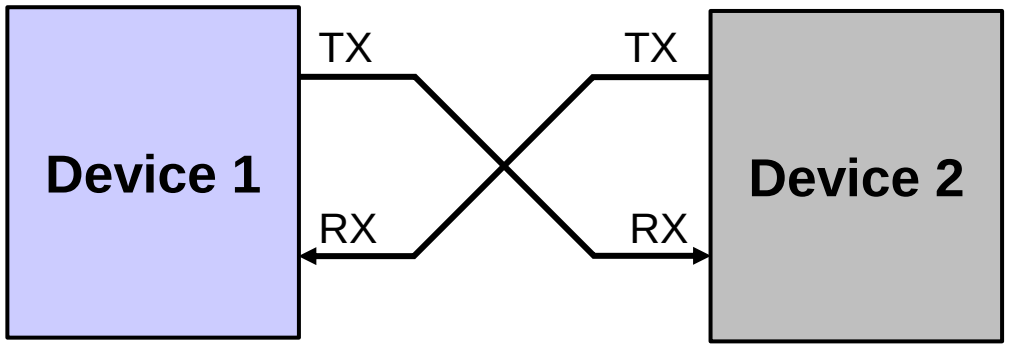
Asynchronous means *no common clock* is shared between receiver and transmitter. The communication requires that both devices have the same baud rate.

Baud rate = physical pulses per second

Because UART has no shared clock for synchronization, both devices are required to have the same baud rate.

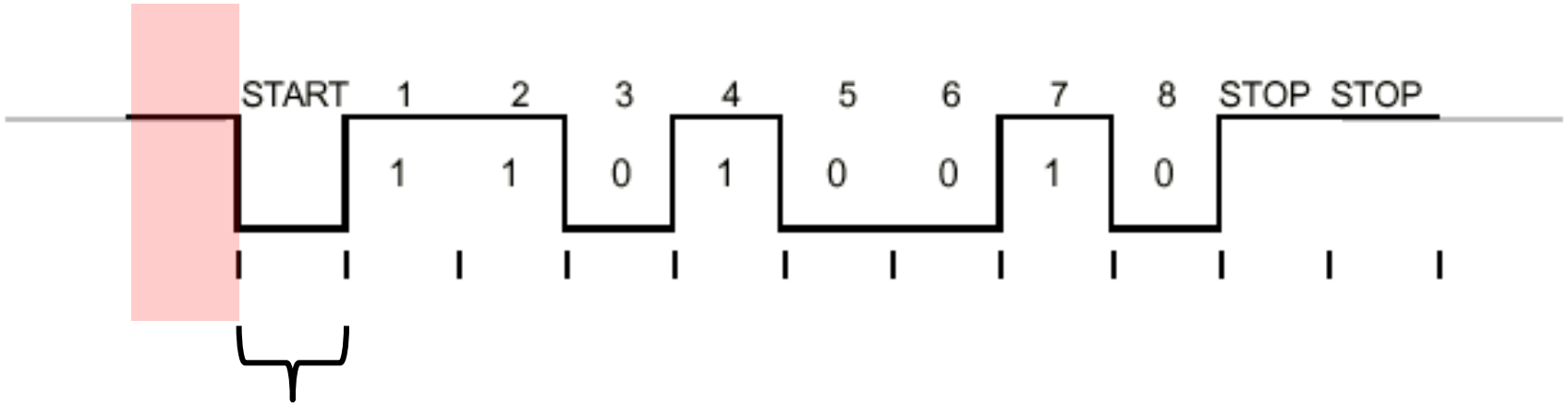
The communication is highly configurable:

- Parity / no parity bit
- Data framing
- Simplex or full-duplex



UART - Interface Protocol

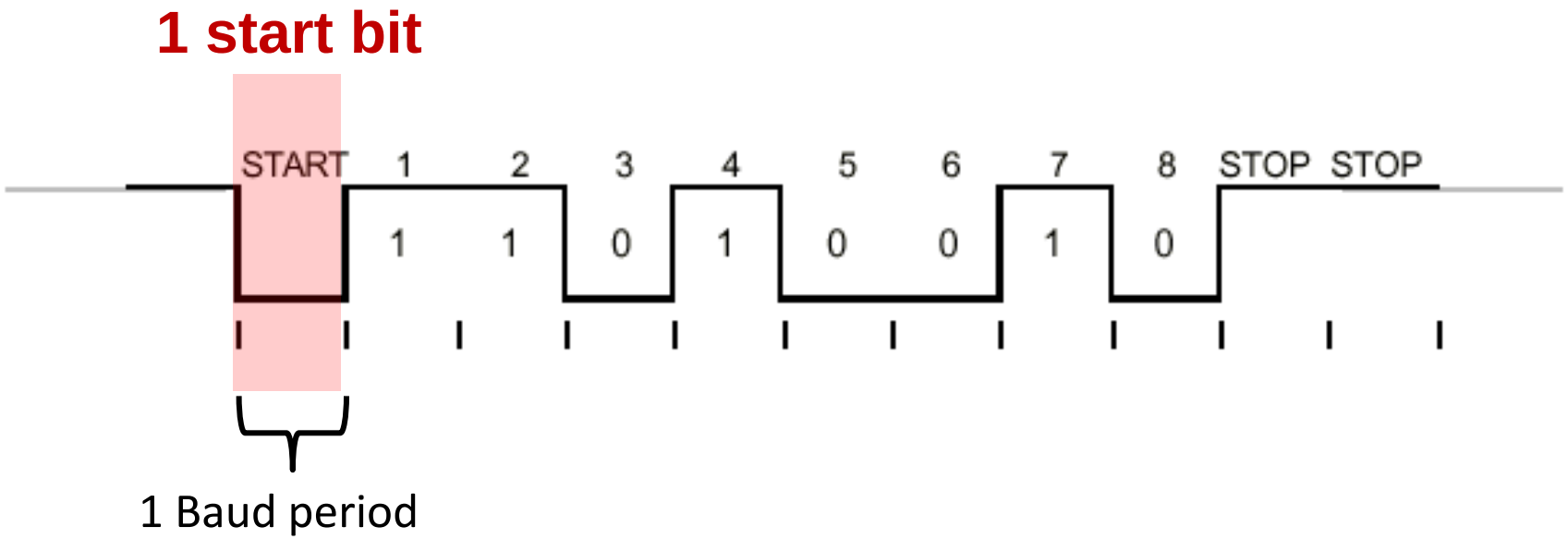
- In idle, the transmission line is driven to 1



1 Baud period

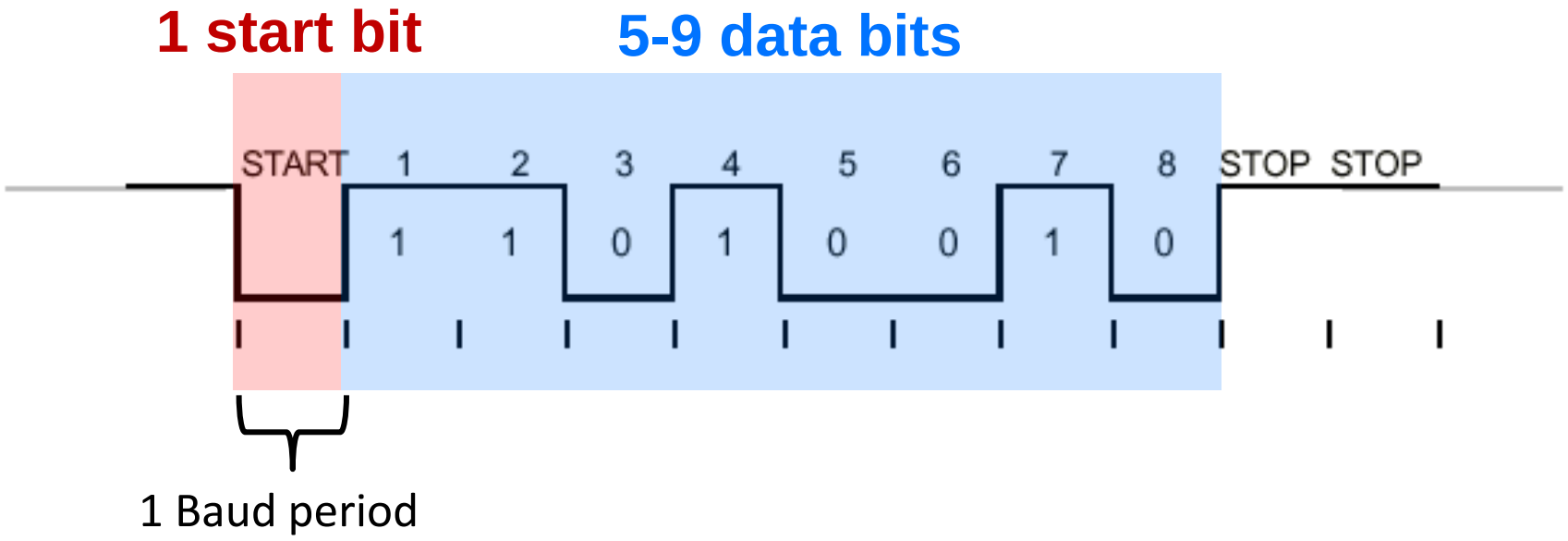
UART - Interface Protocol

- The transfer begins with a start bit:
  - the transmission line is driven to 0



UART - Interface Protocol

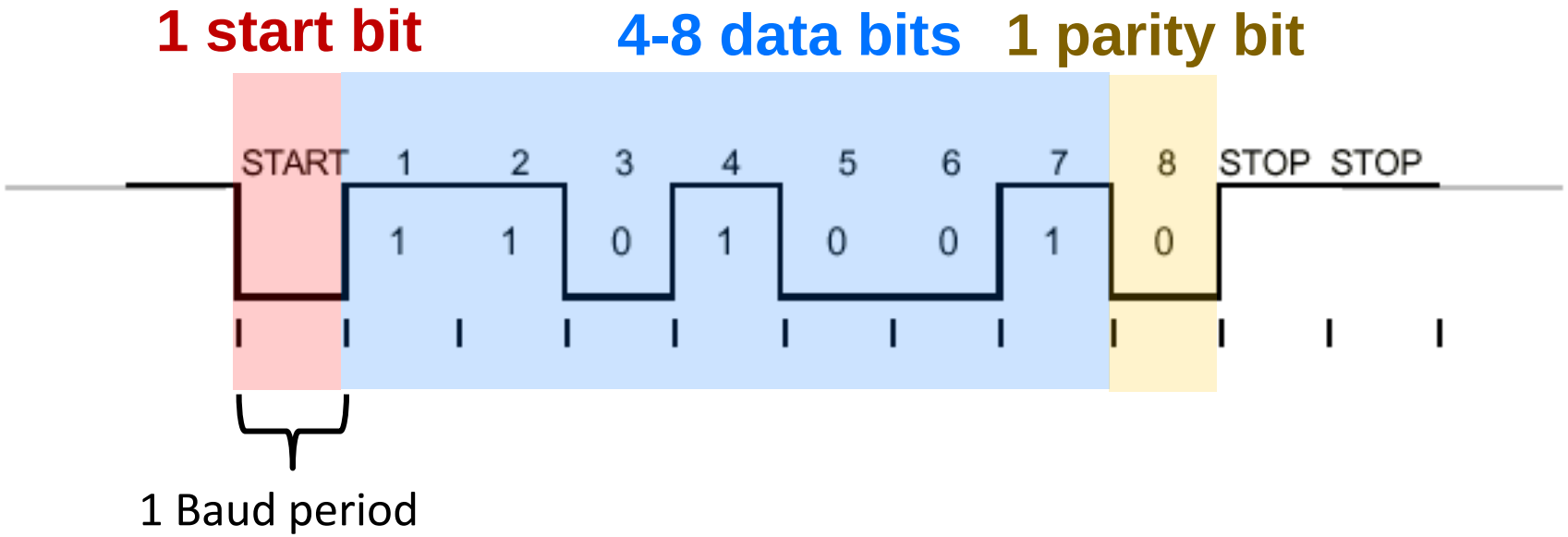
- Then, a symbol of 5 to 9 bits is transmitted:
  - most often, 8 bits (1 ASCII character)
  - the symbol size is defined by the application and known a-priori with respect to the communication



UART - Interface Protocol

One of the data bits can be used for parity (which is checked in HW):

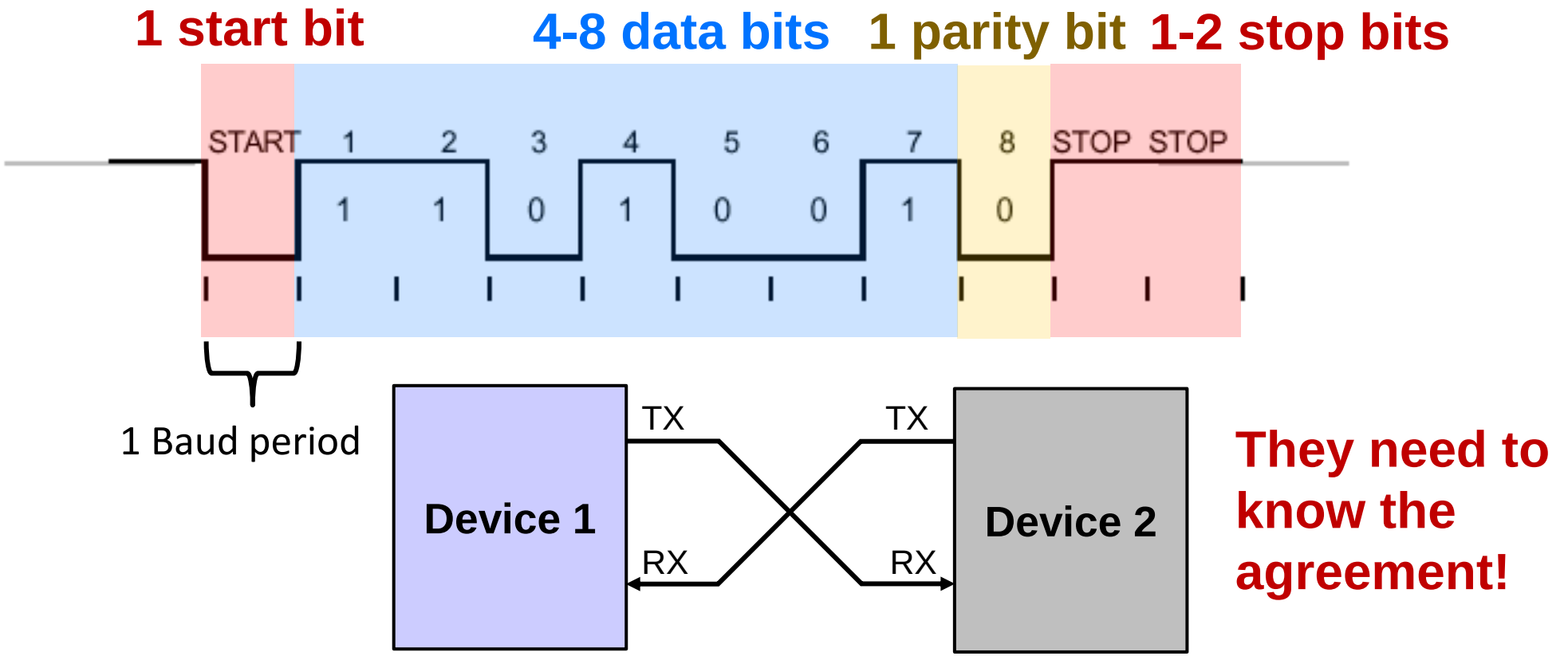
- *Even parity*: Combination of data and parity bit has even number of ones
- *Odd parity*: Combination of data and parity bit has odd number of ones
- If parity is used, 4-8 bits can be used for data



Assuming even parity, there is no parity error

UART - Interface Protocol

- Finally, stop bits:
  - Transmission line brought back to 1
  - 0.5, 1, 1.5 or 2 stop bits depending on application



They need to know the agreement!

UART - Baud Rate vs. Bit Rate

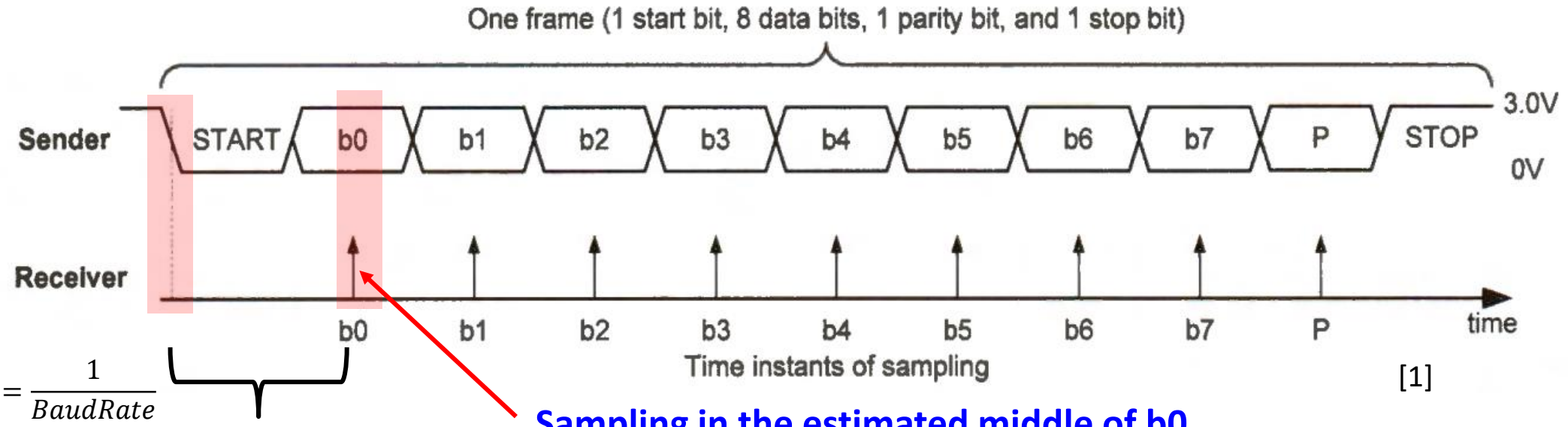
Baud rate is used in telecommunication to represent number of pulses per second.

Bit rate refers to the number of *bits of information* transmitted per second, excluding not-data bits, like start- and stop bits.

- Due to protocol overhead, the *data bit rate < baud rate*
- To be 100% clear, we **always talk of baud rate when referring to UART**
- Together with UART config (parity, stop bits etc.) one can calculate data throughput/bit rate

UART – Oversampling I

- To reliably detect pulses, the falling edge of the start bit must be detected as fast as possible.
- By sampling the Rx-line with a much higher frequency than the baud rate, the falling edge can be detected with minimal delay. The first baud is then sampled after 1.5 baud periods where its middle is estimated



[1]

$$T_{Symbol} = \frac{1}{BaudRate}$$

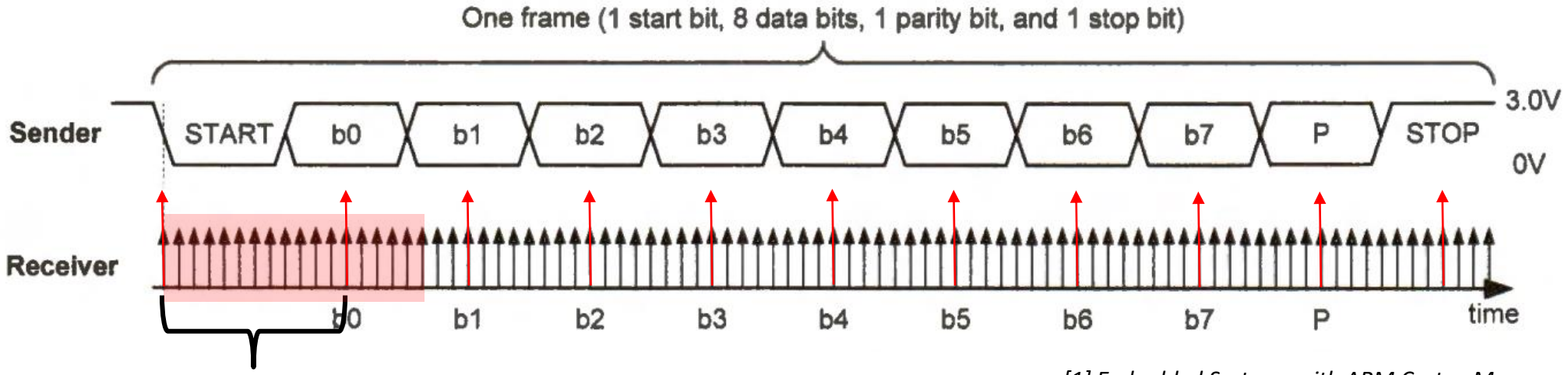
1.5 baud period after falling edge

Sampling in the estimated middle of b0

[1] Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, page 522

UART – Oversampling II

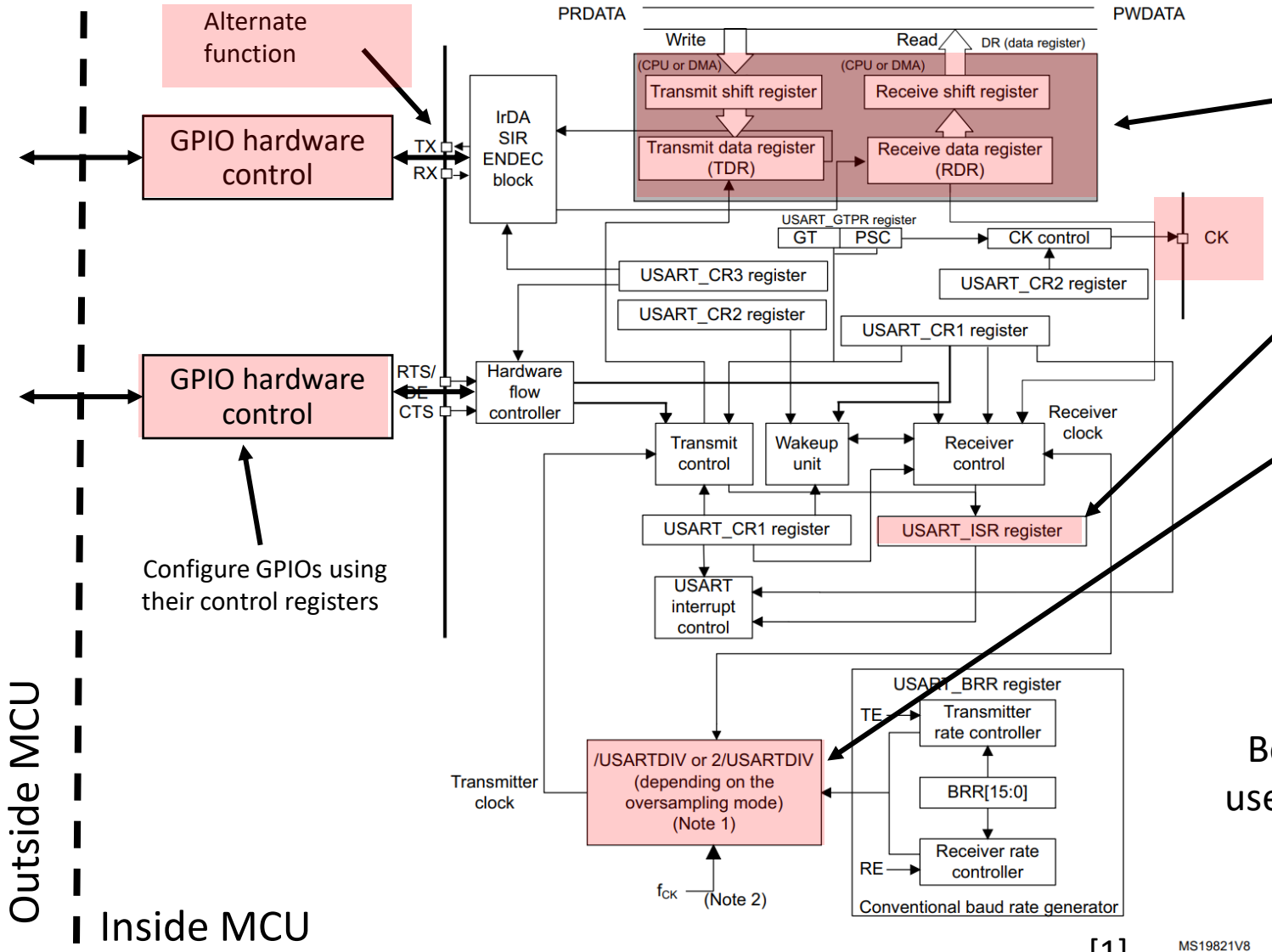
- More precisely, the receiver runs an *internal clock* whose frequency is a multiple of the baud rate and the defined *oversampling constant* (mostly 8 or 16).
- When a *Start bit* (falling edge) is detected, the receiver waits 1.5 baud (12 or 24 clock cycles, depending on the oversampling constant) before it starts with sampling.
- The receiver waits a further 8 or 16 cycles (again depending on the oversampling constant) before each consecutive baud is sampled



1.5 baud period after falling edge

[1] Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, page 522

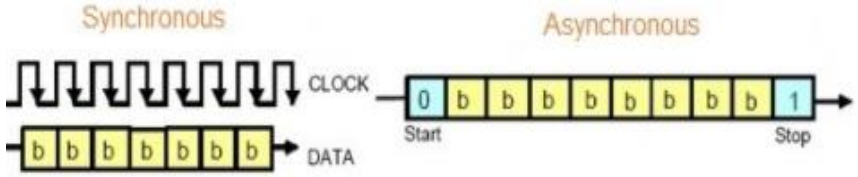
UART-USART – Hardware Peripheral



Data buffers for transmitting and receiving data

USAR Interrupt routine register to inform processor core when a transaction is finished

Define the oversampling constant



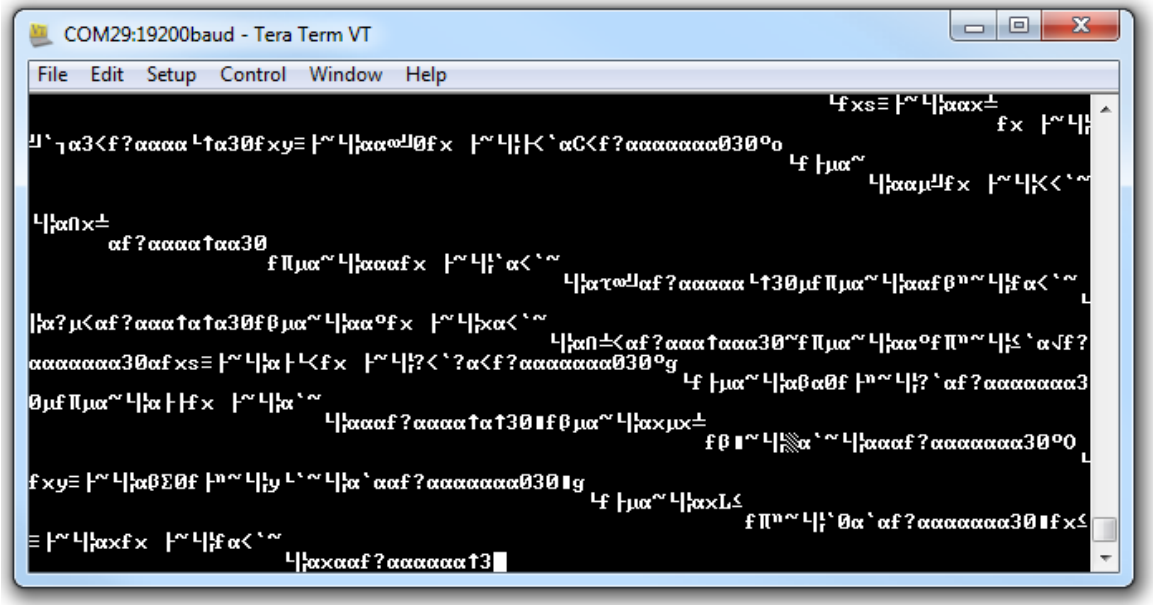
Before the UART/USART peripheral can be used, the hardware needs to be configured!

[1]

[1] RM0351 – STM32L4 Reference Manual, page 1343

Interaction: UART Oversampling

The just-opened UART communication channel looks like this. What could cause this issue?



a) Start/stop bits configured incorrectly

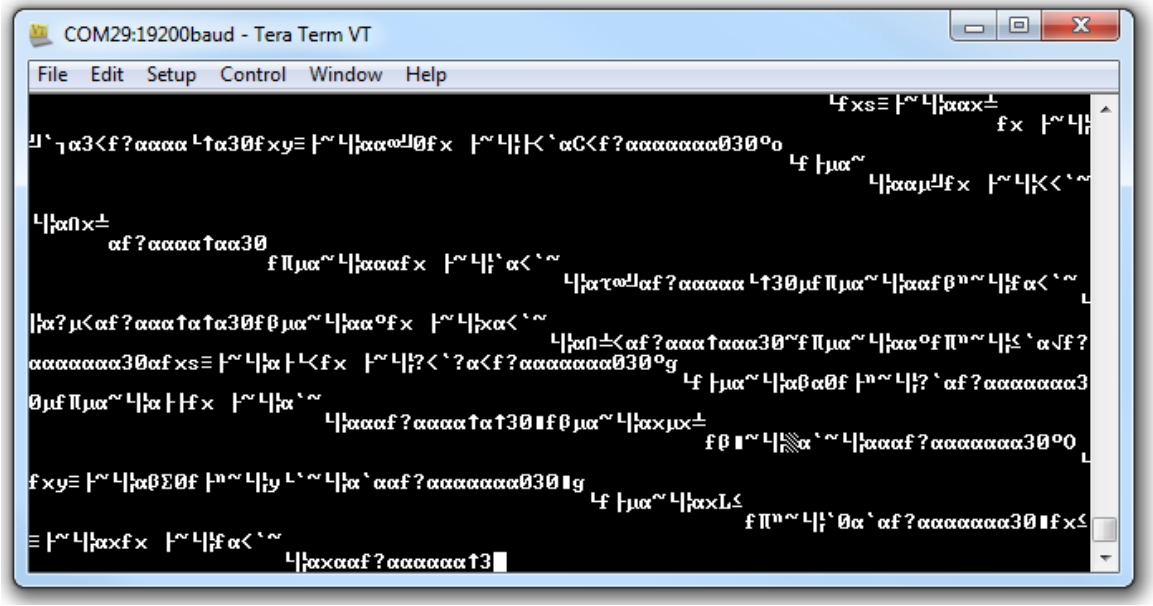
b) Parity bits configured incorrectly

c) Incorrect baud rate selected

d) Hardware clock drift

Interaction: UART Oversampling

The just opened UART communication channel looks like this. What could cause this issue?



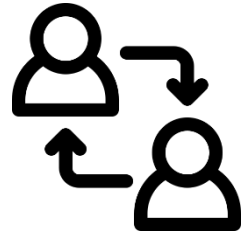
a) **Start/stop bits configured incorrectly**

b) **Parity bits configured incorrectly**

c) **Incorrect baud rate selected**

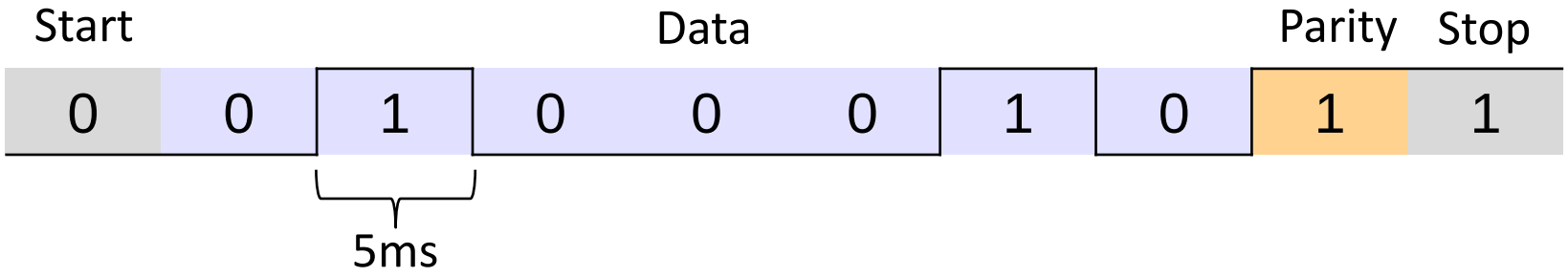
d) **Hardware clock drift**

Interaction: UART Baud Rate



In the following setup, 1 start, 7 data bits, 1 parity (odd parity) and 1 stop bit is used.

The pulse width of one pulse is 5 ms. Answer the following questions:

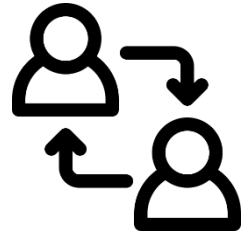


1. Did this message cause a parity error?

2. What is the baud rate?

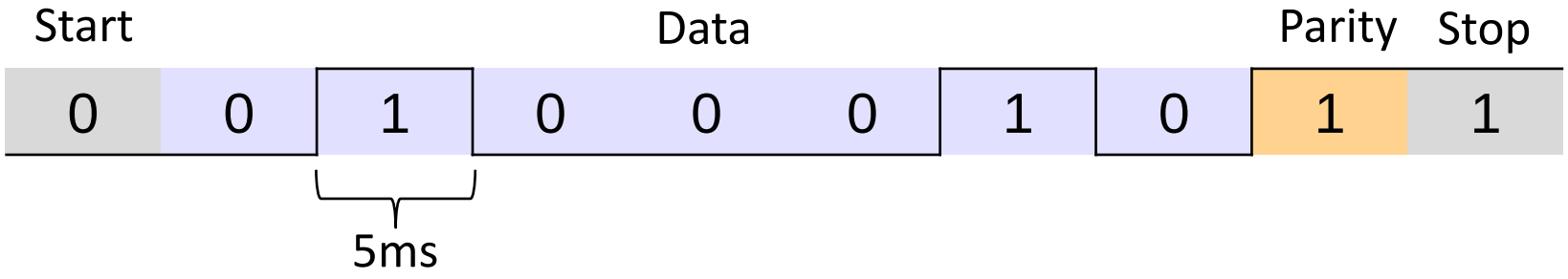
3. How long does it take to send 1000 data bits?

Interaction: UART Baud Rate



In the following setup, 1 start, 7 data bits, 1 odd parity bit and 1 stop bit is used.

The pulse width of one pulse is 5 ms. Answer the following questions:



1. Did this message cause a parity error?

   No: Message has two 1's so parity bit is set to 1

2. What is the baud rate?

   1/5ms = 200 baud/s

3. How long does it take to send 1000 data bits?

   Number of frames: 1000/7=142.8 → 143 frames

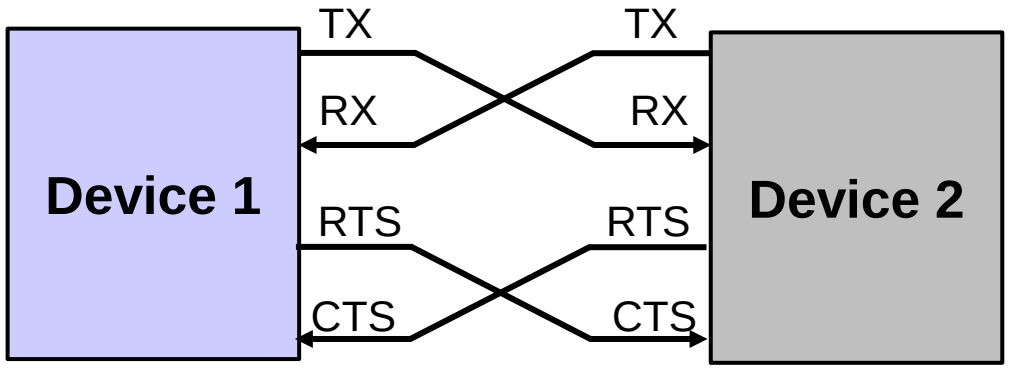
   Number of pulses 143*10=1430 → Time: 1430 * 5ms =7.15s

UART - Hardware Flow control

RTS: Request To Send **CTS**: Clear To Send

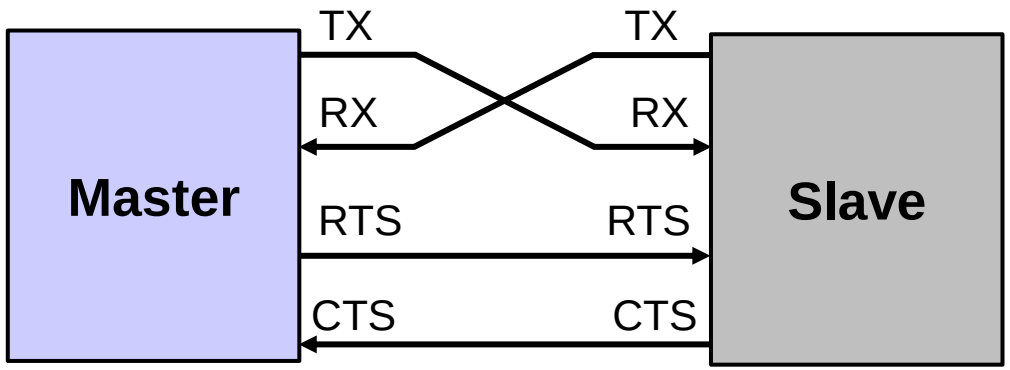
Bidirectional Flow control

- RTS: device signals if it is ready to accept new data
- CTS: device detects if other device is ready to accept data
- Before the receive buffer is full, the receiving device de-asserts RTS



Unidirectional Flow control

- RTS: Master asserts line if it wants to send data
- CTS: Slave responds by asserting CTS if ready to receive
- Transmission happens until the slave de-asserts CTS, indicating that it needs a temporary halt in transmission



I^2C – Inter-Integrated Circuit Bus

I^2C – Inter-Integrated Circuit Bus

Inter-Integrated Circuit (I^2C), usually pronounced "*I-Squared-C*", has been introduced by Philips (now NXP Semiconductors) in 1982.

It is used for *communication* with external peripherals, for example:

- EEPROMs (Memory)
- Thermal sensors
- Real-time clocks

Also used as a *control interface* for signal processing devices with separate data interfaces, for example:

- Radio frequency tuners
- Video decoders and encoders (HDMI)
- Audio processors

I²C – Topology

SDA and SCL internally configured as **open-drain**

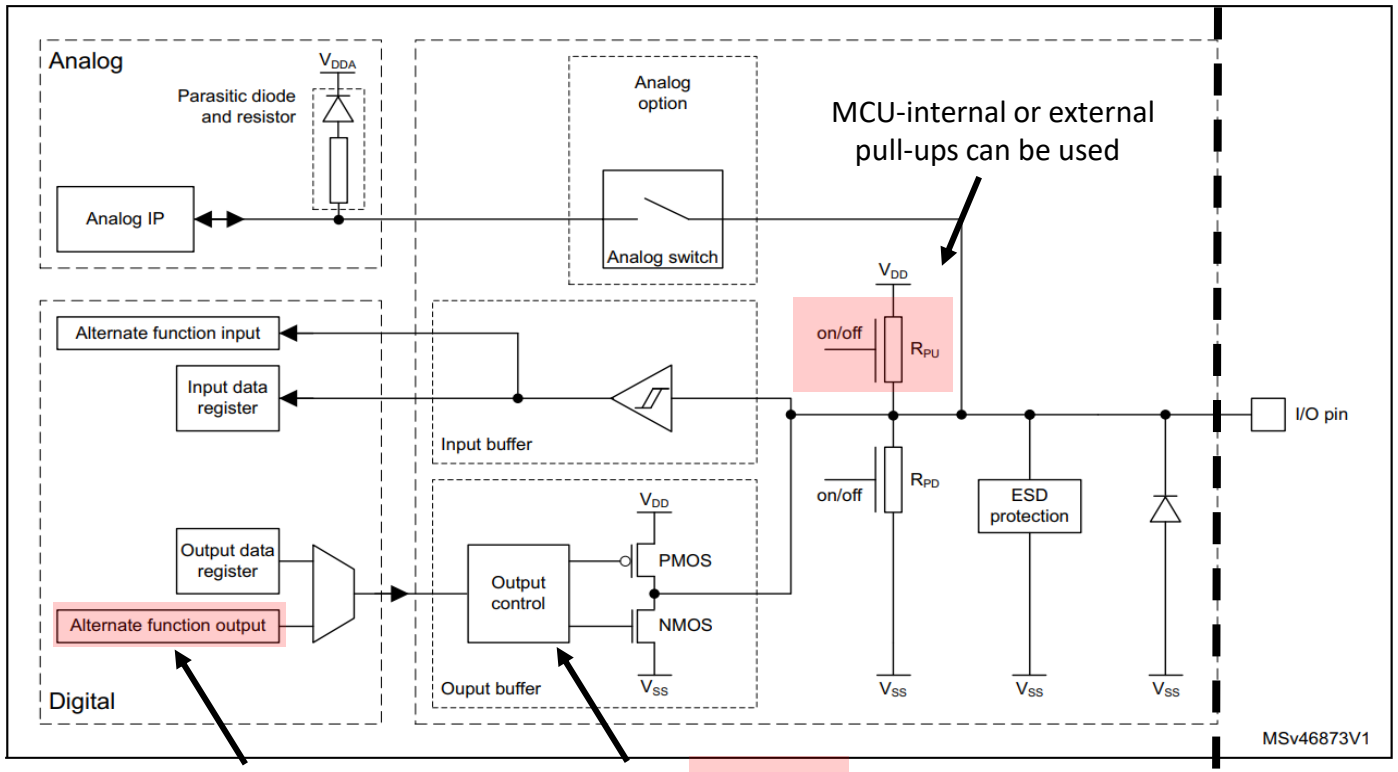
(Lecture 2 GPIO Output Modes)

SCL: Serial Clock

Master provides the clock signal

SDA: Serial Data

Data Line

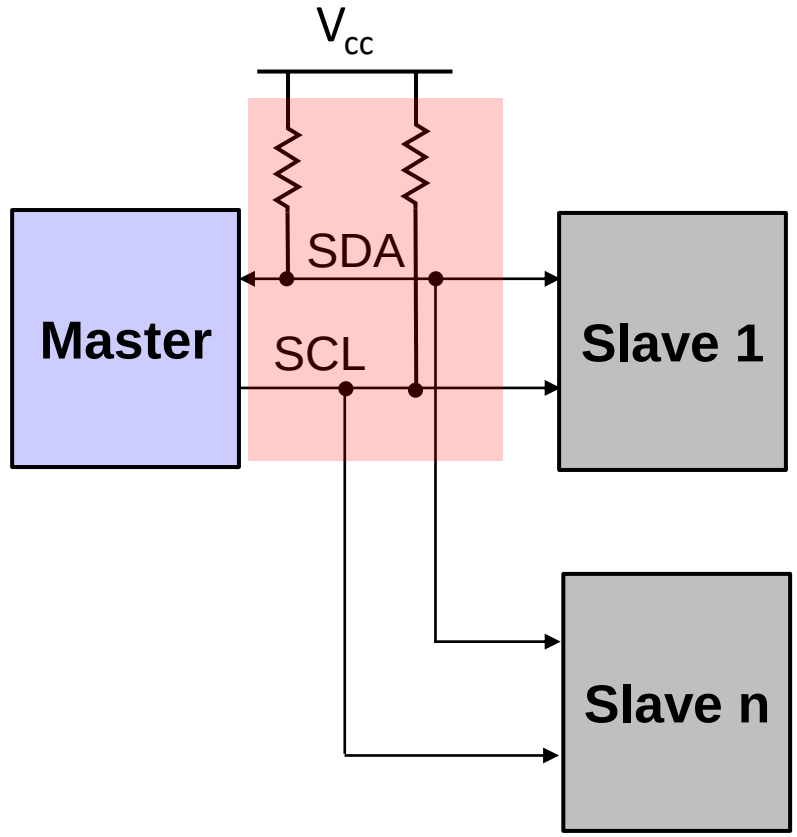


Configure GPIO to open-drain using control registers

Push-Pull, Open-Drain or disabled

Inside MCU

Outside MCU



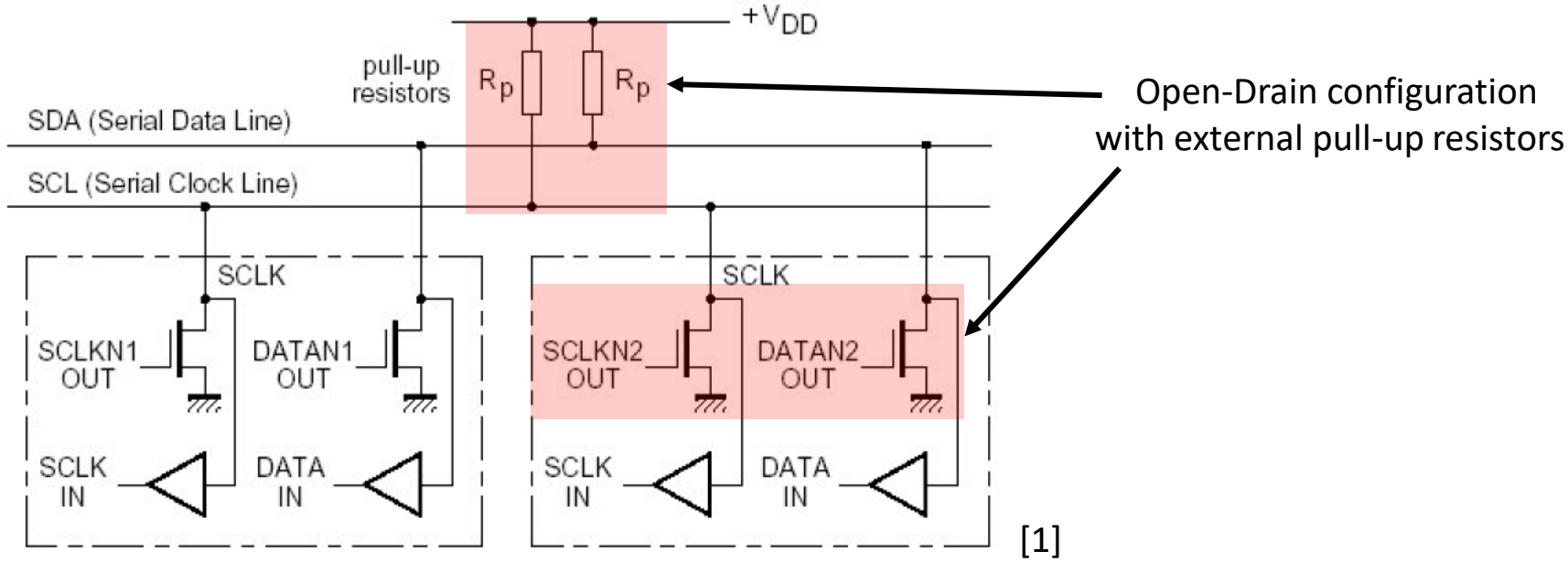
I²C – Topology

SCL: Serial Clock

Master provides the clock signal

SDA: Serial Data

Data Line



Lines are *pulled-up by default* using resistors, and *pulled-down* by open-drain drivers *during communication.*

- Wired-AND: if *any* driver is active, the line is low (*Lecture 2 GPIO Output Modes)* otherwise it is high.
- Any module on the bus can act as master, slave or both. However, the MCU typically acts as master and other peripherals (e.g. sensors) as slaves.

[1] THE I2C-BUS SPECIFICATION VERSION 2.1

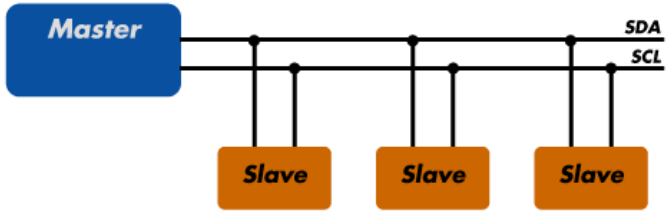
I^2C – Transfer Formats

The I2C bus specification support three transfer formats:

- *I^2C Write* where the master transmits data to the slave
- *I^2C Read* where the master reads from the slave immediately after the first byte
- *I^2C Combined* format where the write and read transactions are combined

I²C – Interface Protocol – Write

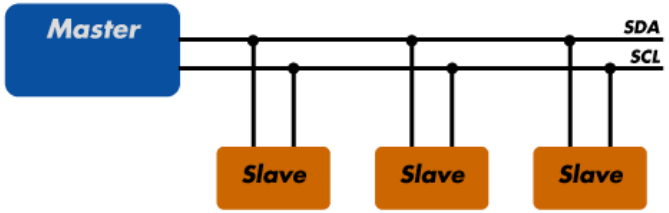
In idle state, both *SCL* and *SDA* are pulled-up by the resistors

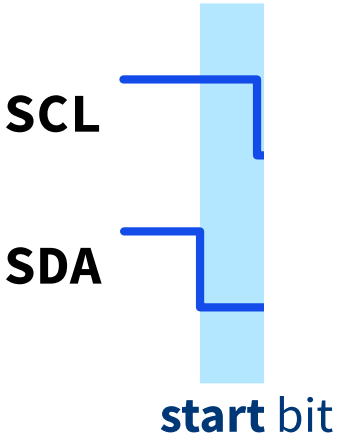


SCL ——

SDA ——

I²C – Interface Protocol – Write





Master

Slave

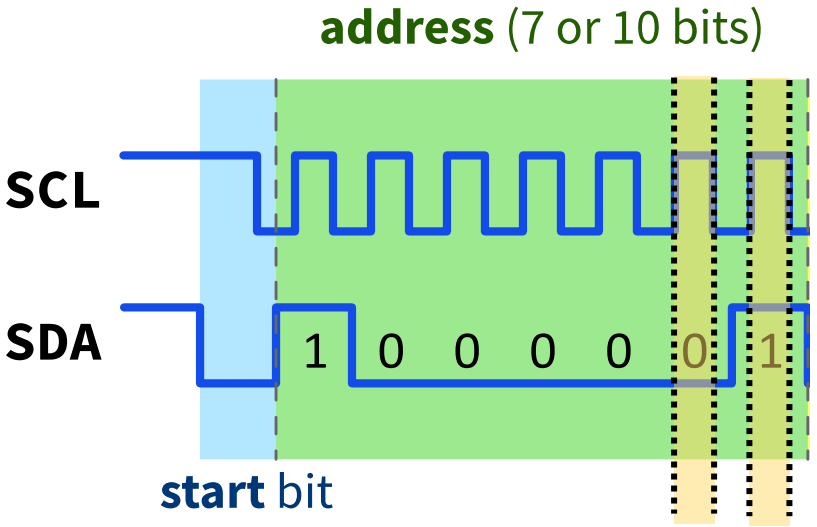
To start the communication, the **master**:

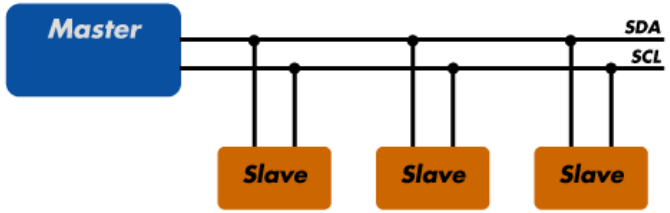
- Sends a start condition (**SDA** 1→0 transition while **SCL** is **still 1**)
- Then, it starts generating the **SCL** clock

Except for the start and stop bits, **SDA** transitions *only* when **SCL** is **0**

SDA must be held **stable** when **SCL** is **high** (1)

I²C – Interface Protocol – Write





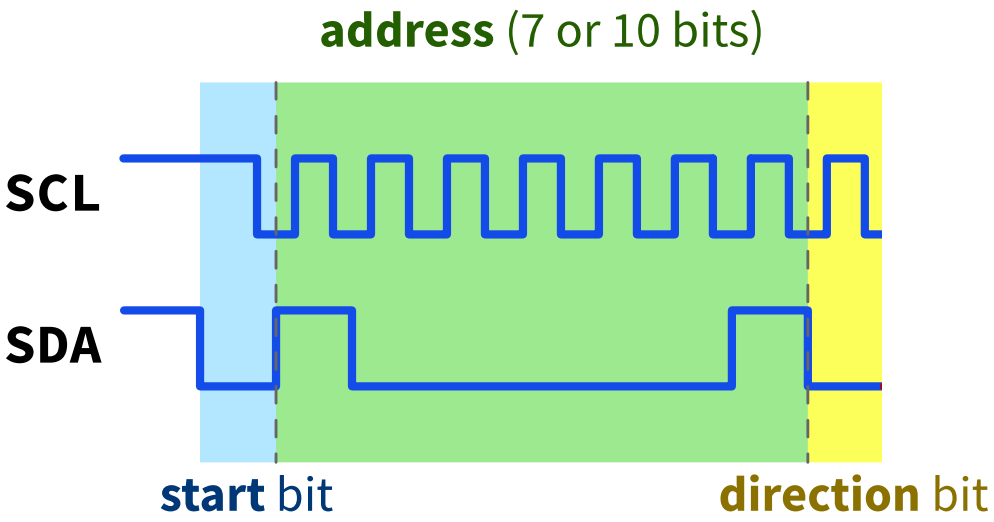
Master

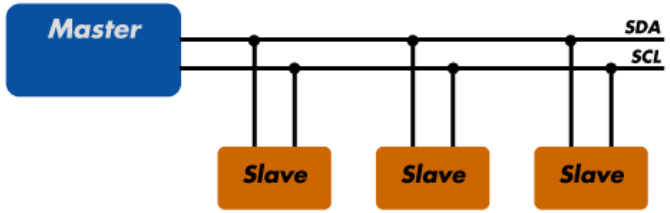
Slave

The **master** transmits the **slave** **address**:

- **Broadcasted** to all devices on the I²C bus to select the target **slave**
- Either **7 bits** or **10 bits** (10 bits for newer devices – 7 bits address space is small!)
- MSB is sent first: In this example, the address is 1000001
- **SDA** must be held **stable** when **SCL is high** (1)

I²C – Interface Protocol – Write





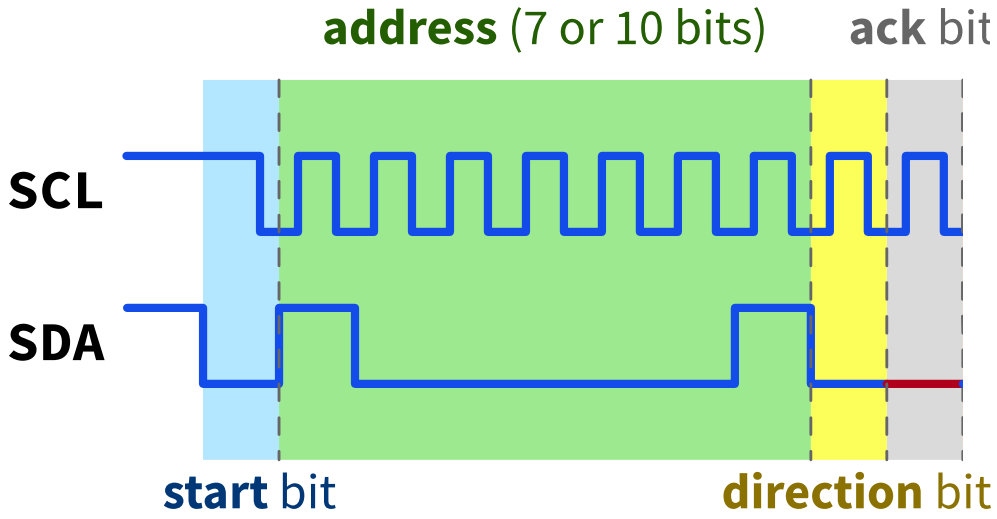
Master

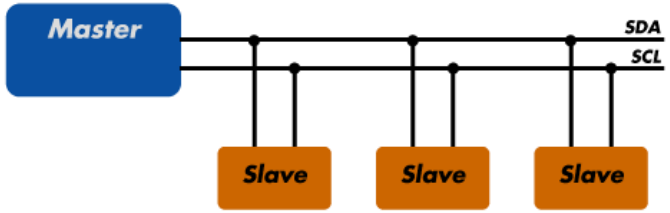
Slave

The **master** transmits a **direction** bit:

- A **0** for **master → slave** (write) transfer
- A **1** for **slave → master** (read) transfer
- *In this example, it is a write transfer (direction bit is 0)*

I²C – Interface Protocol – Write





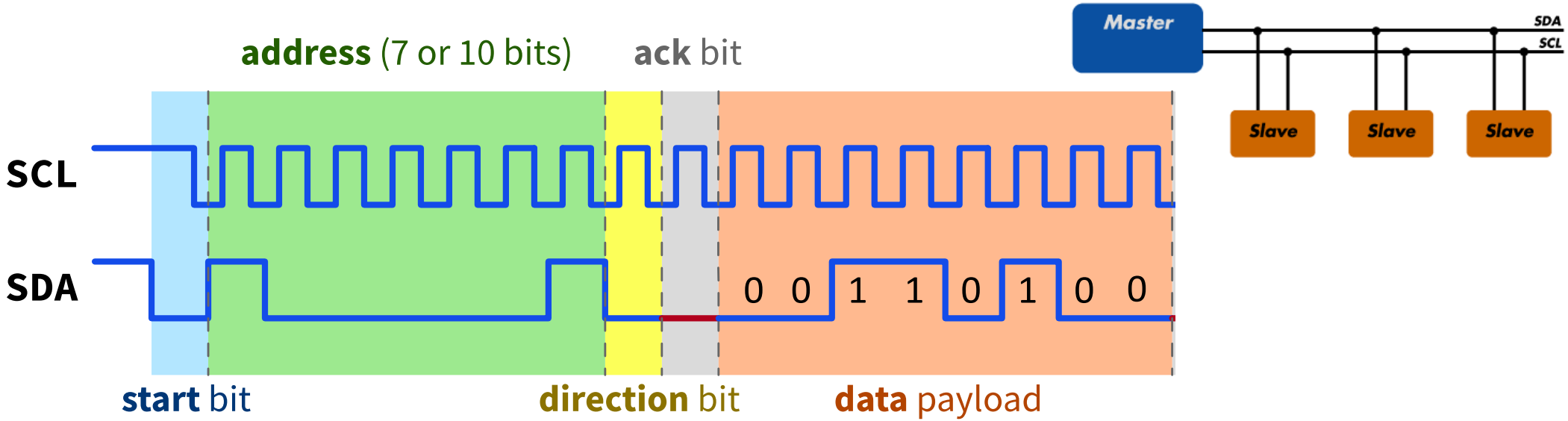
Master

Slave

The **slave** then **acknowledges** reception:

- Master does nothing so **SDA** would be **1** due to **pull-up configuration**
- Slave drives **SDA** to **0**
- If not acknowledged, the transaction is treated as invalid

I²C – Interface Protocol – Write



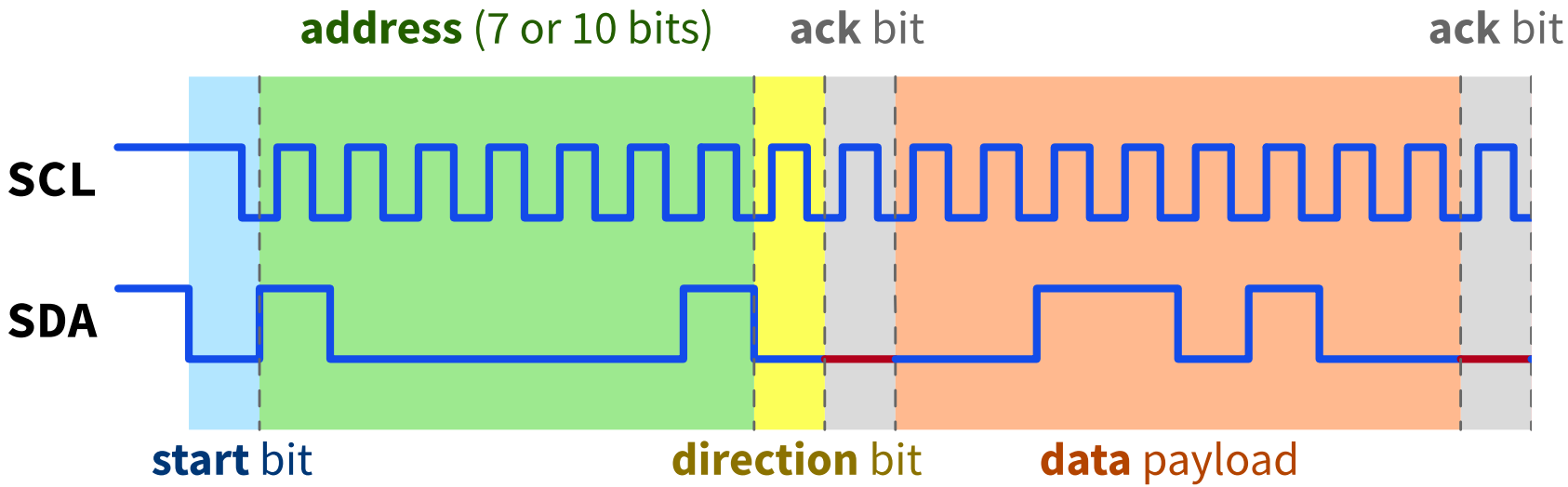
Master

Slave

The **master** transmits its data payload:

- Each payload packet is **8 bits**
- There might be more than one packet
- *In this example, the data payload is 8'b00110100*

I²C – Interface Protocol – Write



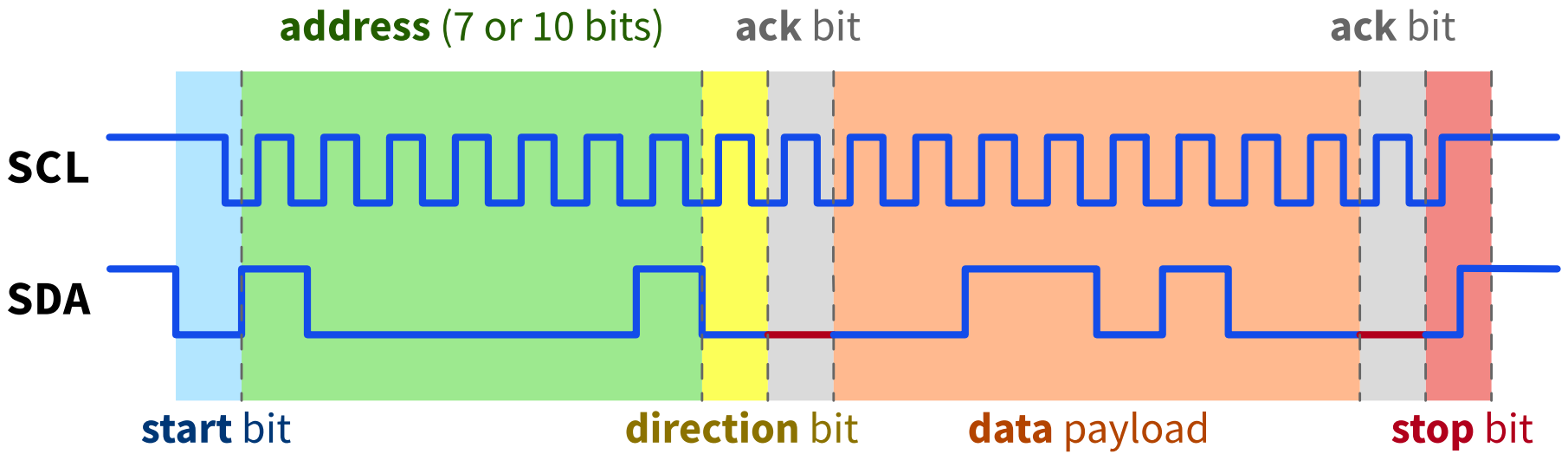
Master

Slave

The **slave** acknowledges reception of the data packet:

- **1 ACK bit** every **8 payload bits**
- Slave must acknowledge each packet
- Slave drives **SDA** to **0**

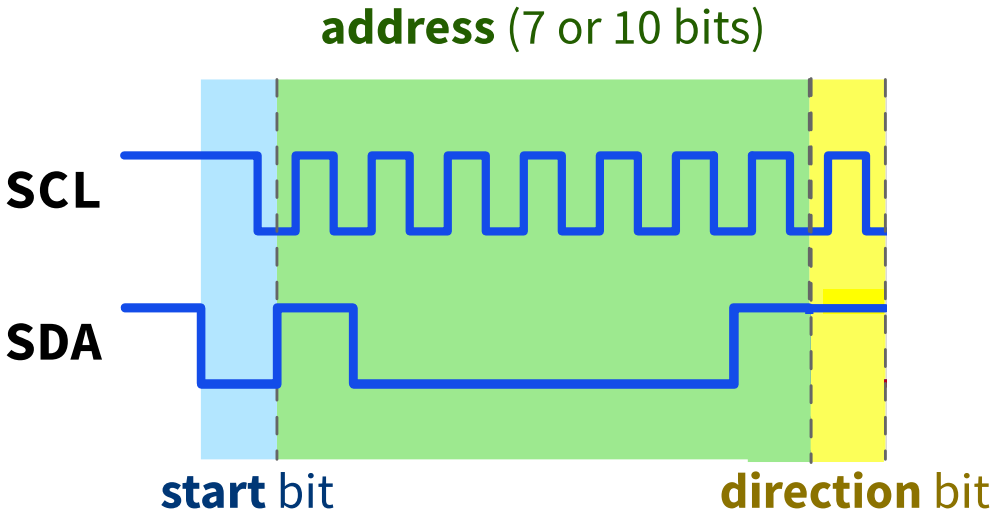
I²C – Interface Protocol – Write

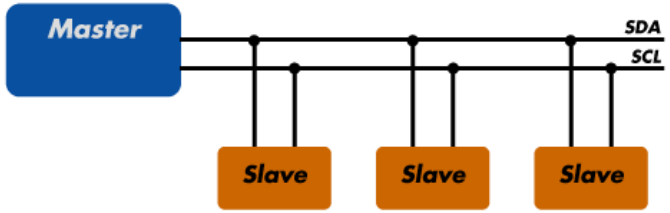


At the end of the transfer, the **master** generates a **stop condition**:

- First, it sets **SDA** to **0**
- Then it releases **SCL** (i.e. it lets it go to **1**)
- Finally, it releases **SDA** which also goes to **1**
- Except for the start and stop bits, **SDA** transitions *only* when **SCL** is **0**

I²C – Interface Protocol – Read





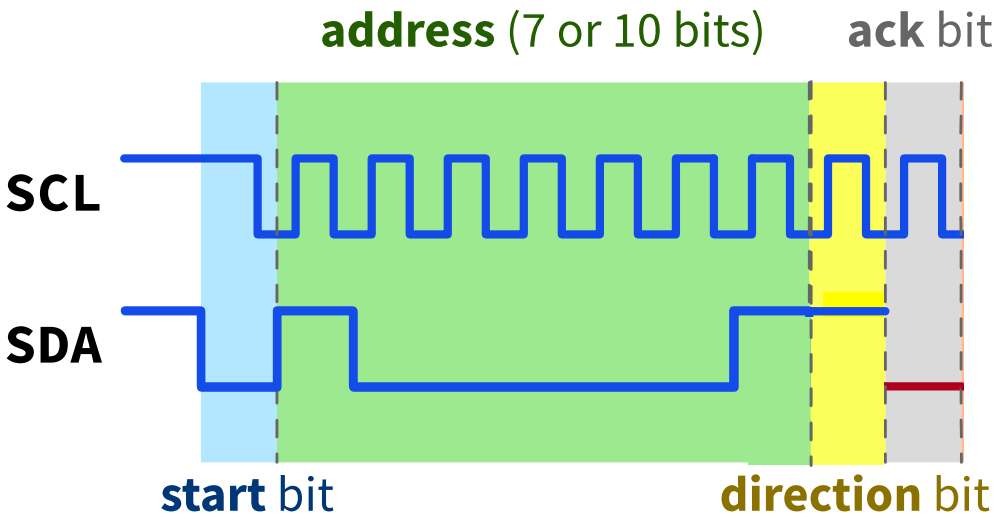
Master

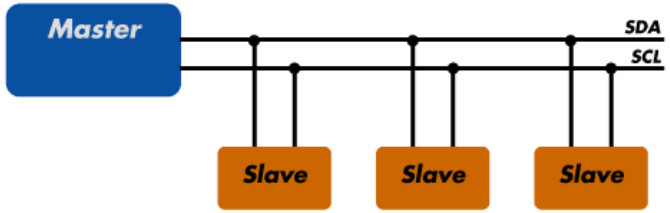
Slave

Reads work similarly, but the ack roles during data transfer are reversed:

- Start bit and address are the same as in the write procedure
- Direction bit is now 1 instead of 0

I²C – Interface Protocol – Read





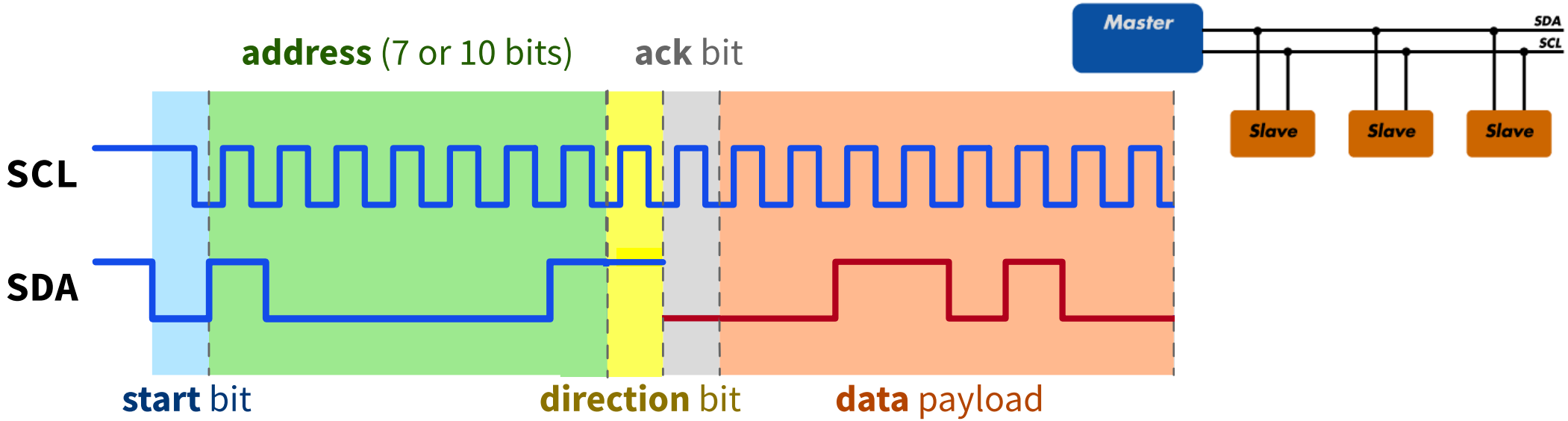
Master

Slave

Reads work similarly, but data transfer – ack roles are reversed:

- Start bit and address are the same as in the write procedure
- Direction bit is now 1 instead of 0

I²C – Interface Protocol – Read



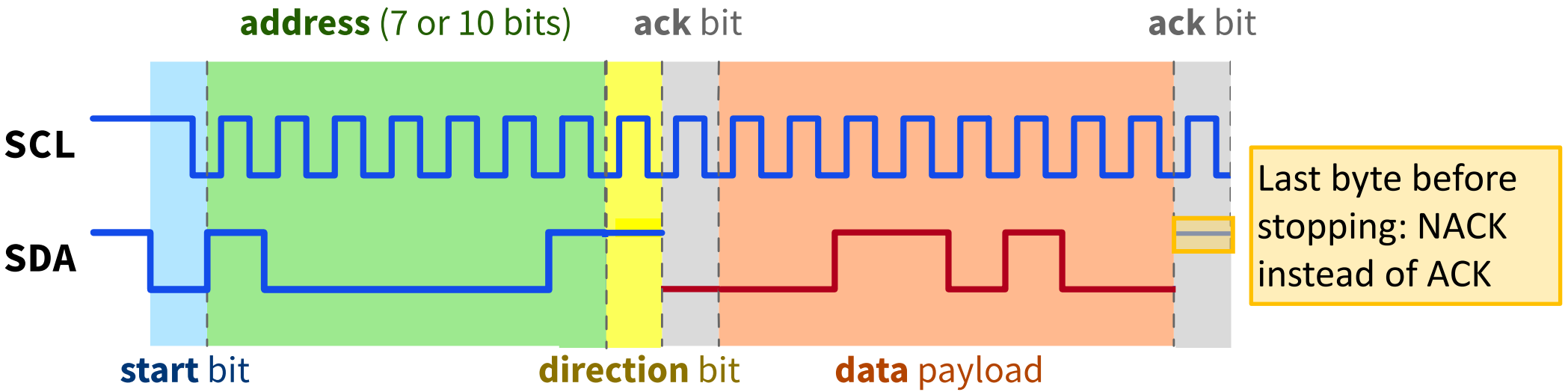
Master

Slave

Reads work similarly, but data transfer – ack roles are reversed:

- Start bit and address are the same as in the write procedure
- Direction bit is now 1 instead of 0
- The **slave** drives **SDA** when transmitting the data byte

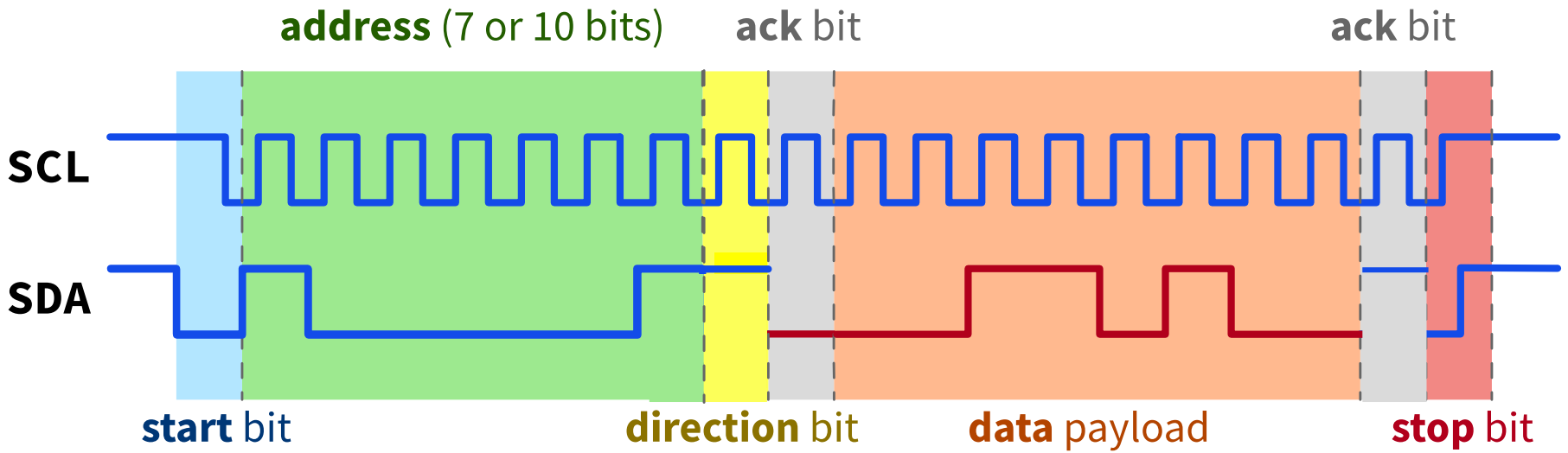
I²C – Interface Protocol – Read



Reads work similarly, but data transfer – ack roles are reversed:

- Start bit and address are the same as in the write procedure
- Direction bit is now 1 instead of 0
- The **slave** drives **SDA** when transmitting the data byte
- The **master** acknowledges the transfer of each received byte except the last one

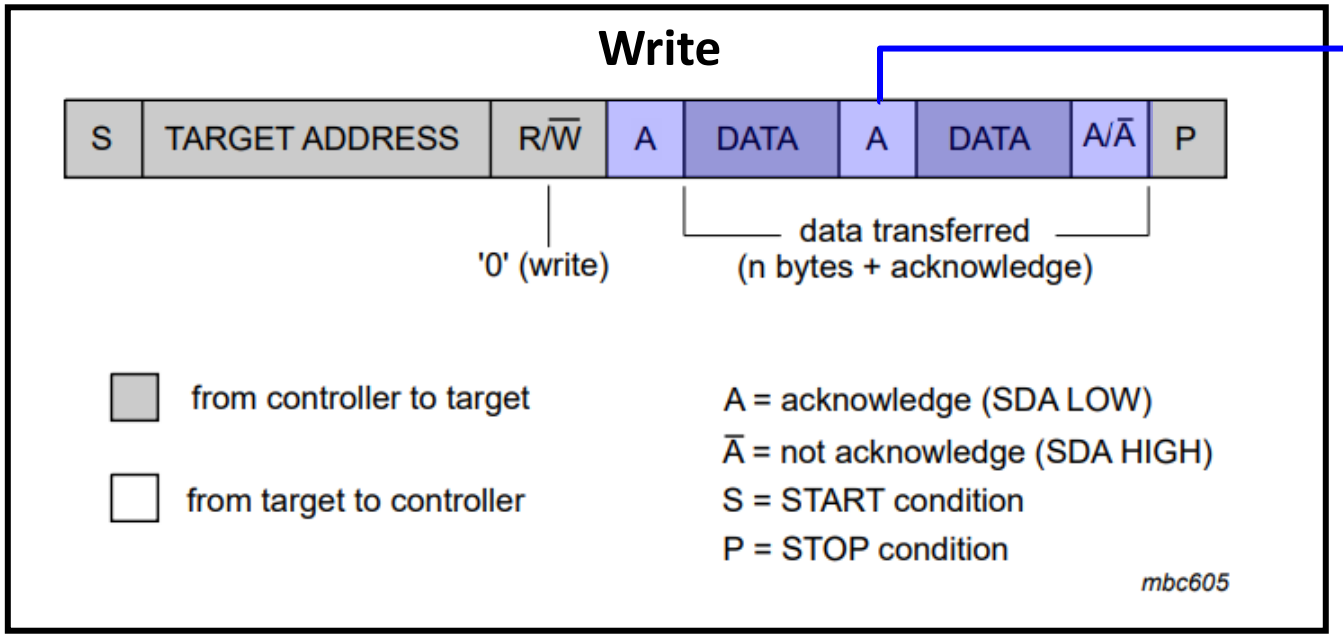
I²C – Interface Protocol – Read



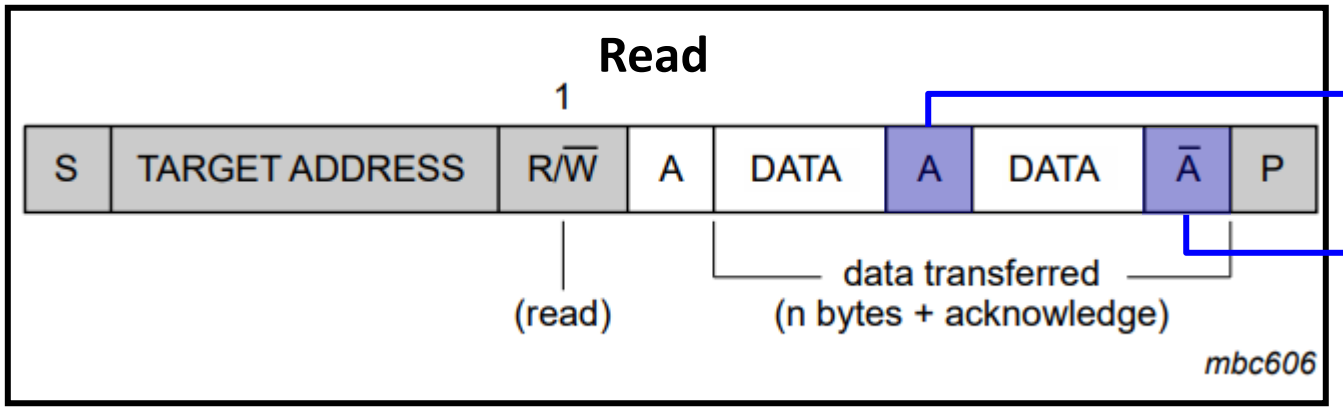
Reads work similarly, but data transfer – ack roles are reversed:

- Start bit and address are the same as in the write procedure
- Direction bit is now 1 instead of 0
- The **slave** drives **SDA** when transmitting the data byte
- The **master** acknowledges the transfer of each received byte except the last one
- The **master** stops communication

I^2C – Interface Protocol Read/Write Summarized



Writing multiple bytes after each other



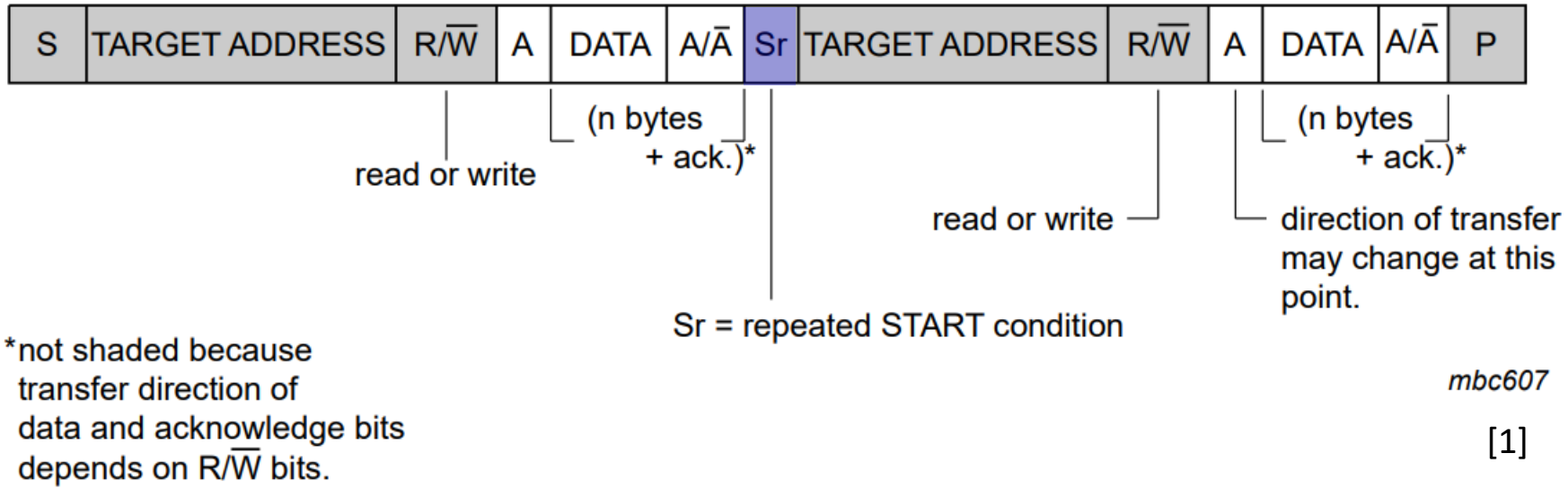
Ack for every packet

Except last one

[1]

I²C – Combined Format

- Multiple read and write combined by not stopping the communication but restarting it with the *repeated start condition*
- Afterwards it follows the same protocol as if there would have been a normal start condition
- This way, in multi-master setup another master can not interrupt the ongoing communication

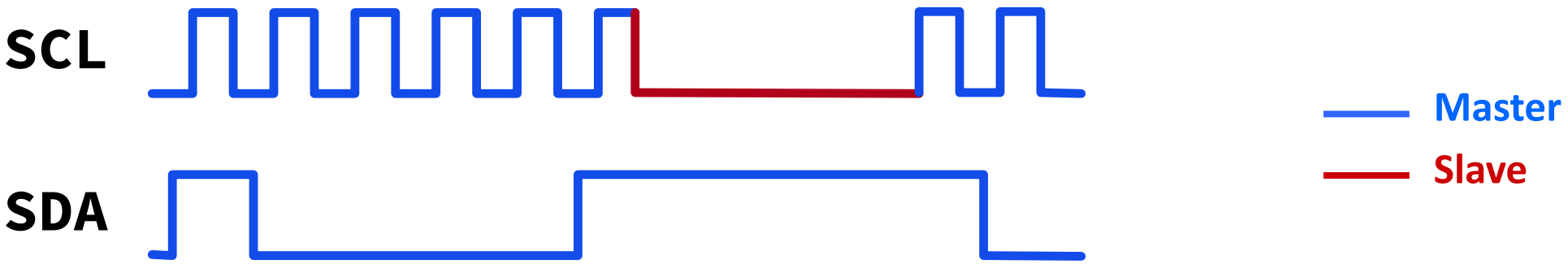


[1]

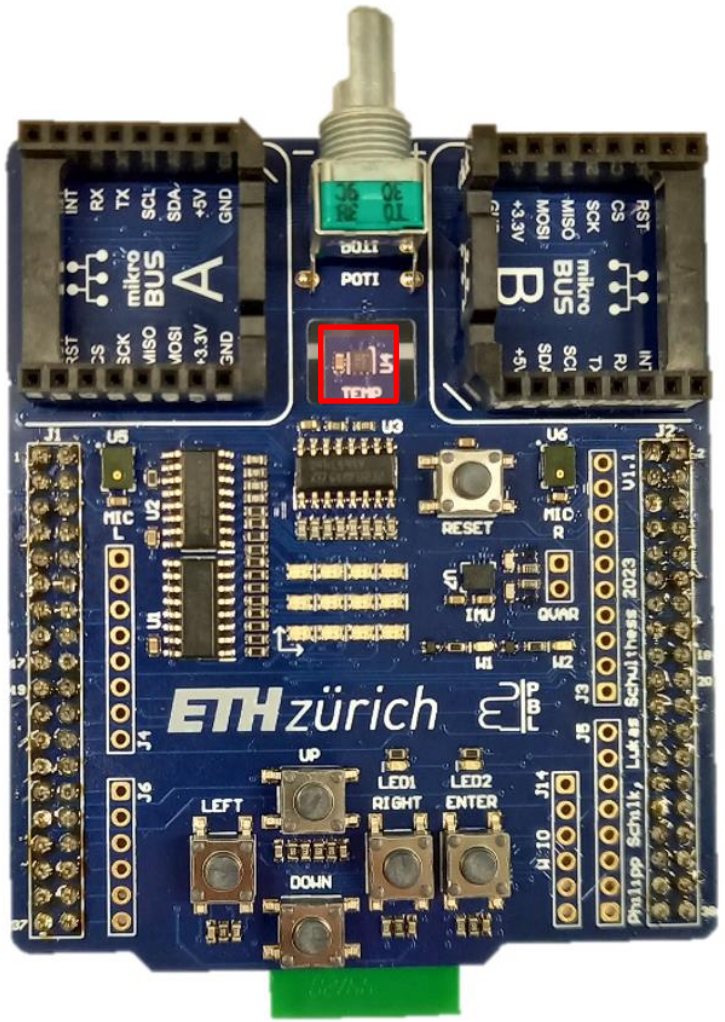
I^2C – Clock Stretching

The **Slave** can ask for more time to process a bit by *clock stretching*:

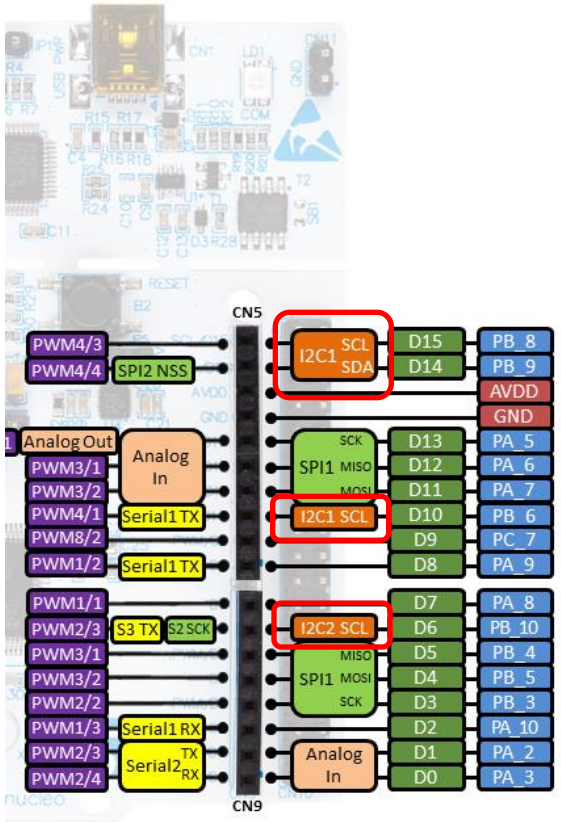
- Drive **SCL** to **0**
- **Master can't** drive **SCL** to **1** as I^2C uses open-drain



I²C on Sensor Shield



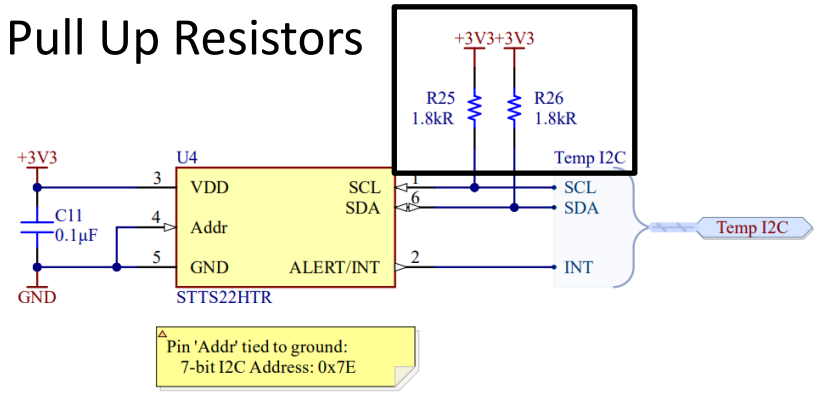
Embedded systems sensor shield



Left Side of Nucleo-L476RG

Temperature Sensor

Pull Up Resistors



I²C – Hardware Peripheral

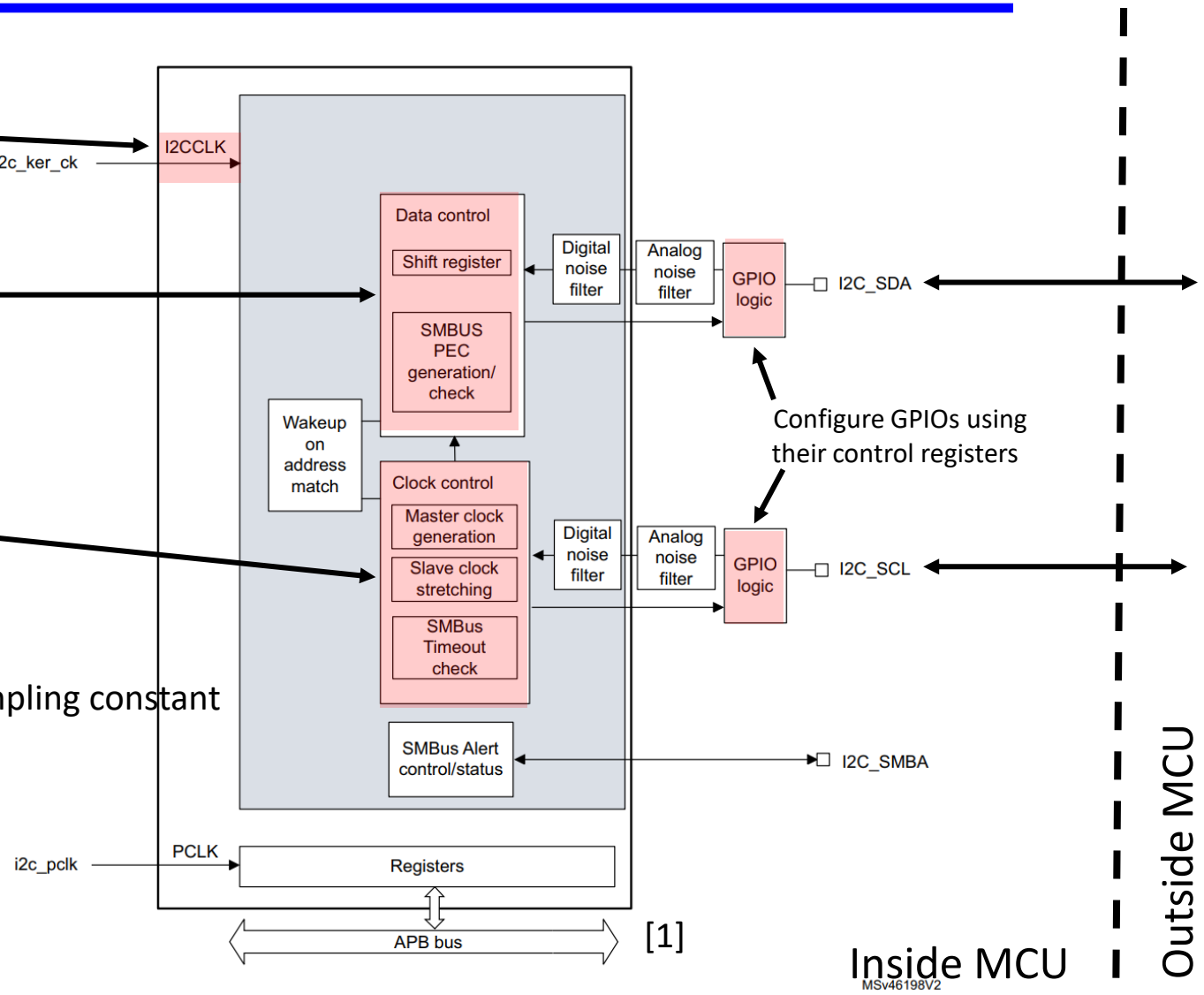
I2C peripheral clock coming from the microcontroller's clock tree

Configure the data control registers on order to receive transmit the correct amount of data

Configure the SCL (Serial Clock) to get the needed clock speed

Define the oversampling constant

Before the I²C peripheral can be used, the hardware needs to be configured!



[1]

[1] RM0351 – STM32L4 Reference Manual, page 1273

I²C – Temperature Sensor – STTS22H

Datasheet of sensor might have an additional protocol on top of I²C to specify the register access

I²C read and write sequences

The previous sequences are used to implement the actual write and read sequences described in the tables below.

Transfer when the master is writing one byte to the slave:

Master	ST	SAD+W		SUB		DATA		SP
Slave			SAK		SAK		SAK	

Transfer when the master is writing multiple bytes to the slave:

Master	ST	SAD+W		SUB		DATA		DATA		SP
Slave			SAK		SAK		SAK		SAK	

Transfer when the master is receiving (reading) one byte of data from the slave:

Master	ST	SAD+W		SUB		SR	SAD+R			NMAK	SP
Slave			SAK		SAK			SAK	DATA		

Transfer when the master is receiving (reading) multiple bytes of data from the slave:

Master	ST	SAD+W		SUB		SR	SAD+R			MAK		MAK		NMAK	SP
Slave			SAK		SAK			SAK	DATA		DATA		DATA		

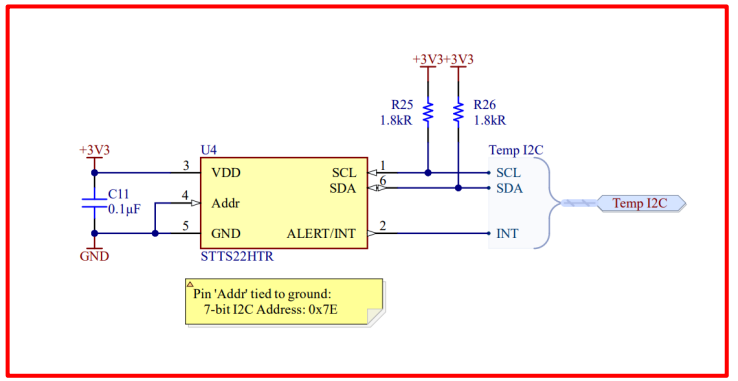


Table 6. **SMBus WRITE protocol**

Start	Slave address	WR	ACK	Register address	ACK	data	ACK	stop
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	8 bits	1 bit	1 bit

[1] Datasheet - STTS22H , page 12

I²C – Temperature Sensor – STTS22H

Inside the sensor there are different registers one wants to read or write: Configuration, Status, Temperature, Max./Min. Temp

- How to access them? Registers have an 8-bit address

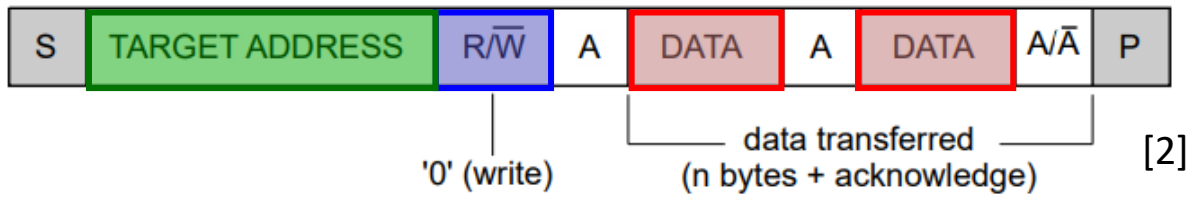
ST	Start
SAD	Slave Address
W	Write
SAK	Slave Ack
SUB	**Register** Sub Address
SP	Stop

Transfer when the master is writing one byte to slave:

Master	ST	SAD+W		SUB		DATA		SP
Slave			SAK		SAK		SAK	

[1]

Still follows standard protocol, but first data packet is treated as sub address



[2]

Table 12. **Register map**

Addr	Type[1]	Name	7	6	5	4	3	2	1	0	Default
01h	RO	WHOAMI	whoami7	whoami6	whoami5	whoami4	whoami3	whoami2	whoami1	whoami0	A0h
02h	RW	TEMP_H_LIMIT	THL7	THL6	THL5	THL4	THL3	THL2	THL1	THL0	00h
03h	RW	TEMP_L_LIMIT	TLL7	TLL6	TLL5	TLL4	TLL3	TLL2	TLL1	TLL0	00h
04h	RW	CTRL	LOW_ODR_START	BDU	AVG1	AVG0	IF_ADD_INC	FREERUN	TIME_OUT_DIS	ONE_SHOT	00h
05h	RO	STATUS	0	0	0	0	0	UNDER_THL	OVER_THH	BUSY	output
06h	RO	TEMP_L_OUT	T7	T6	T5	T4	T3	T2	T1	T0	output
07h	RO	TEMP_H_OUT	T15	T14	T13	T12	T11	T10	T9	T8	output

1. *RW designates a read/write register. RO designates a read-only register*

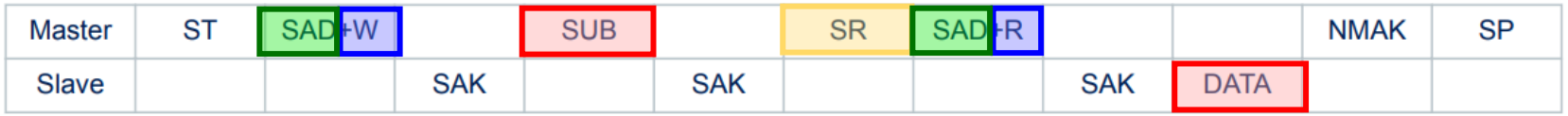
[1] Datasheet - STTS22H , page 12
[2] THE I2C-BUS SPECIFICATION VERSION 2.1

I²C – Temperature Sensor – STTS22H

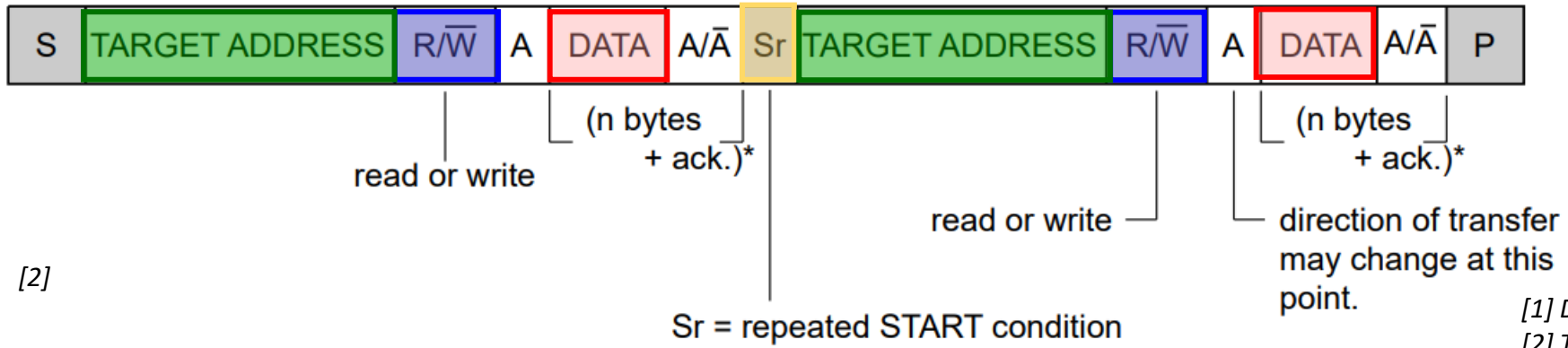
How to read the registers? First do a write informing the sensor which registers should be returned in the next read

Combined Format

Transfer when master is receiving (reading) one byte of data from slave:



[1]



[2]

ST	Start
SAD	Slave Address
W	Write
SAK	Slave Ack
SUB	Register Sub Address
SP	Stop
SR	Repeated Start

Table 12. Register map

Addr	Type[1]	Name	7	6	5	4	3	2	1	0	Default
01h	RO	WHOAMI	whoami7	whoami6	whoami5	whoami4	whoami3	whoami2	whoami1	whoami0	A0h
02h	RW	TEMP_H_LIMIT	THL7	THL6	THL5	THL4	THL3	THL2	THL1	THL0	00h
03h	RW	TEMP_L_LIMIT	TLL7	TLL6	TLL5	TLL4	TLL3	TLL2	TLL1	TLL0	00h
04h	RW	CTRL	LOW_ODR_START	BDU	AVG1	AVG0	IF_ADD_INC	FREERUN	TIME_OUT_DIS	ONE_SHOT	00h
05h	RO	STATUS	0	0	0	0	0	UNDER_THL	OVER_THH	BUSY	output
06h	RO	TEMP_L_OUT	T7	T6	T5	T4	T3	T2	T1	T0	output
07h	RO	TEMP_H_OUT	T15	T14	T13	T12	T11	T10	T9	T8	output

1. RW designates a read/write register. RO designates a read-only register

[1] Datasheet - STTS22H , page 12
[2] THE I2C-BUS SPECIFICATION VERSION 2.1

I^2C - Protocol

- I^2C supports different speed modes :
  - Standard (up to 100 Kbps), fast (up to 400 Kbps), fast + (up to 1Mbps),

    high-speed (up to 3.4 Mbps)
  - Trace capacitances, trace length, and pull-up resistor value are important factors when high speed needs to be reached!
- Data Overhead: 1 bit ACK for 8bit of data plus address
- Can support multi-master mode (not discussed)
  - For complex applications
  - Communication is always started by a master, both in single-master and multi-master mode
- Half-duplex synchronous communication scheme

Interaction: I²C Bit Sequence

You want to read the temperature from one of three temperature sensors. Internally, the temperature sensors have two 8-bit registers:

- 0x00: temperature register, represented in two's complement
- 0x01: configuration register

What is the bit sequence of SDA for a temperature read of TC74A1-3.3VCT Assume the temperature sensor has sensed -10° C?

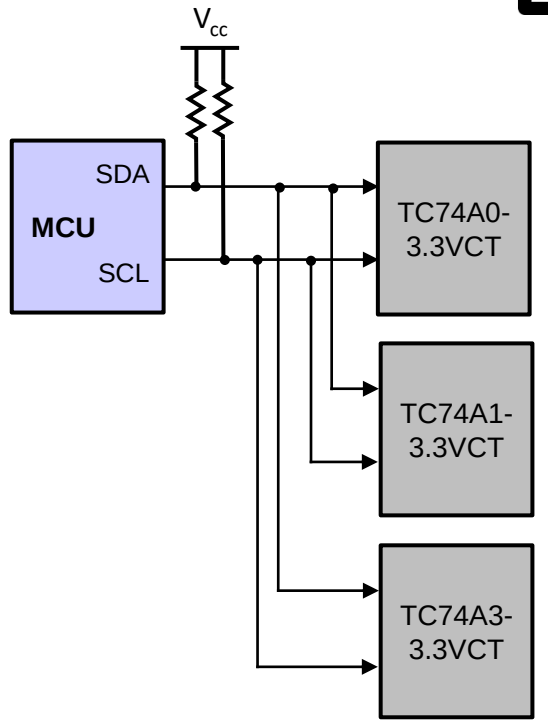


Fig.1 Topology

Read Byte Format

S	Address	WR	ACK	Command	ACK	S	Address	RD	ACK	Data	NACK	P
	7 Bits			8 Bits			7 Bits			8 Bits		

Slave Address

Command Byte: selects which register you are reading from.

Slave Address: repeated due to change in data-flow direction.

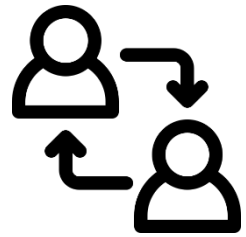
Data Byte: reads from the register set by the command byte.

Fig.2 Read protocol

SOT-23 (V)	Address
TC74A0-3.3VCT	1001 000
TC74A1-3.3VCT	1001 001
TC74A2-3.3VCT	1001 010
TC74A3-3.3VCT	1001 011

Fig.3 Sensor to address conversion

Interaction: I²C Bit Sequence



- Address: 1001 001
- Value for -10°C: 11110110
- S: Start, R: Read, A: Ack, W: Write, N: Nack, P: Stop

Master controlling SDA
Sensor controlling SDA

	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0		1	0	0	1	0	0	1	1	0	1	1	1	1	0	1	1	0	1	
S	Address							W	A	Register								A	S	Address							R	A	Value							N	P

I2C Write

I2C Read

Read Byte Format

S	Address	WR	ACK	Command	ACK	S	Address	RD	ACK	Data	NACK	P
	7 Bits			8 Bits			7 Bits			8 Bits		

Slave Address

Command Byte: selects which register you are reading from.

Slave Address: repeated due to change in data-flow direction.

Data Byte: reads from the register set by the command byte.

Fig.2 Read protocol

SPI – Serial Peripherals Interface

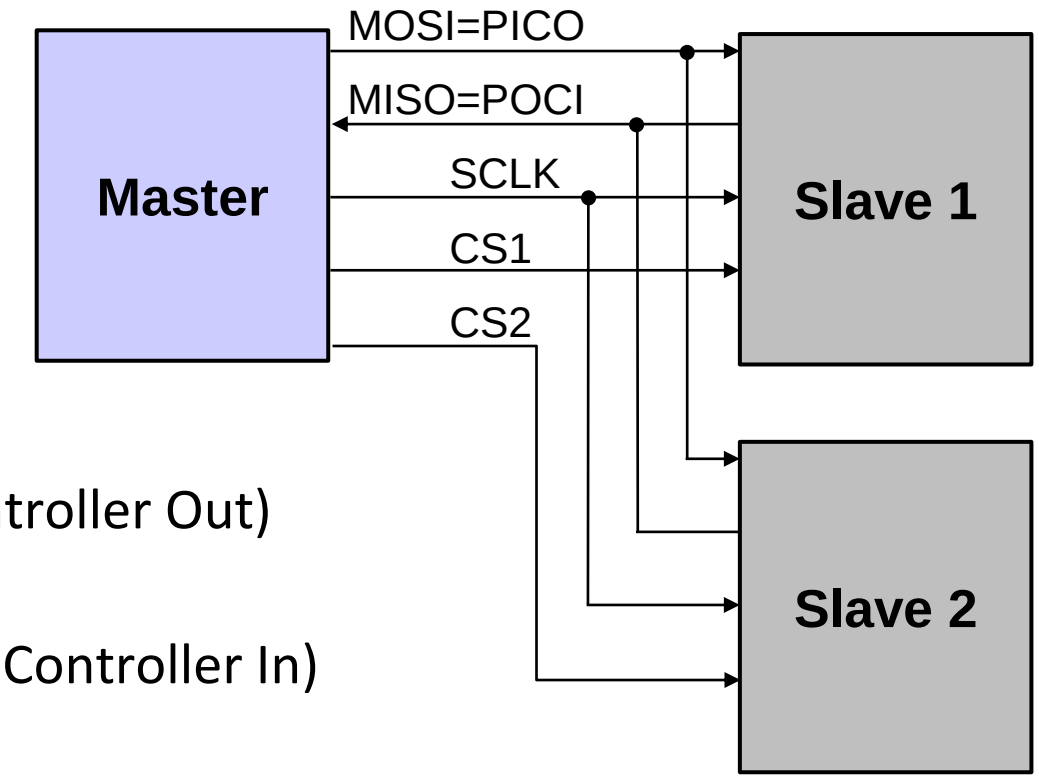
SPI – Serial Peripheral Interface

The *Serial Peripheral Interface (SPI)* has been introduced by Motorola (now Freescale Semiconductors) for the MC68HCxx microcontroller line.

- Communication with peripheral using four wires:
  - SD cards
  - Sensors
  - External ADC

SPI – Topology

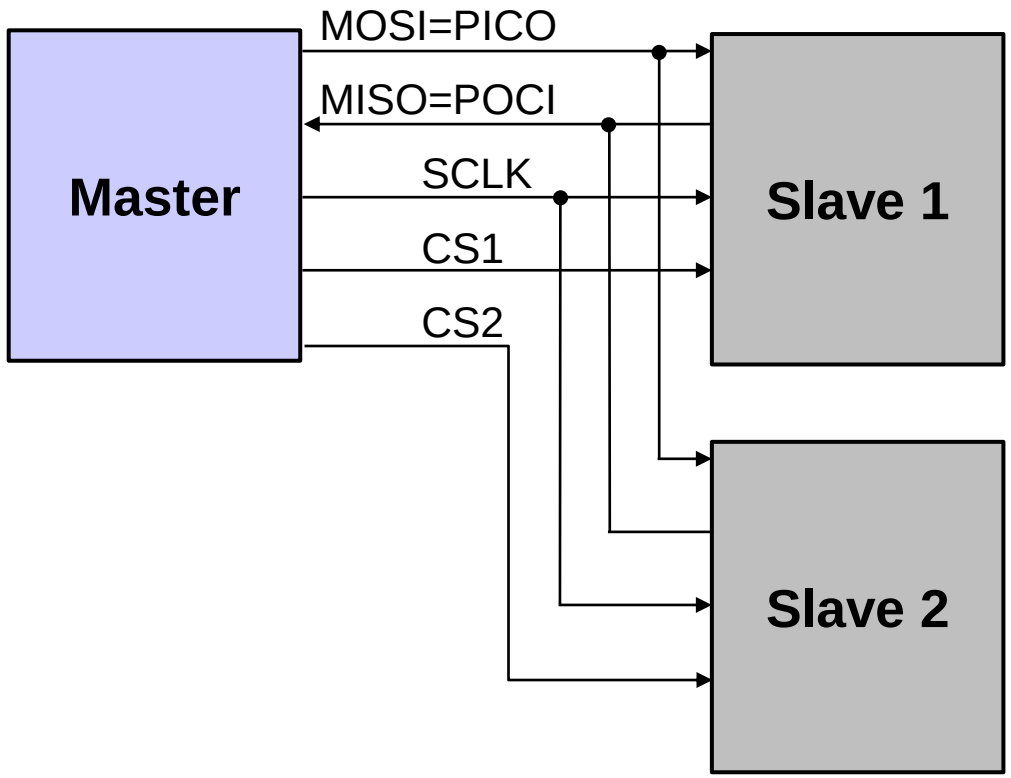
- communication lines:
  - MISO (master-in, slave-out data)
  - MOSI (master-out, slave-in data)
  - SCK (clock)
  - CSN (chip select, one per slave – usually active low)



- Names are not standard, beware!
  - SDI (SPI data in), PICO (Peripheral In, Controller Out) instead of MOSI
  - SDO (SPI data out) POCI (Peripheral Out, Controller In) instead of MISO
  - SCLK, CLK, SPC, … instead of SCK
  - CS, SS (slave select), SSN (slave select, active low) ... instead of CSN

SPI – Topology

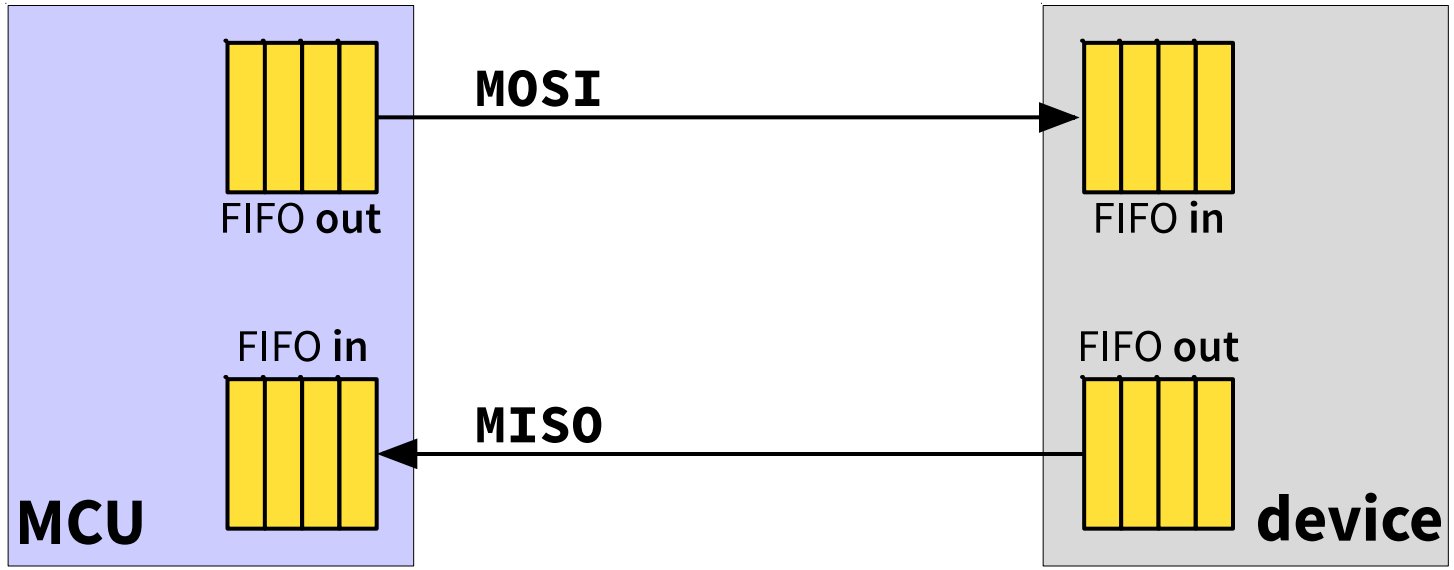
- Data synchronously transmitted on MISO/MOSI (full-duplex)
- Master drives SCLK which is shared by all SPI Slaves
- Master choses slave it wants to communicate with the CS line
- Bits are written to the data line on clock . The other device samples at the opposite edge of the same clock period
- Lines configured as Push-Pull



SPI – Peripheral Architecture

Full-duplex transfer: data is streamed between master and slave FIFO buffers:

- The master pushes the content of its buffer to the slave via MOSI
- The slave pushes the content of its buffer to the master via MISO

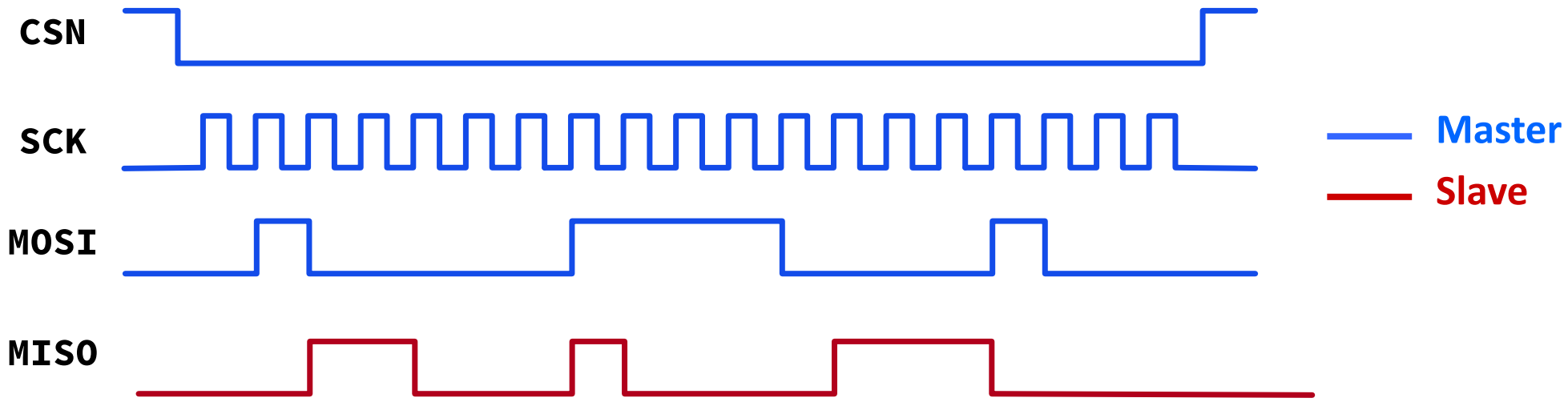


SPI – Protocol

1. MCU selects slave by pulling down the corresponding CS

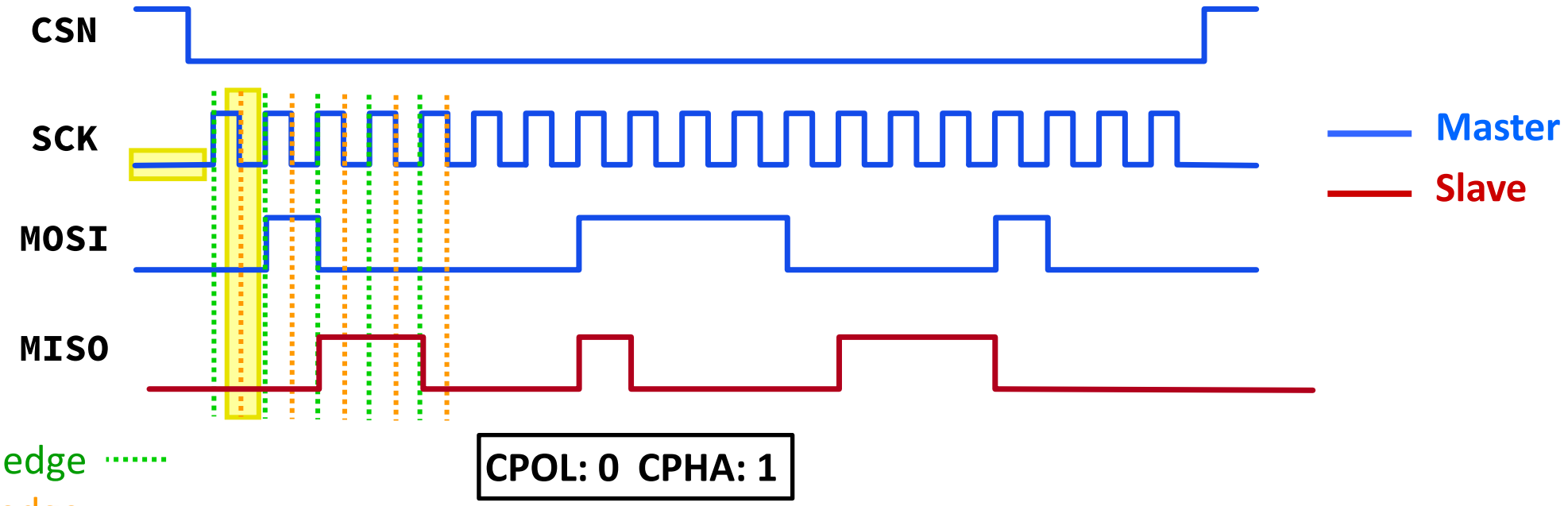
SPI – Protocol

1. MCU selects slave by pulling down the corresponding CS
2. MCU generates the clock for signal transmission
   - Depending on the configuration, data is sampled on falling/rising edge
   - Depending on the configuration, clock signal is idle low/high



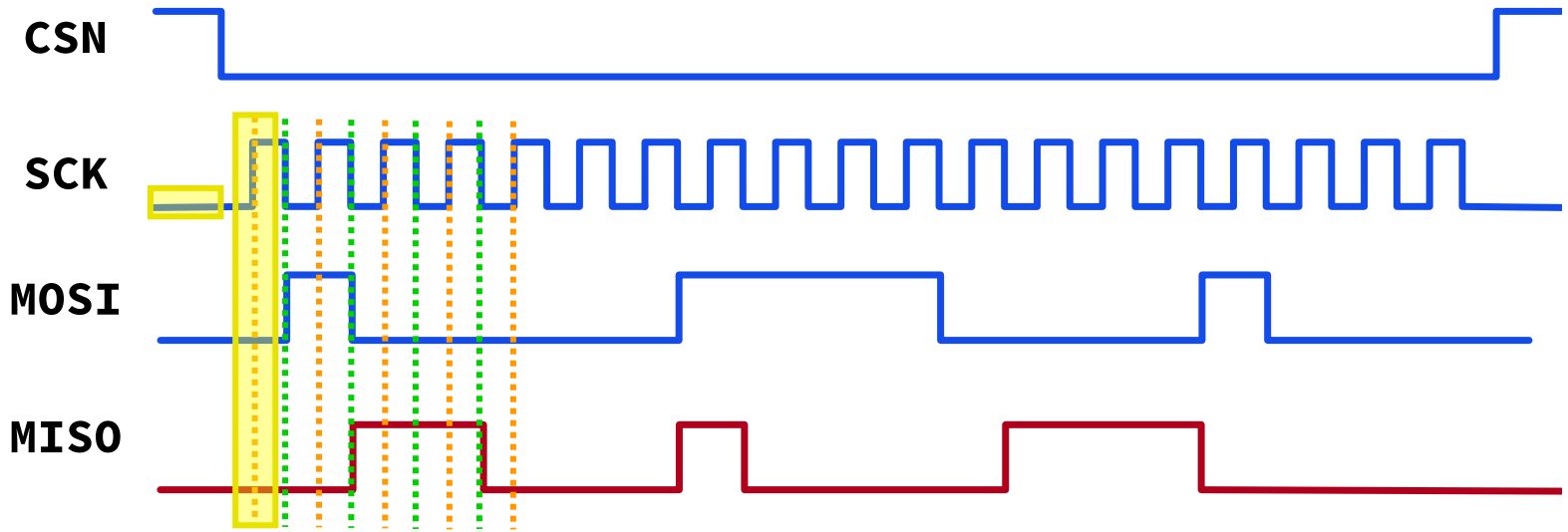
SPI – Protocol – CPOL & CPHA

- *CPOL (Clock Polarity)* during idle State: Low: 0, High: 1
- *CPHA: (Clock Phase)* defines which edge will be used for sampling
  - 0: Sample at first clock transition (from idle to active)
  - 1: Sample at second clock transition (from active to idle)



SPI – Protocol – CPOL & CPHA

- *CPOL (Clock Polarity)* during idle State: Low: 0, High: 1
- *CPHA: (Clock Phase)* defines which edge will be used for sampling
  - 0: Sample at first clock transition (from idle to active)
  - 1: Sample at second clock transition (from active to idle)



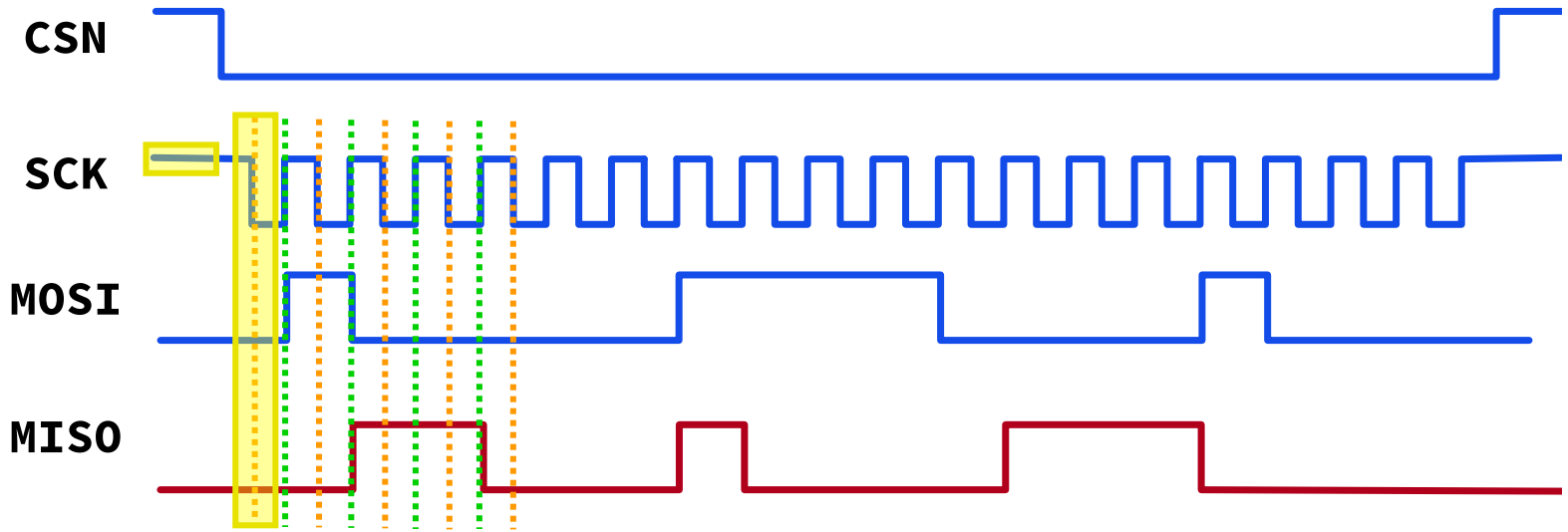
Switching edge

Sampling edge

CPOL: 0 CPHA: 0

SPI – Protocol – CPOL & CPHA

- *CPOL (Clock Polarity)* during idle State: Low: 0, High: 1
- *CPHA: (Clock Phase)* defines which edge will be used for sampling
  - 0: Sample at first clock transition (from idle to active)
  - 1: Sample at second clock transition (from active to idle)



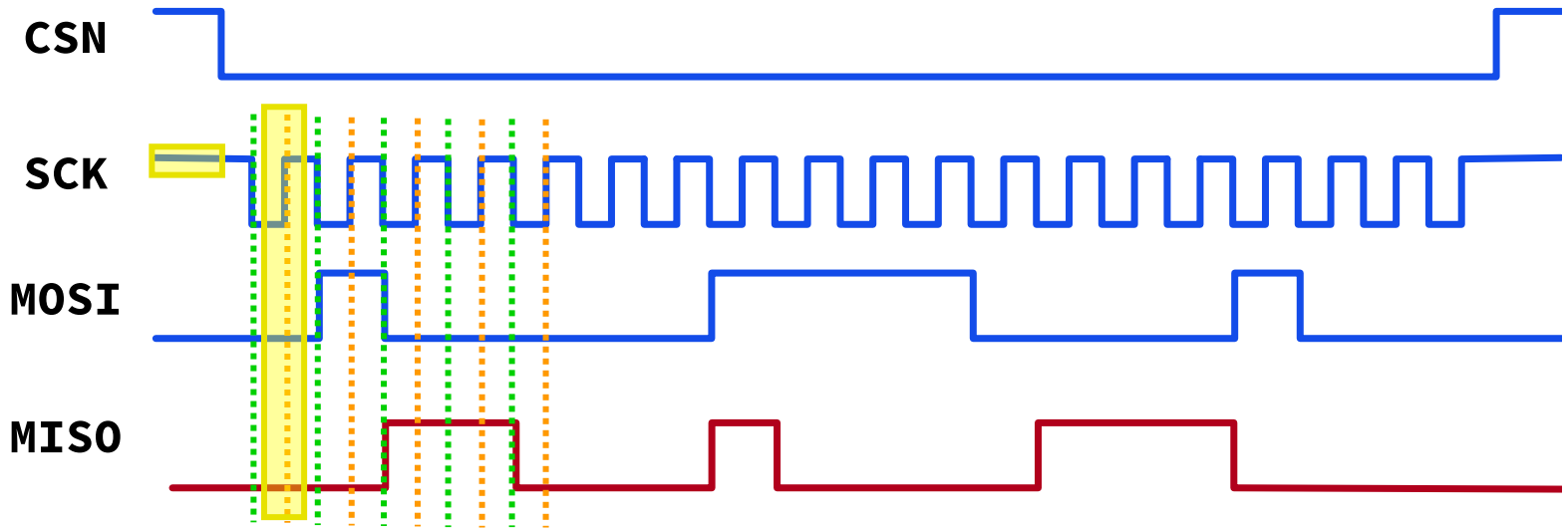
Switching edge

Sampling edge

CPOL: 1 CPHA: 0

SPI – Protocol – CPOL & CPHA

- *CPOL (Clock Polarity)* during idle State: Low: 0, High: 1
- *CPHA: (Clock Phase)* defines which edge will be used for sampling
  - 0: Sample at first clock transition (from idle to active)
  - 1: Sample at second clock transition (from active to idle)

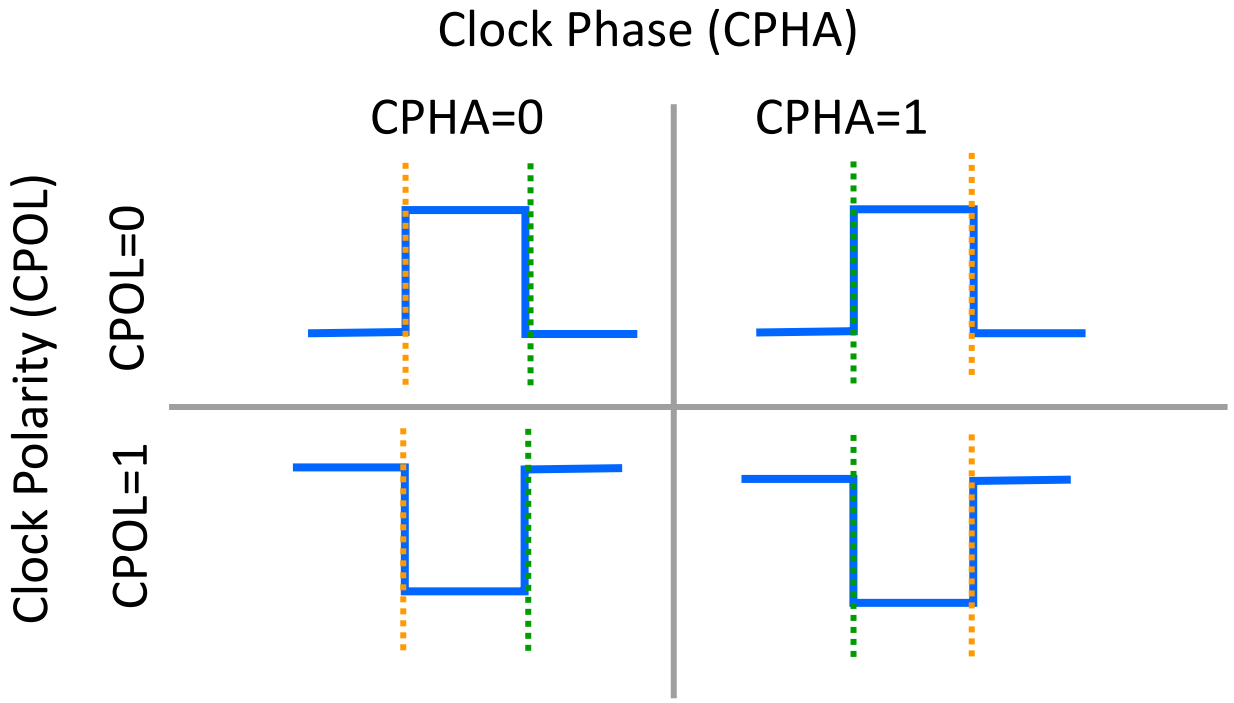


Switching edge
Sampling edge

CPOL: 1 CPHA: 1

SPI – Protocol – CPOL & CPHA

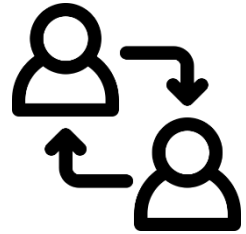
Configurations Summary:



Switching edge

Sampling edge

Interaction: SPI vs. I^2C

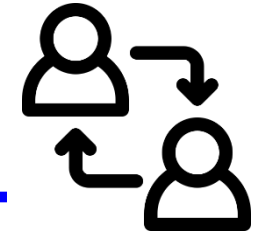


I^2C needs external pull-up resistor, why are they not needed for SPI as well?

Select the correct answer(s).

a) Because addressing is based on CS line and not by sending the address
b) The devices don’t need to control the same line simultaneously
c) The default state is low
d) SPI uses a push-pull configuration whereas I^2C uses open-drain
e) SPI uses an open-drain configuration whereas I^2C uses push-pull

Interaction: SPI vs. I^2C



I^2C needs external pull-up resistors, why are they not needed for SPI as well?

Select the correct answer(s).

a) Because addressing is based on theCS line and not by sending the address

b) The devices don’t need to control the same line simultaneously

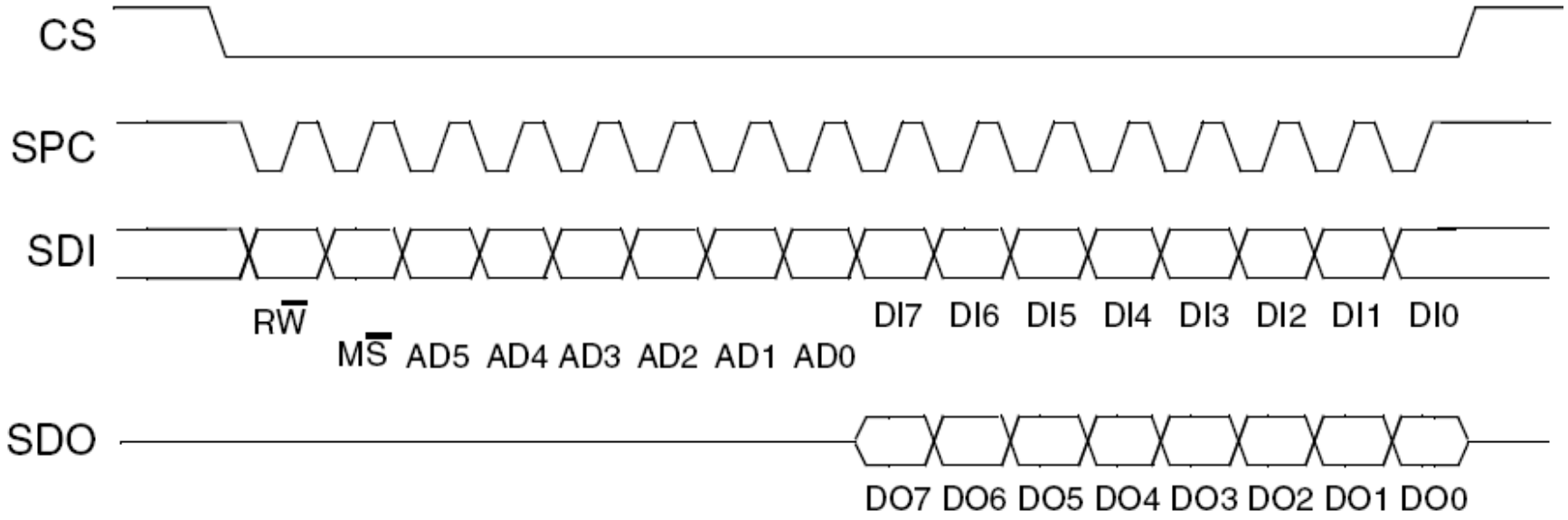
c) The default state is low

d) SPI uses a push-pull configuration whereas I^2C uses open-drain

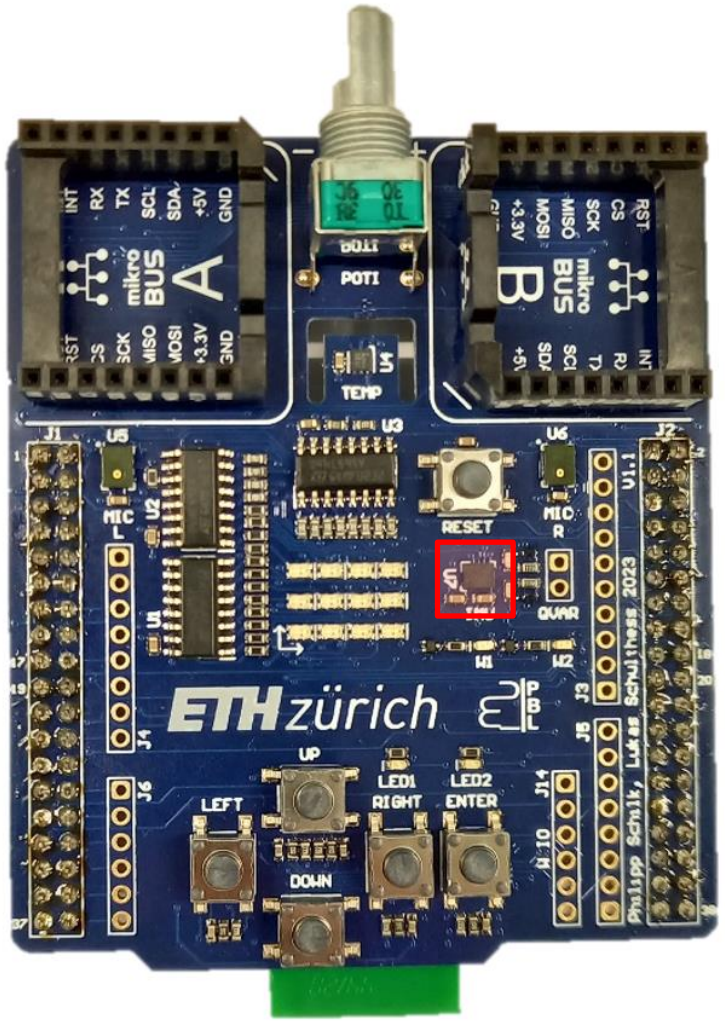
e) SPI uses an open-drain configuration whereas I^2C uses push-pull

SPI – Interface Protocol

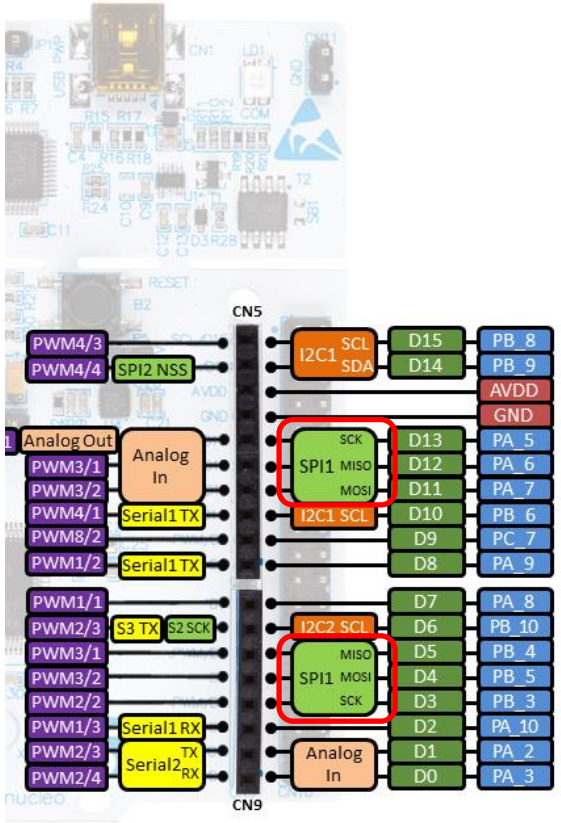
More complex behavior than simple data streaming can be mapped on top of SPI protocol, for example: *command + address + data streaming This is device dependent*



SPI on Sensor Shield

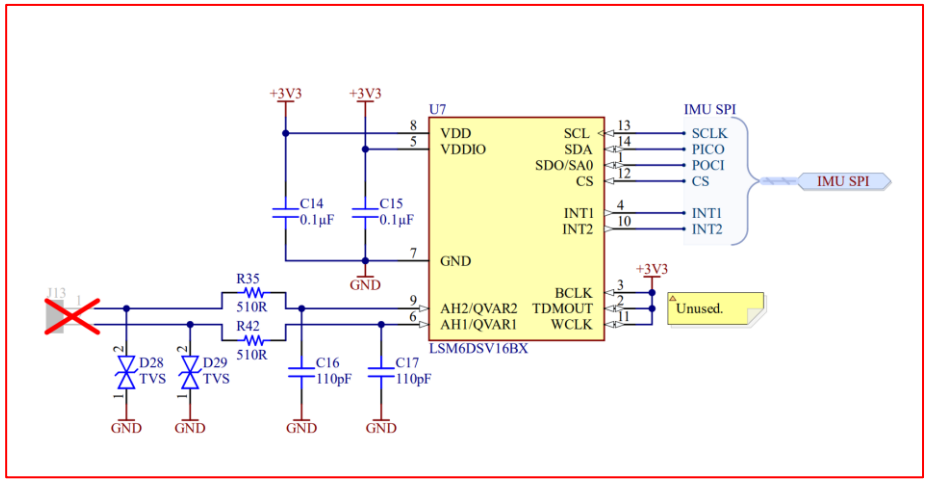


Embedded systems sensor shield



Left Side of Nucleo-L476RG

6-Axis IMU (Inertial Measurement Unit)



SPI vs. I²C

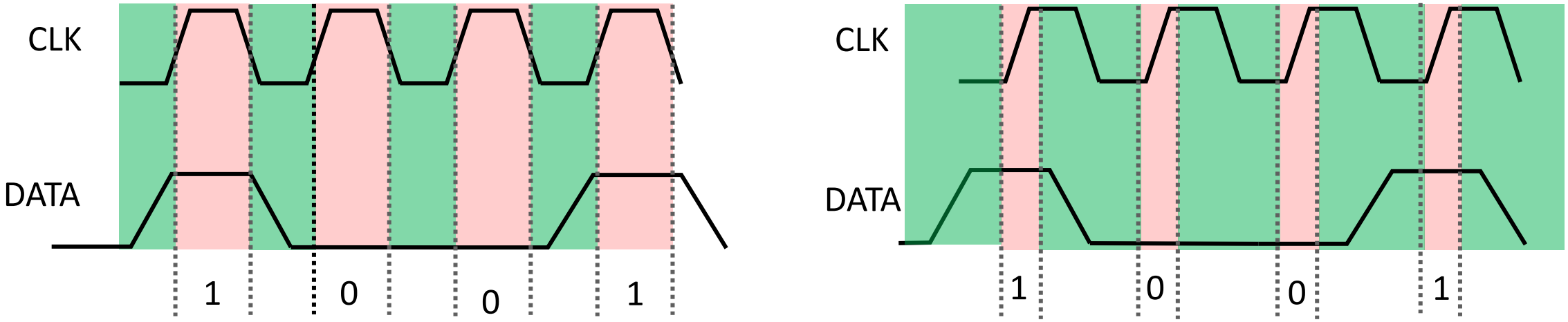
I²C

- Open drain with external pull-ups
- Addressing causes overhead
- No extra hardware for additional slave
- Level triggered

SPI

- Push-pull configuration
- No addressing overhead
- Extra hardware for each additional slave
- Edge triggered

Switching/Holding Times (CPOL:0,CPHA: 0)



Comparison

	SPI	I^2C	UART
HW complexity	4 lines (+1 for additional slave)	2 lines	2 lines (+2 if HW flow control)
Sync/Async	Synchronous	Synchronous	Asynchronous
Relationship	Single master multiple slaves	Multiple master multiple slaves	One to one
Addressing	CS line	Address is sent	-
System	Full-Duplex	Half-duplex	Full-Duplex
Speed	20 Mbps	3.4 Mbps	5Mbps (mostly less)
Advantages	No processing overhead Simple hardware	Few data lines to communicate with many devices	No master-salve relationship, both can initiate data transmission
Disadvantages	Extra connection per slave	Low speed, protocol overhead, RC speed limitations	Only two devices, no synchronization, relies on accurate hardware clock

What Did You Learn?

- ✓ Asynchronous/Synchronous Communication
- ✓ Parallel/Serial Communication
- ✓ UART
  - ✓ Baud rate vs. Bit rate
- ✓ I^2C
  - ✓ GPIO configuration, open-drain, pull-up resistor
  - ✓ I^2C Read
  - ✓ I^2C Write
  - ✓ I^2C Combined
- ✓ SPI
  - ✓ Protocol
  - ✓ Clock polarity