Embedded Systems
Lecture 4.
Hardware Software Architecture.
Interrupts and Serial interfaces.

© Lothar Thiele

Computer Engineering and Networks Laboratory

Michele Magno

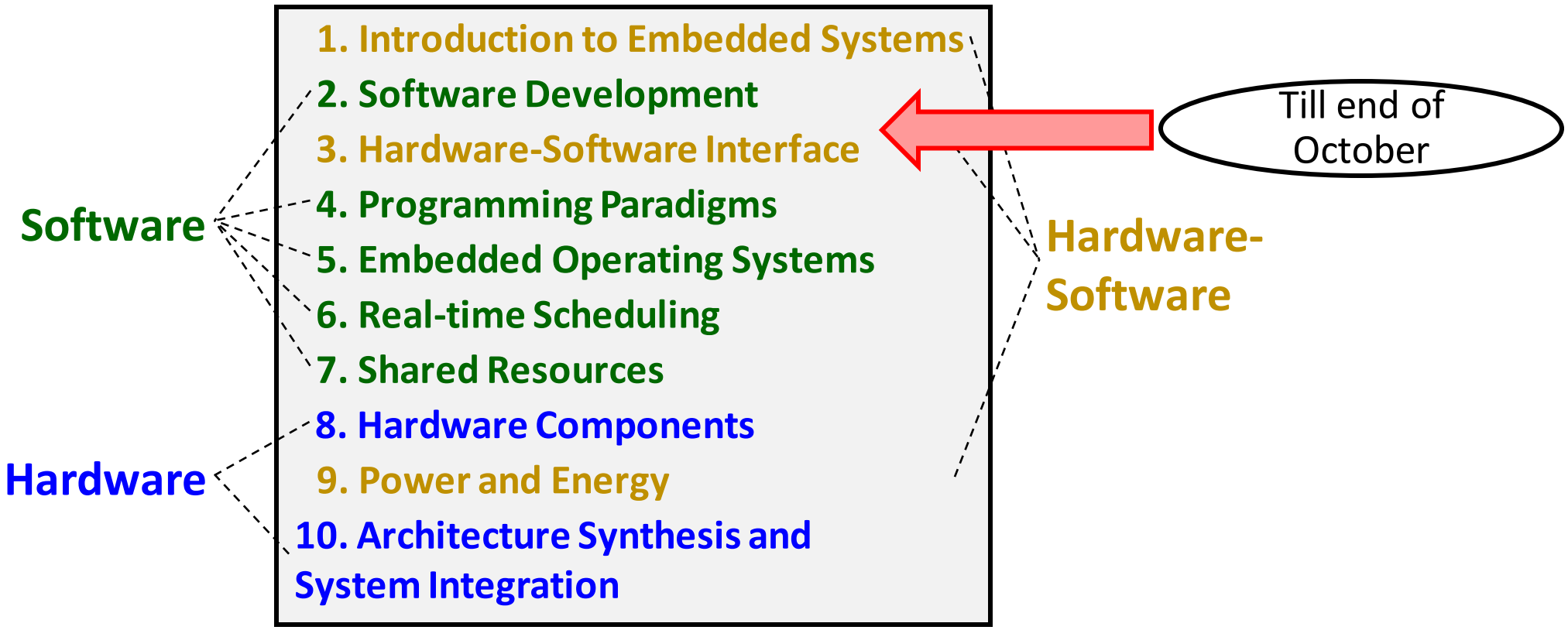
D-ITET center for project based learning

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Credits: Philipp Mayer, Francesco Conti.

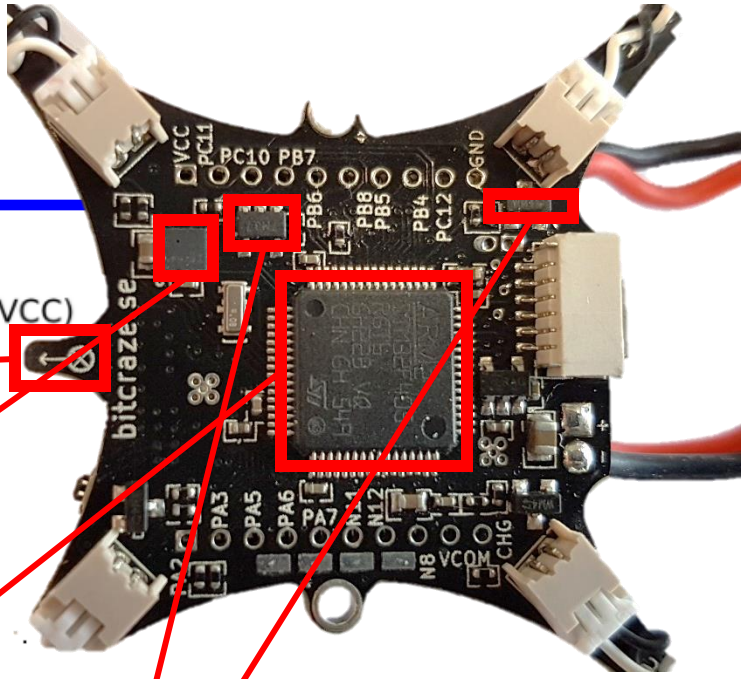
Where we are …

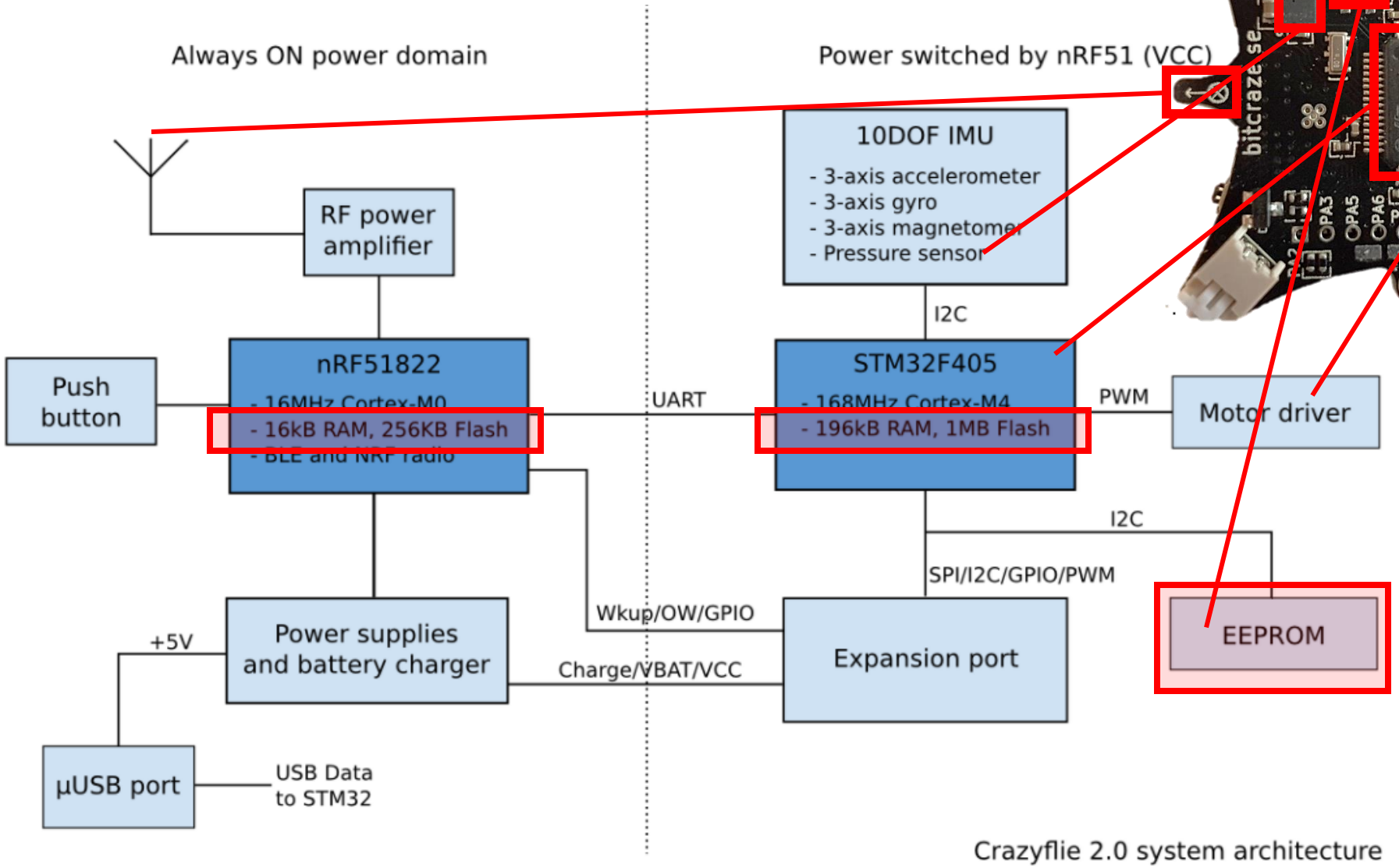

1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis and System Integration
Software
Hardware
Hardware-Software
Till end of October

Do you Remember ?

- Memory
  - SRAM, DRAM, Flash
  - Addresses and value
  - Memory map
- Clocks
  - Many clocks
  - Flexibility
  - Energy and latency
- GPIO first peripherals going out.

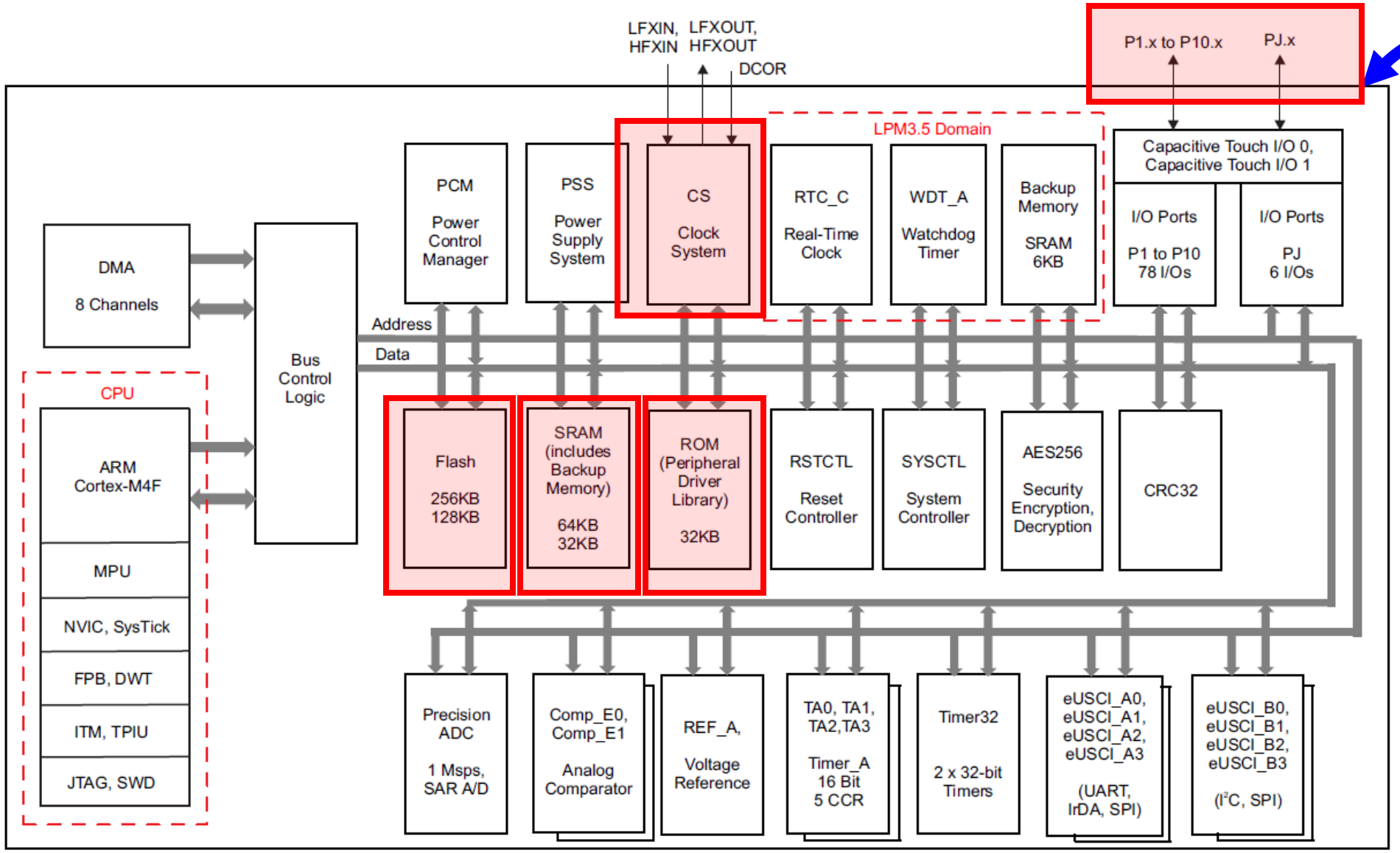
Remember … ?

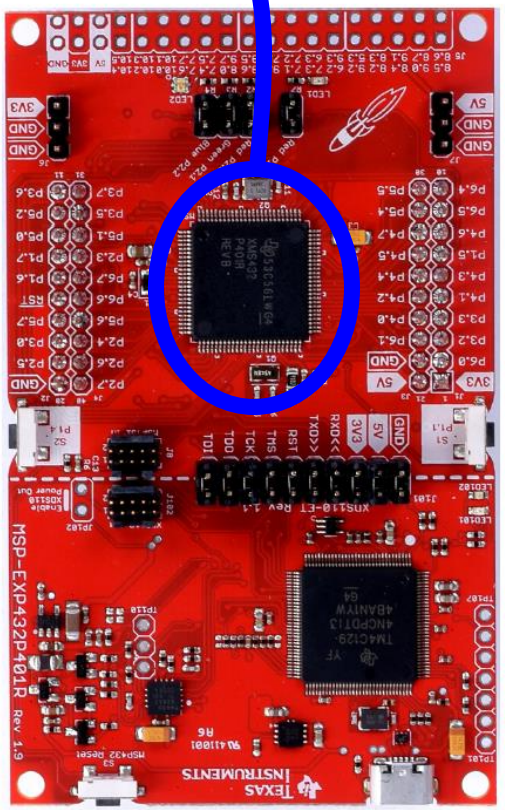




Crazyflie 2.0 system architecture

MSP432P401R (Last year lab - ES-Lab)

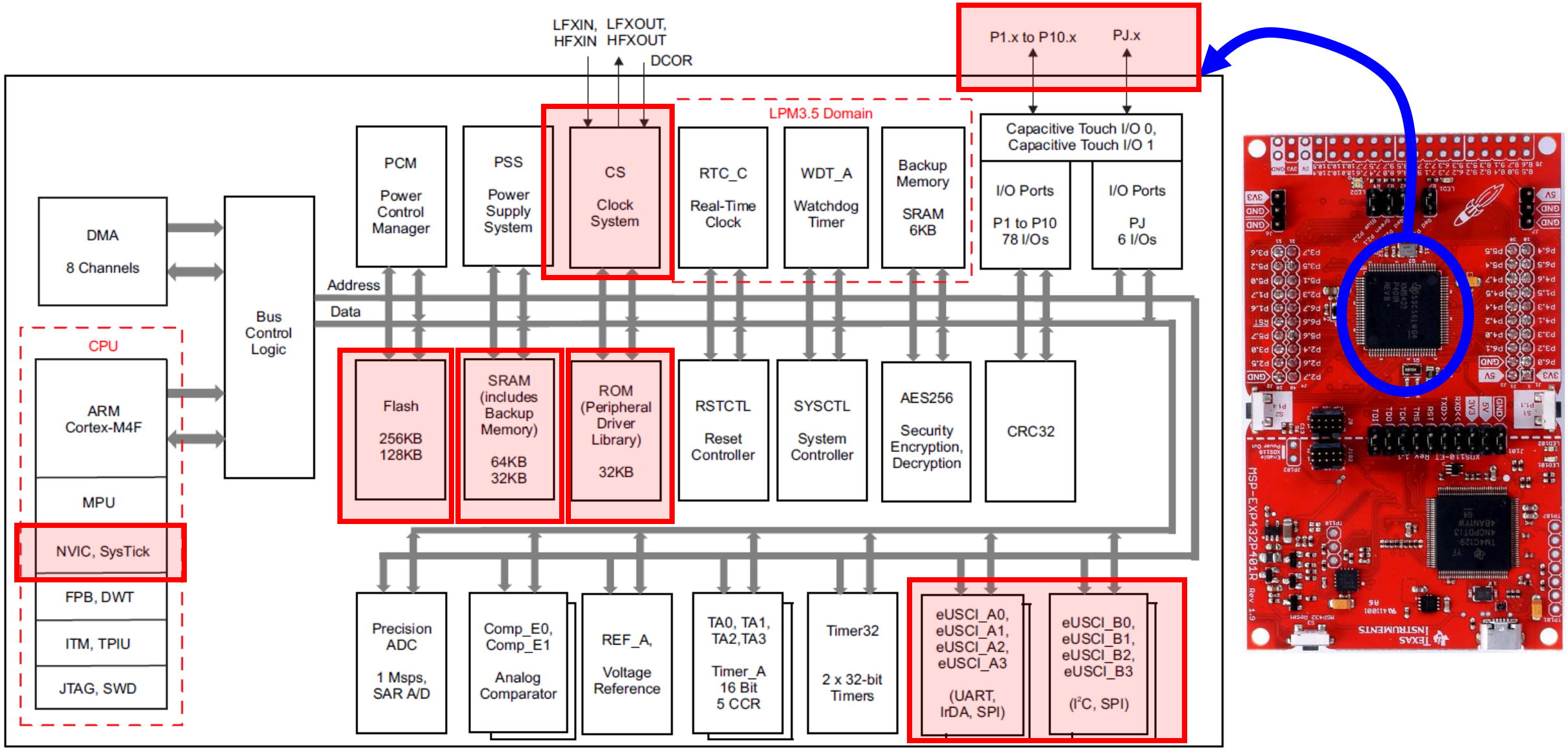




Today class will focus on

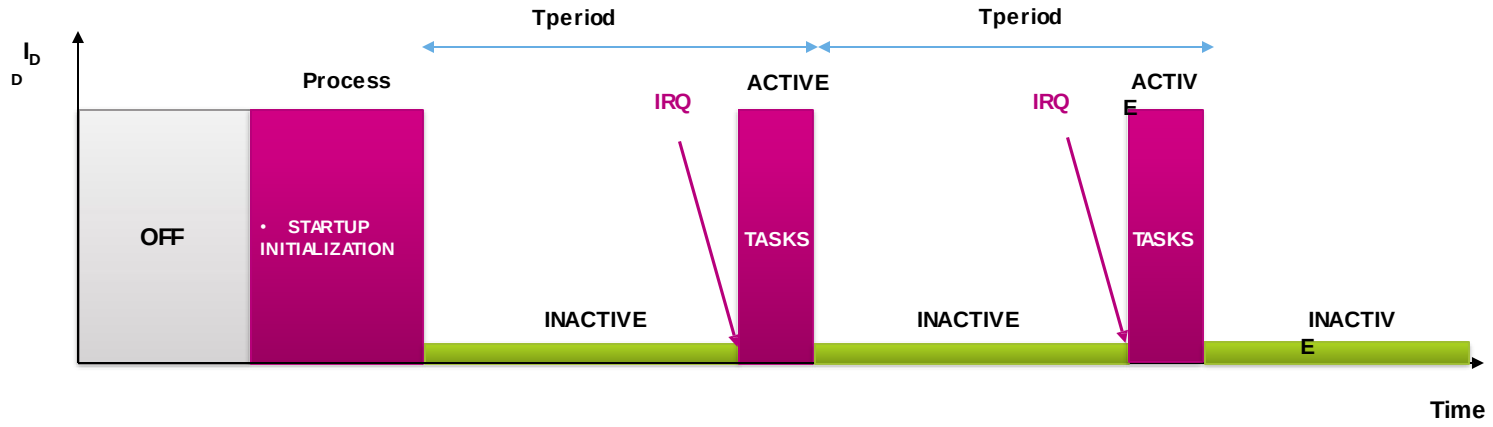
- Interrupts
- Input / Output
  - Serial communication
  - SPI
  - I2C
  - UART
- Link with the lab board

MSP432P401R (Last year lab - ES-Lab)



Interrupts

Typical application profile with Embedded systems



- Application phases:
  - OFF – power is not applied to MCU
  - STARTUP INITIALIZATION – MCU performs configuration (peripherals, clocks, …)
  - Tperiod
    - INACTIVE – MCU is in low power mode to reduce power consumption
    - ACTIVE – MCU is in normal mode and performs tasks

‘Interrupt’ Defined

From Wikipedia:

> *A hardware interrupt is an electronic alerting signal sent to the processor from an external device, either a part of the [device, such as an internal peripheral] or an external peripheral.*

Waiting for an Event: Family Vacation

Polling

Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?

An engineering example...

Waiting for an Event: Family Vacation

Polling

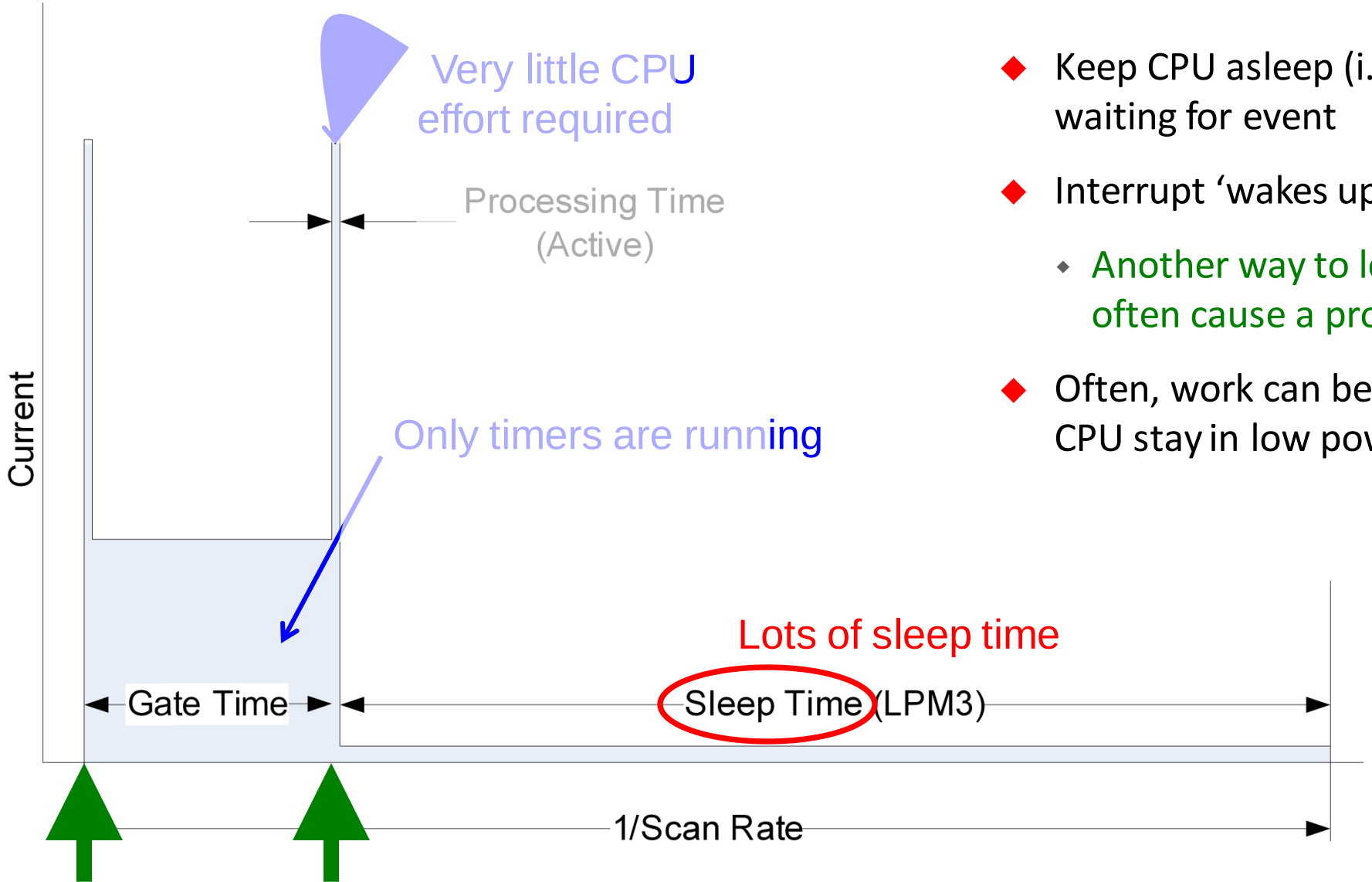
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?
Are we there yet?

Interrupts

Wake me up when we get there...

An engineering example...

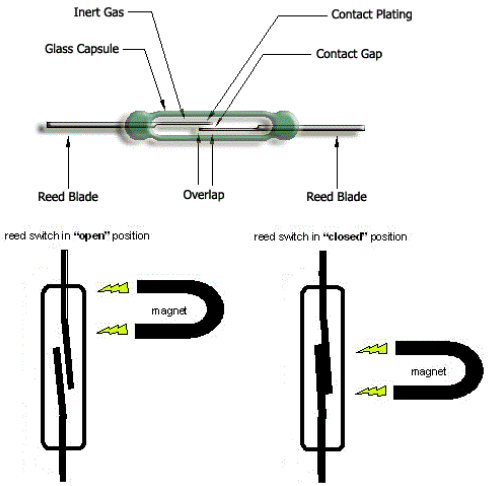
Interrupts Help Support Ultra Low Power



- Keep CPU asleep (i.e. in Low Power Mode) while waiting for event
- Interrupt 'wakes up' CPU when it's required
  - Another way to look at it is that interrupts often cause a program state change
- Often, work can be done by peripherals, letting CPU stay in low power mode (e.g. *Gate Time*)

Waiting for an Event: Button Push

Polling



Interrupts

```
while(1) {
 // Polling GPIO button
 while
(GPIO_getInputPinValue()==1)
   GPIO_toggleOutputOnPin();
}
```

```
// GPIO button interrupt
#Iterrupt servise routine
__interrupt void rx (void){
  GPIO_toggleOutputOnPin();
}
```

> 0.1% CPU Load

How interrupts can affect system design…

Interrupts

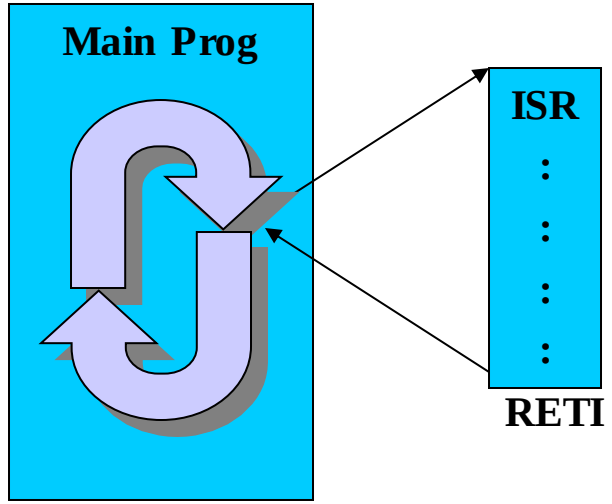
- A way to respond to an external event (i.e., flag being set) without polling

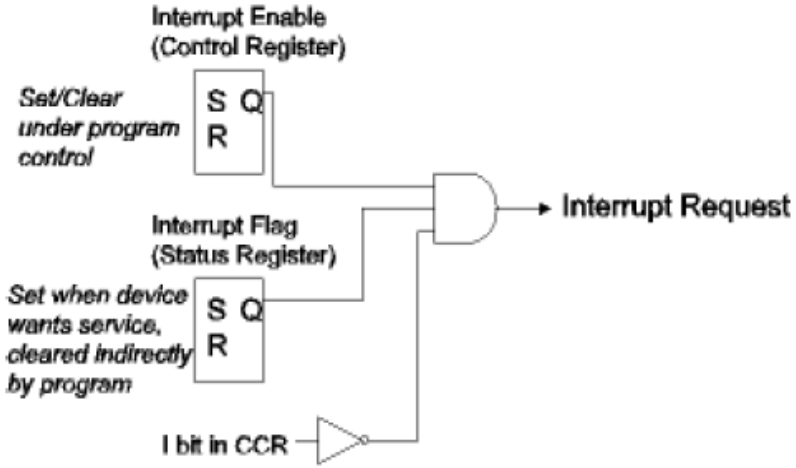
How it works:

- H/W senses flag being set
- Automatically transfers control to s/w that "services" the interrupt
- When done, H/W returns control to wherever it left off

Advantages:

- Transparent to user
- cleaner code
- µC doesn't waste time polling

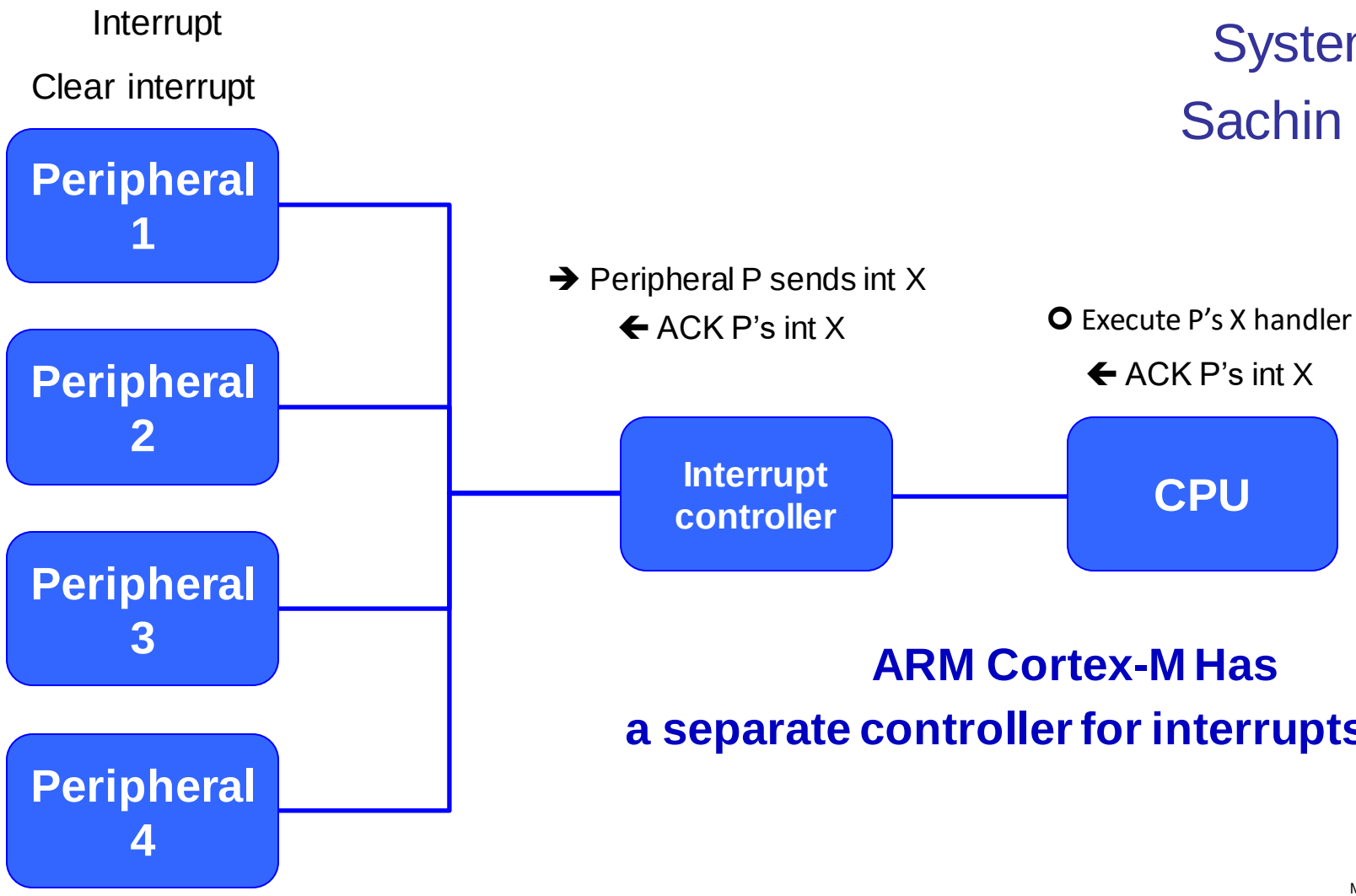




How do interrupts work?

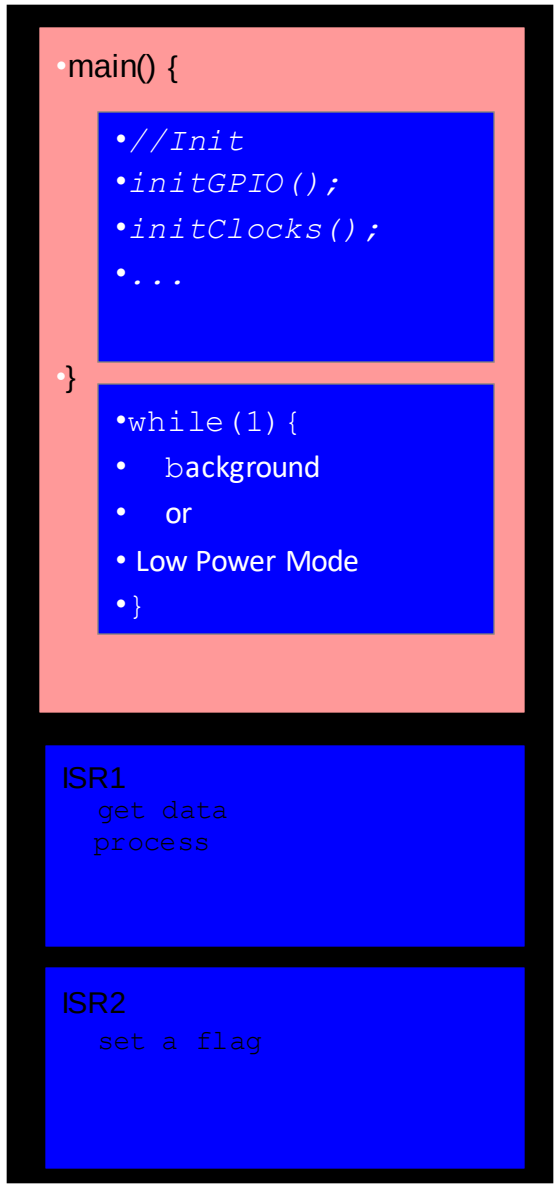
Credits: Networked Embedded Systems

Sachin Katti



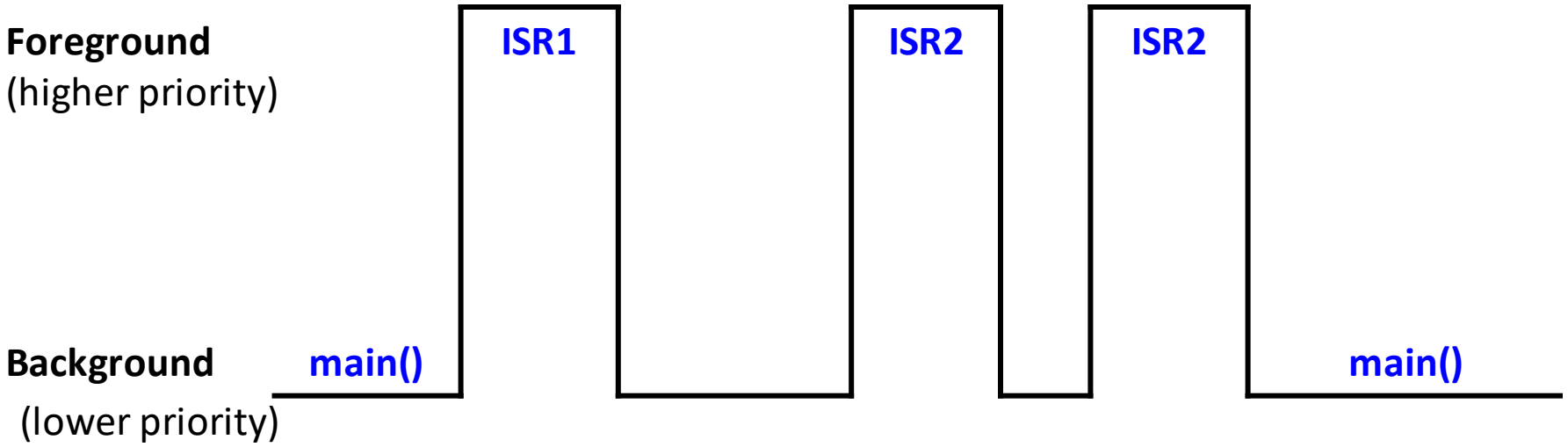
ARM Cortex-M Has a separate controller for interrupts! WHY?

Foreground / Background Scheduling

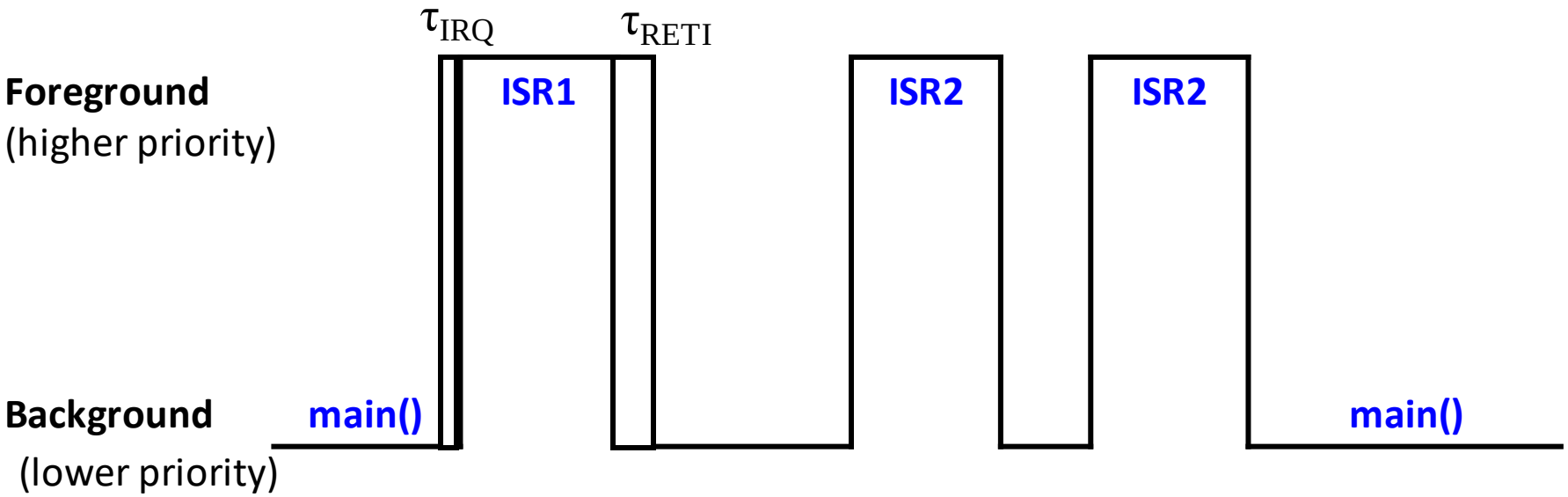


- System Initialization
  - The beginning part of main() is usually dedicated to setting up your system
- Background
  - Most systems have an endless loop that runs 'forever' in the background
  - In this case, 'Background' implies that it runs at a lower priority than 'Foreground'
  - In microcontrollers, the background loop often contains a Low Power Mode command – this sleeps the CPU/System until an interrupt event wakes it up
- Foreground
  - Interrupt Service Routine (ISR) runs in response to enabled hardware interrupt
  - These events may change modes in Background – such as waking the CPU out of low-power mode
  - ISR's, by default, are not interruptible
  - Some processing may be done in ISR, but it's usually best to keep them short

Foreground / Background (States)



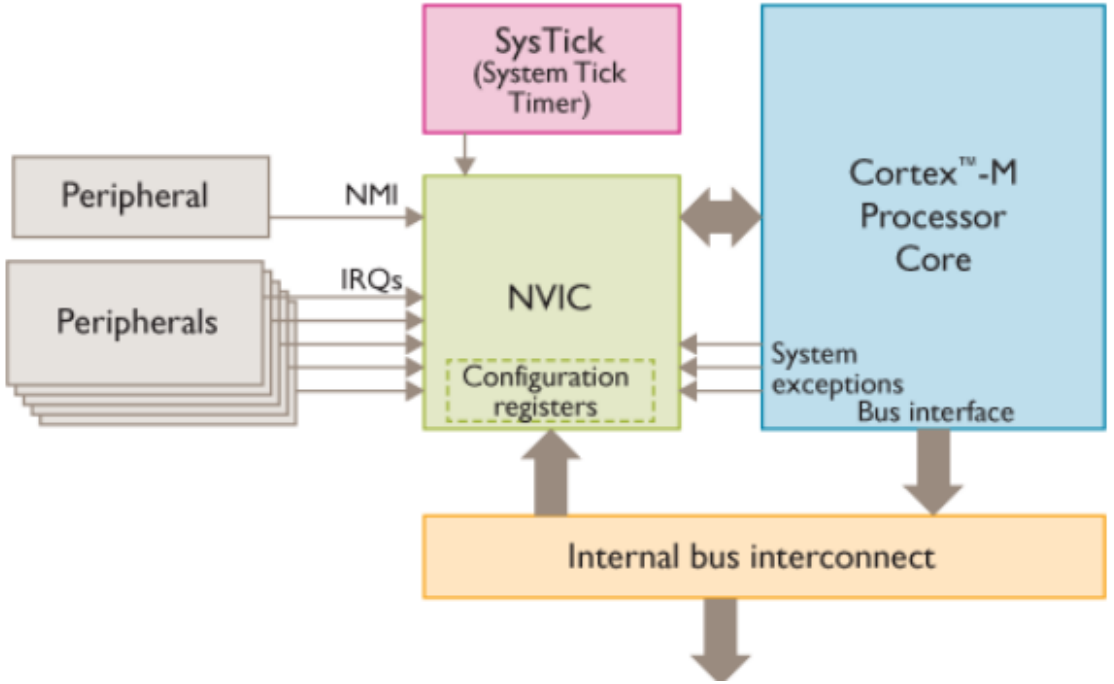
The interrupt has a latency cost as everything



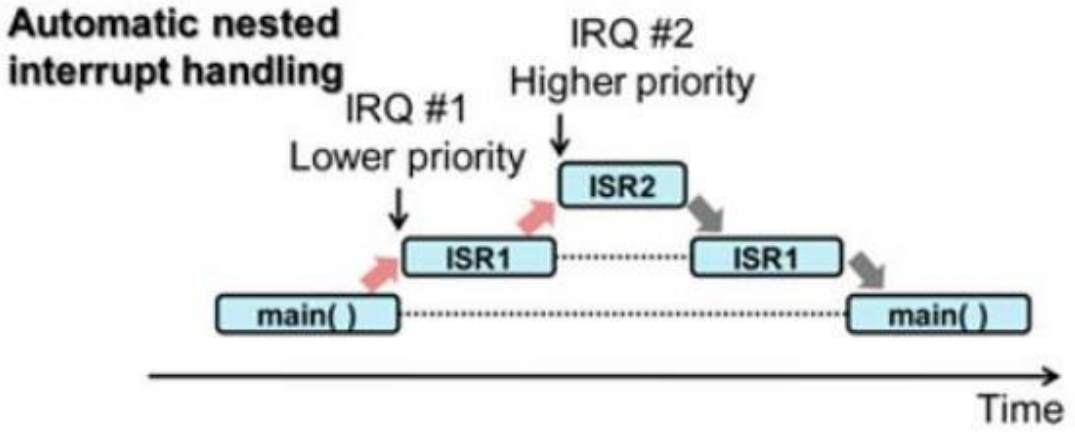
ARM Cortex-Mx / Nested Vector Interrupt Controller

The **NVIC** includes the following features:

- A large number of **maskable interrupt channels**
- Several programmable priority levels (4 bits of interrupt priority are used)
- Pow-latency exception and interrupt handling (**12 Cycles, can change with architecture**!!!)
- Power management control
- Implementation of system control registers



Nested Vector Interrupt Controller

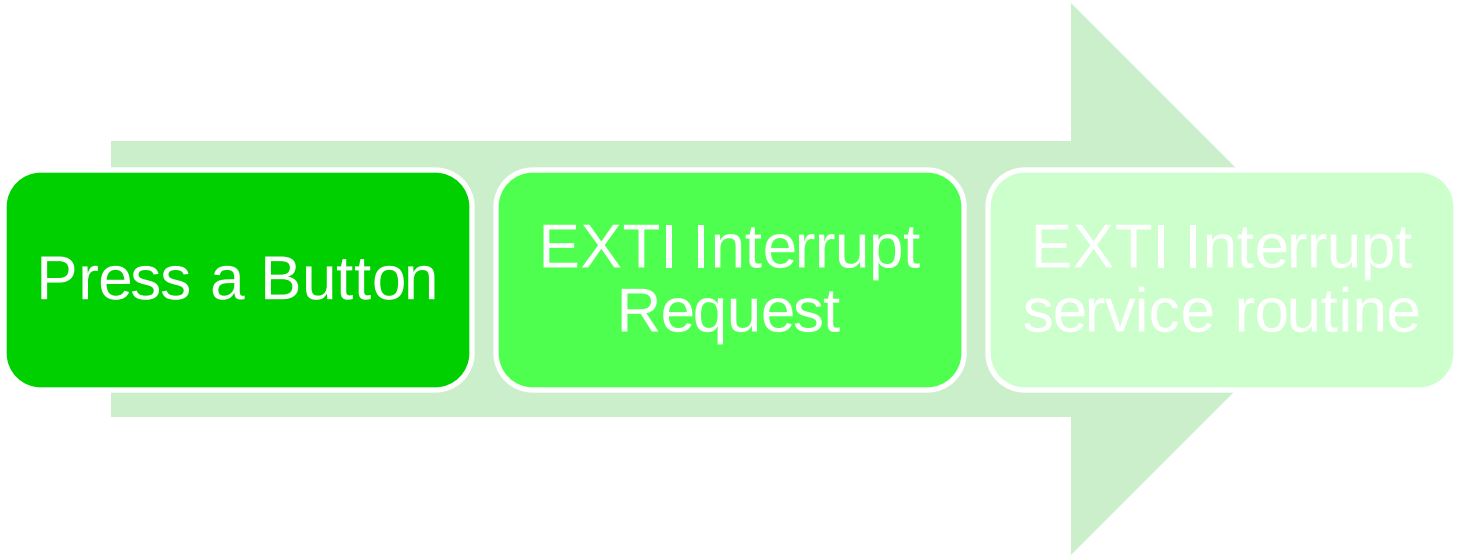


- **Nested Interrupt**: If an interrupt request (IRQ) with higher priority is raised, it is served first

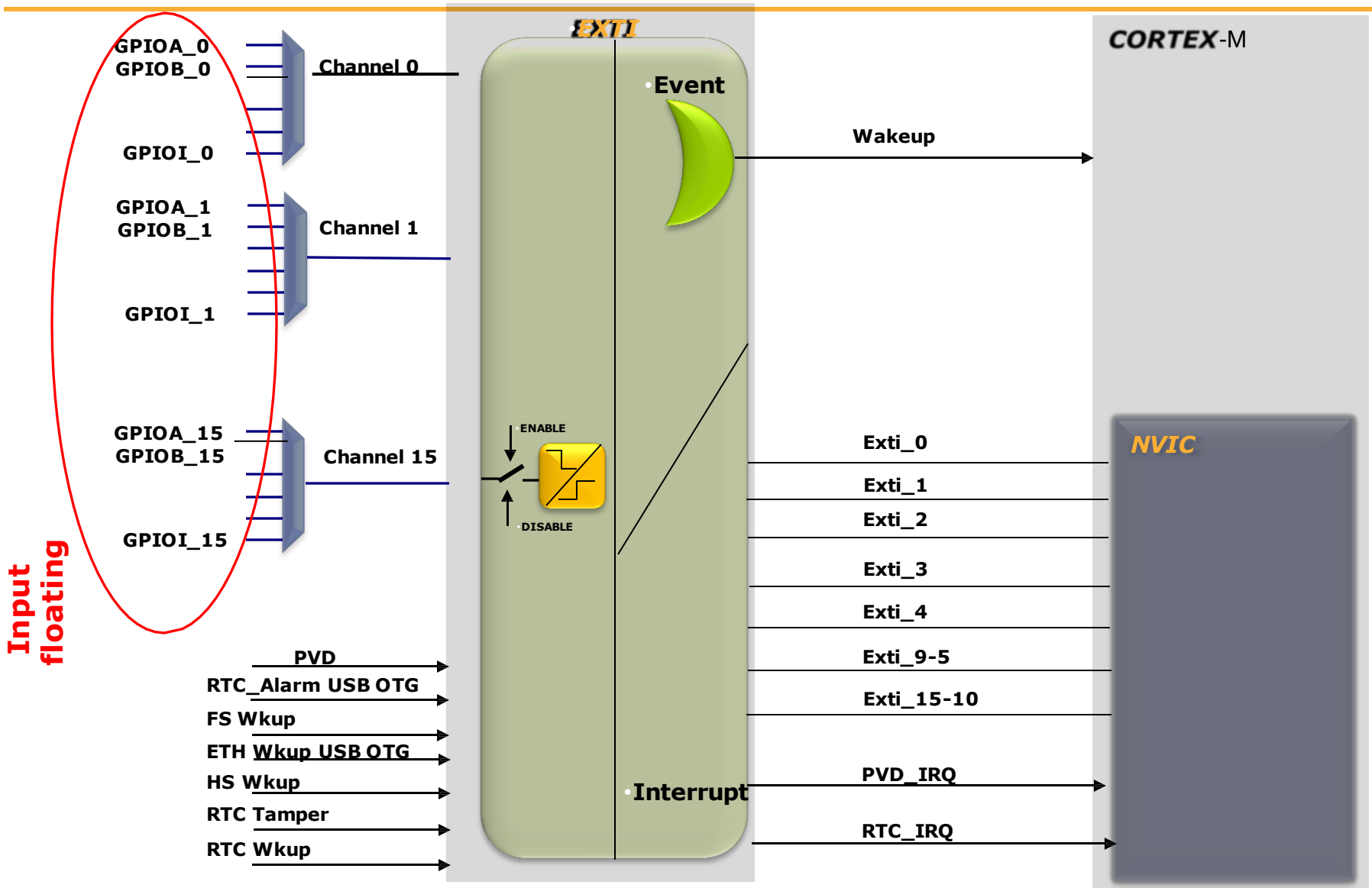
EXTernal Interrupt (EXTI)

EXTI – Purpose

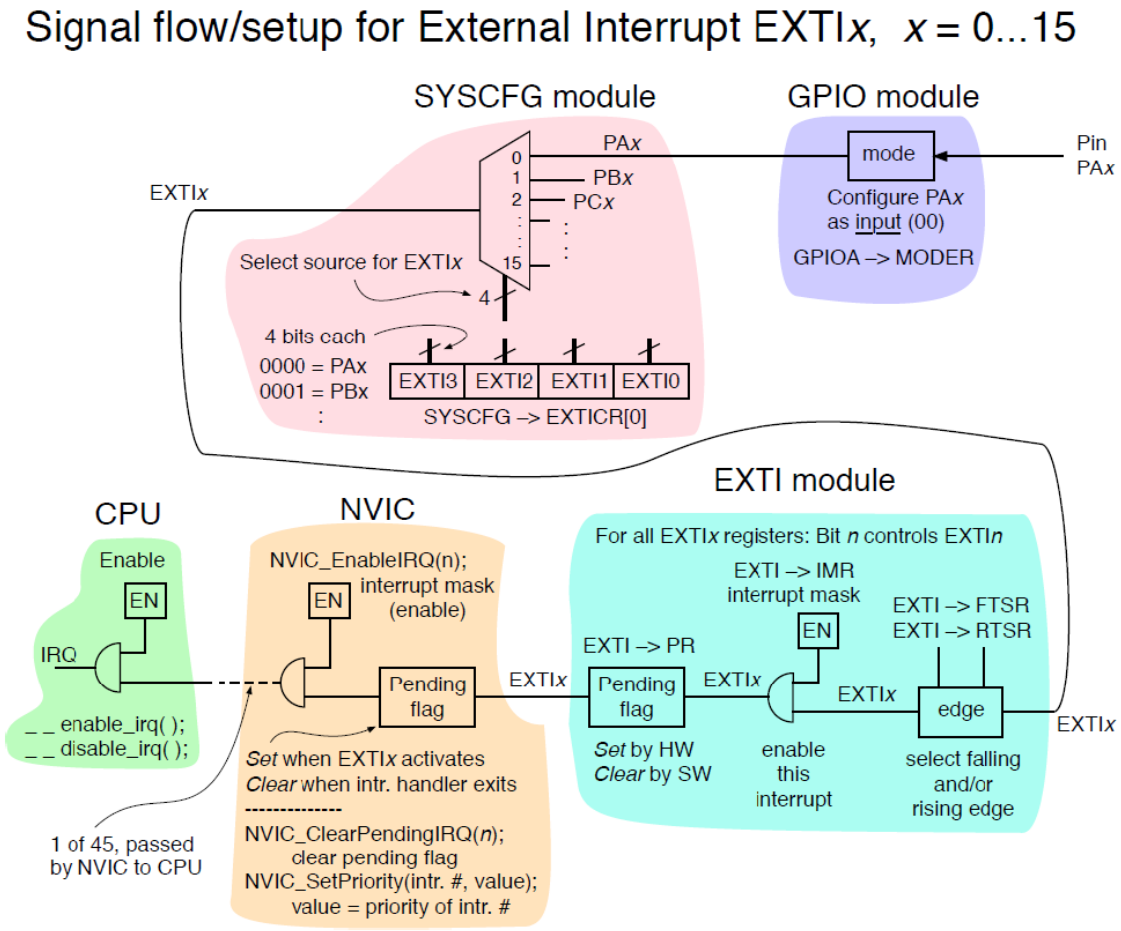
- We want to configure an external interrupt line.
- An EXTI line is configured to generate an interrupt on each falling edge.
- In the interrupt routine a led connected to a specific GPIO pin is toggled.



EXTI module: from pin to NVIC



Functional Signal Flow



SYSCFG

remap the memory accessible in the code area

manage the external interrupt line connection to the GPIOs.

EXTI

Configure GPIO pin as a digital input

Select the pin as the EXTInsource (in SYSCFG module)

Enable interrupt to be requested when a flag is set by the desired event (rising/falling edge)

Clear the pending flag (to ignore any previous events)

NVIC

Enable interrupt: *NVIC_EnableIRQ(IRQn);*

Set priority: *NVIC_SetPriority(IRQn, priority);*

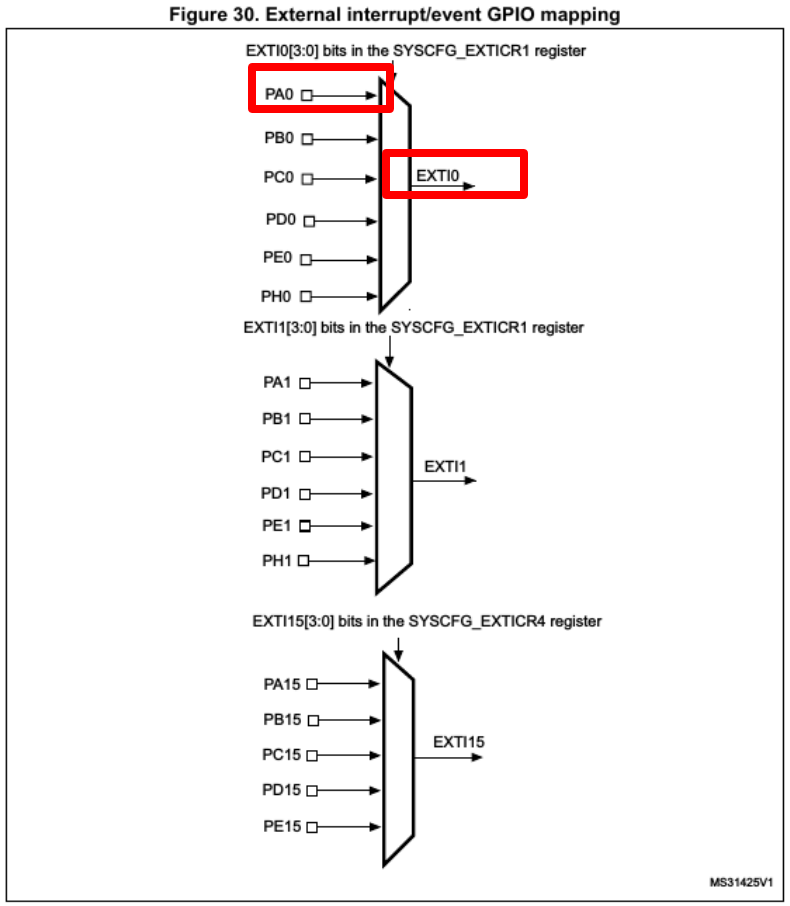
Clear pending status: *NVIC_ClearPendingIRQ(IRQn);*

Initialize counters, pointers, global variables, etc.

Enable CPU Interrupts:

Many steps clocked means many clocks to be served

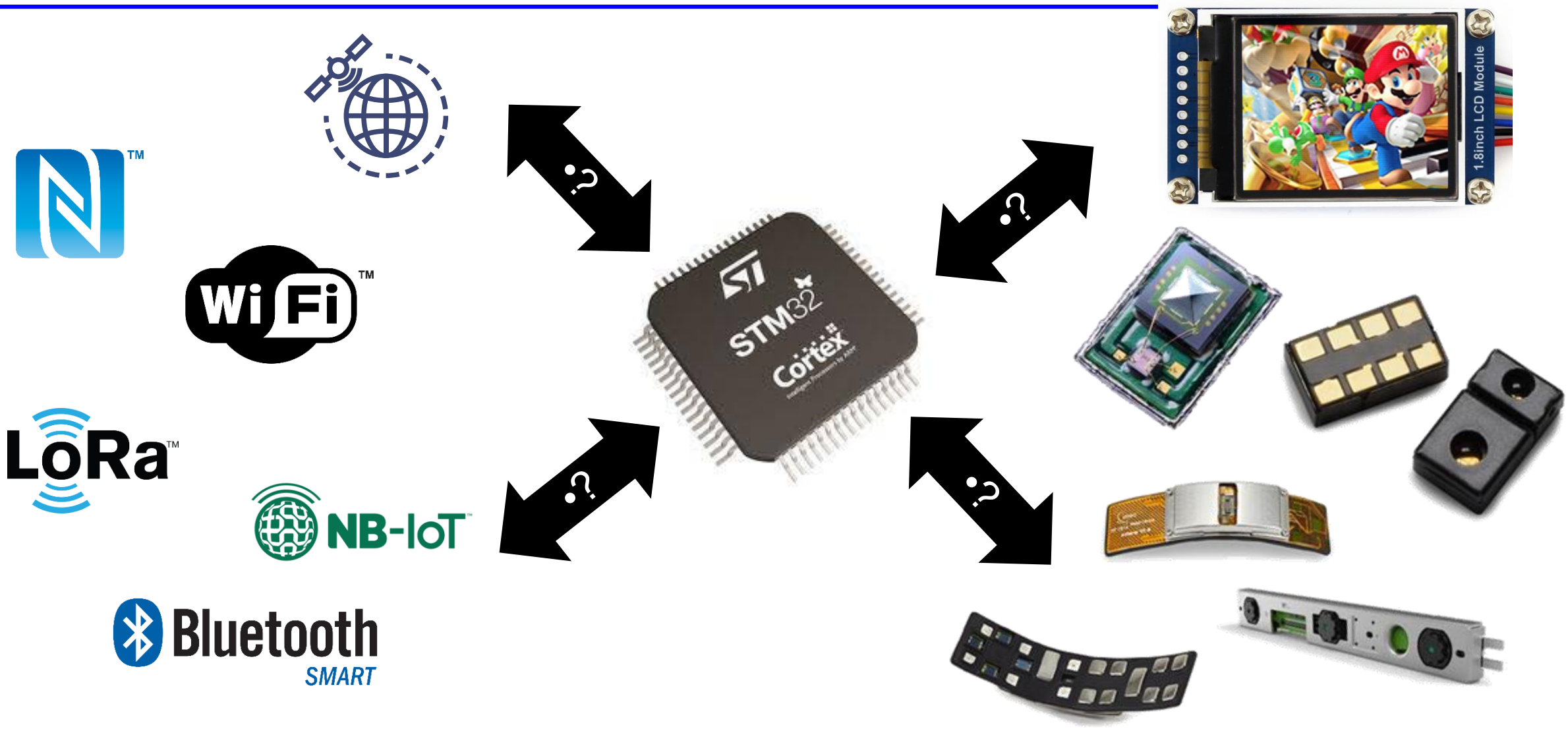
EXTI - Capabilities



- There are 16 EXTI lines connected to GPIOs
- All pins with the same pin number are connected on the same EXTI line (eg. Pin_2 Port A and Pin_2 Port C share the same EXTI2)
- EXTI 16 – 22 are reserved for RTC, USB etc…

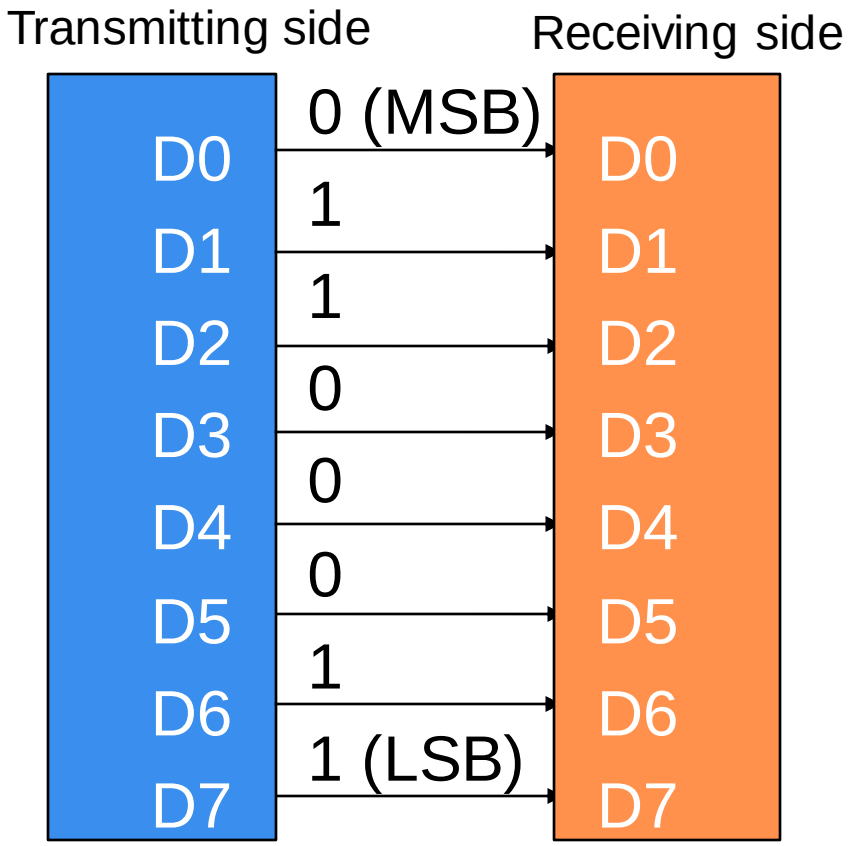
Input and Output

Serial Interfaces: Sensors, Display, Radio etc.

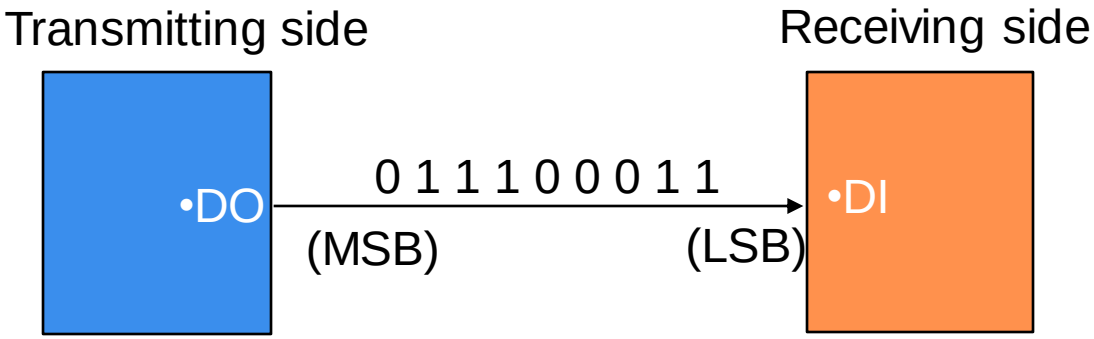


Comparison between Parallel and Serial Communication

Parallel interface example



Serial interface example (LSB first)



Comparison between Parallel and Serial Communication

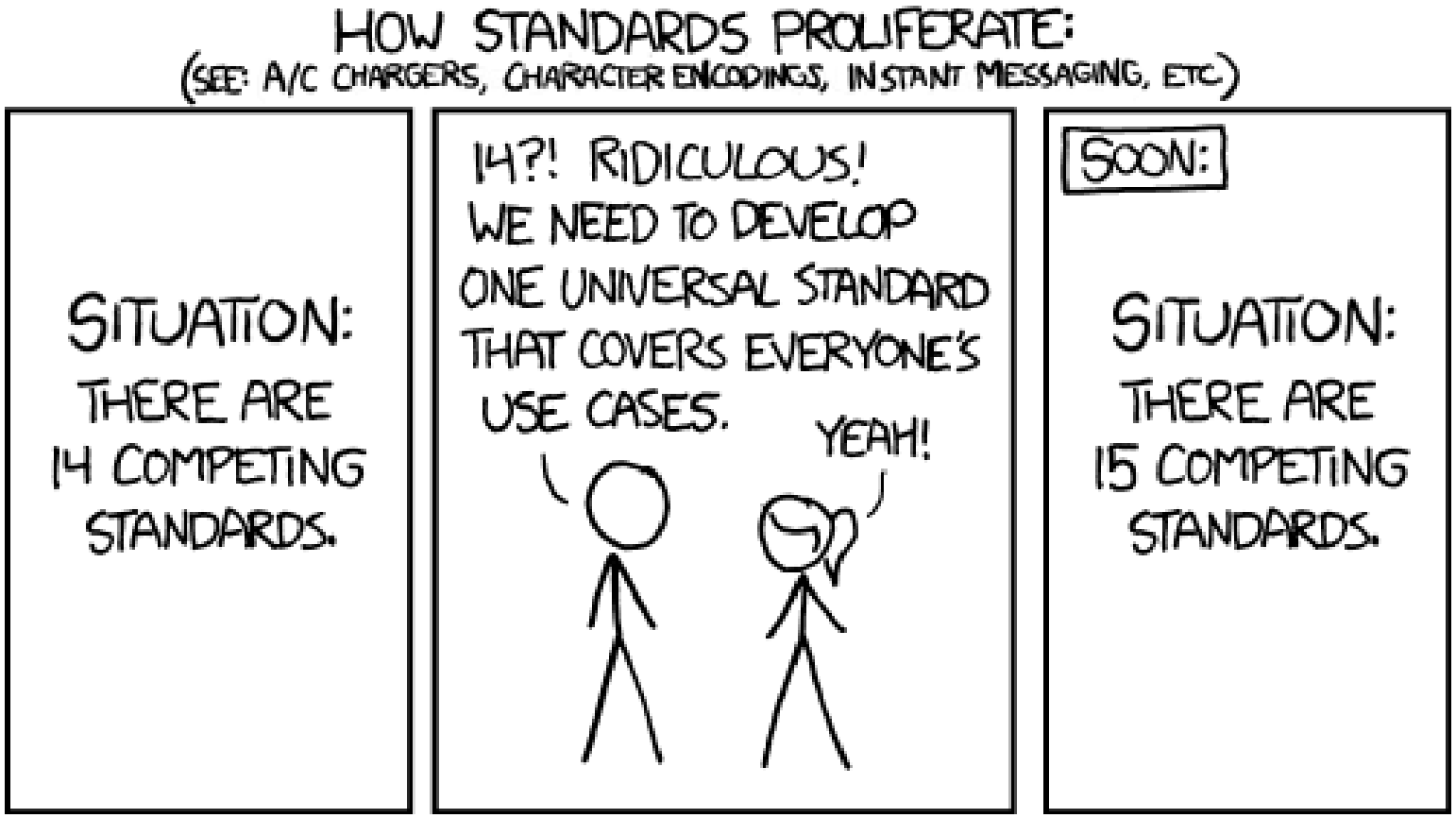
Characteristic	Parallel	Serial
Bus line	One line per bit	One line
Sequence	All bits of one word simultaneously	Sequence of bits
Transmission rate	High	Low
Bus length	Short distances	Short and long distances
Cost	High	Low
Critical characteristics	Synchronization between the different bits is demanding	Asynchronous transmission needs start and stop bits Synchronous transmission needs some other synchronization

MCU Interfaces

- Digital, GPIOs
  - Several protocols for inter-chip communication
- Very often, a processor needs to *exchange information with other processors* or devices. To satisfy various needs, there exists many different *communication protocols*, such as
  - ***UART*** (Universal Asynchronous Receiver-Transmitter)
  - ***SPI*** (Serial Peripheral Interface Bus)
  - ***I2C*** (Inter-Integrated Circuit)
  - ***USB*** (Universal Serial Bus)

- As the principles are similar, we will just explain a representative of an asynchronous protocol (*UART,* no shared clock signal between sender and receiver) and one of a synchronous protocol (*SPI* , shared clock signal).

Serial Interface Standards



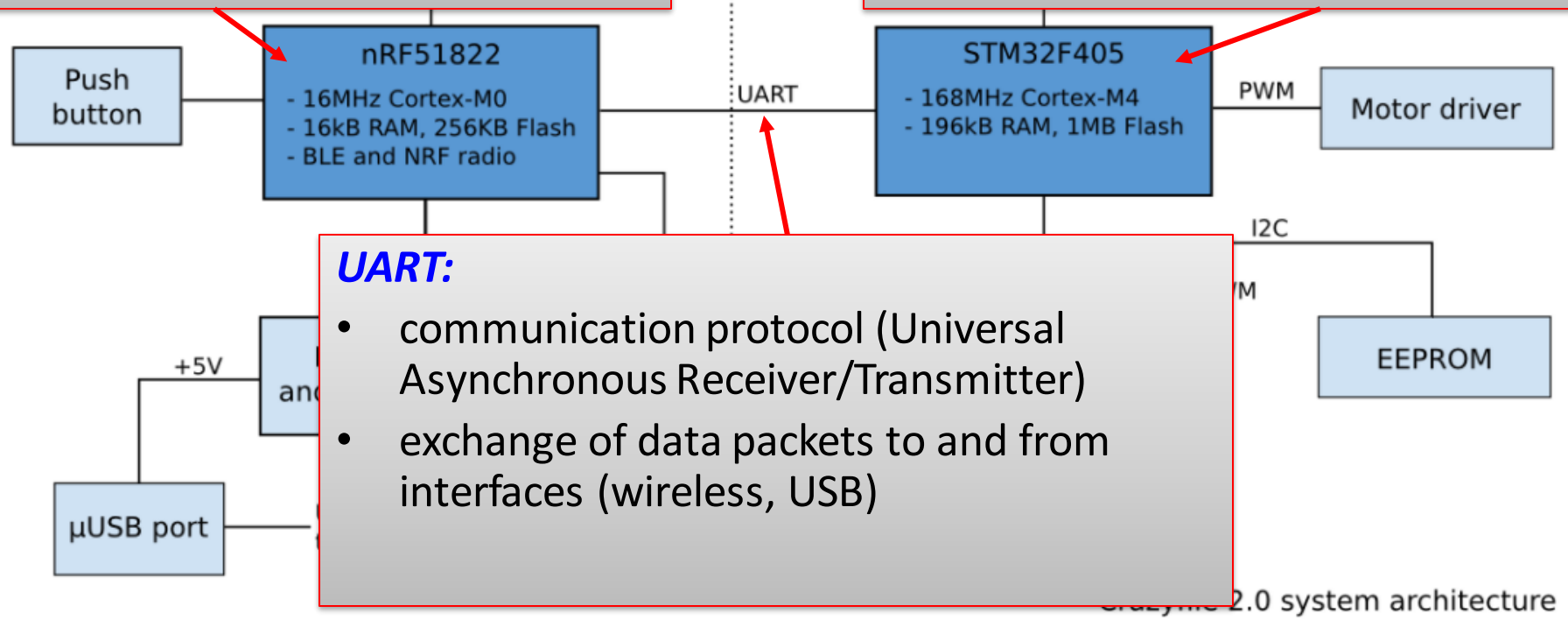
Remember?

low power CPU

- enabling power to the rest of the system
- battery charging and voltage measurement
- wireless radio (boot and operate)
- detect and check expansion boards

higher performance CPU

- sensor reading and motor control
- flight control
- telemetry (including the battery voltage)
- additional user development
- USB connection



UART:

- communication protocol (Universal Asynchronous Receiver/Transmitter)
- exchange of data packets to and from interfaces (wireless, USB)

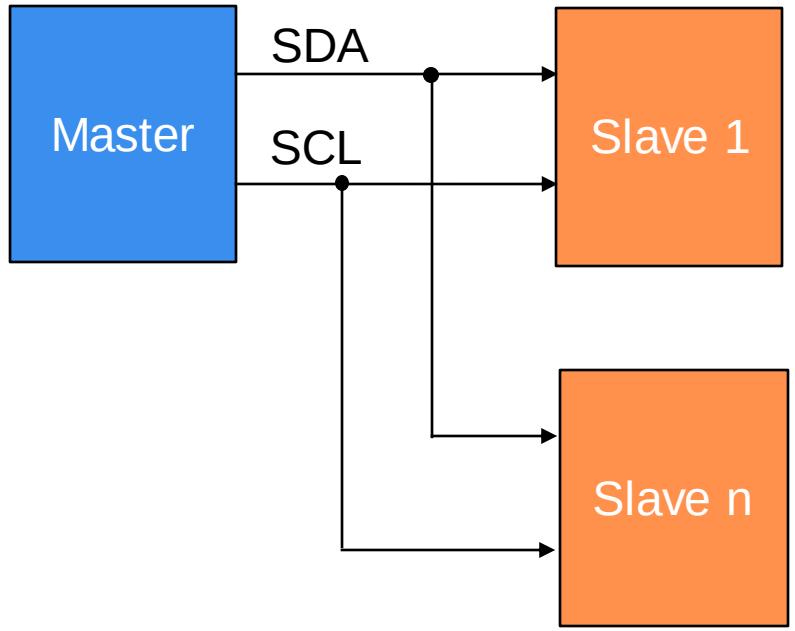
ETH zürich

I^2C – Inter-Integrated Circuit Bus

I^2C: Inter-Integrated Circuit Bus - 1

- Usually pronounced “I-Squared-C”
- Introduced by Philips (now NXP Semiconductors) in 1982
- Used for communication with external peripherals, for example:
  - EEPROMs
  - 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
  - audio processors

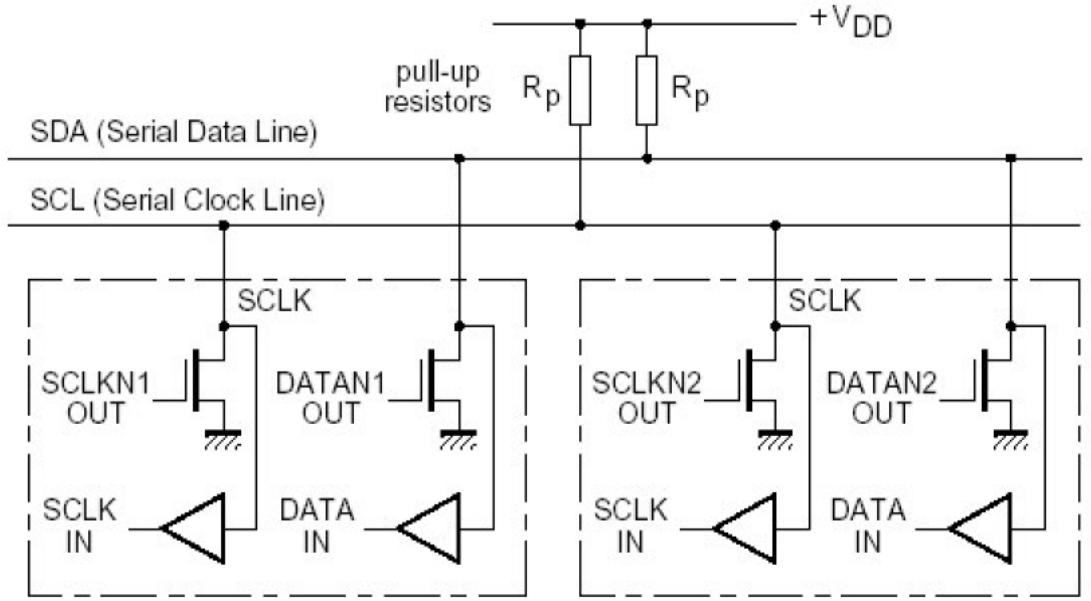
I²C: Inter-Integrated Circuit Bus - 2



- Three supported speed modes:
  - slow (under 100 Kbps)
  - fast (400 Kbps)
  - high-speed (3.4 Mbps) – in I2C v.2.0
- Maximum inter-IC distance of about 3 meters
  - (for moderate speeds, less for high-speed)
- Can support multi-master mode
  - For complex applications
  - Communication is always started by a master, both in single-master and multi-master mode
- Half-duplex synchronous communication scheme
  - the master of the communication generates the clock (SCL) on which data (SDA) is synchronized

I²C: Inter-Integrated Circuit Bus - 3

- Based on two lines:
  - SCL (serial clock)
  - SDA (serial data)



Pull-Up resistors, **Pull-Down** by open-drain drivers

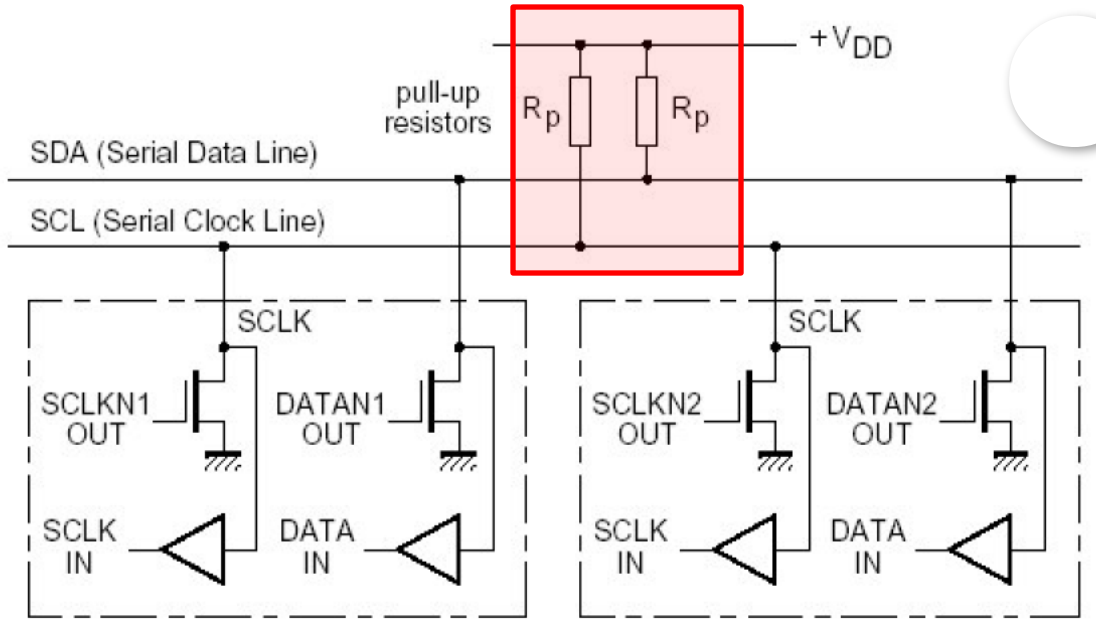
Wired-AND: if *any* driver pulls down, the line is low (avoids short circuits)

Any module on the bus can act as **master**, **slave** or both

typical case: MCU is the master, peripherals/sensors are slaves

I²C: Inter-Integrated Circuit Bus - 3

- Based on two lines:
  - SCL (serial clock)
  - SDA (serial data)



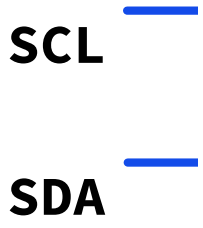
Pull-Up resistors, **Pull-Down** by open-drain drivers

Wired-AND: if *any* driver pulls down, the line is low (avoids short circuits)

Any module on the bus can act as **master**, **slave** or both

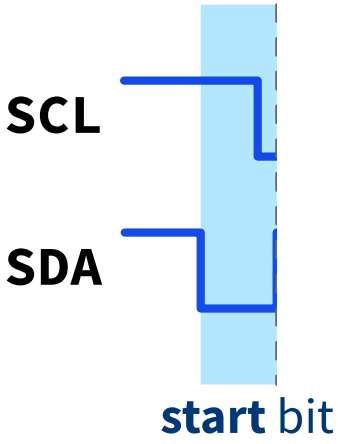
typical case: MCU is the master, peripherals/sensors are slaves

I²C: Interface Protocol



- In idle, both **SCL** and **SDA** are pulled-up to **1**

I²C: Interface Protocol



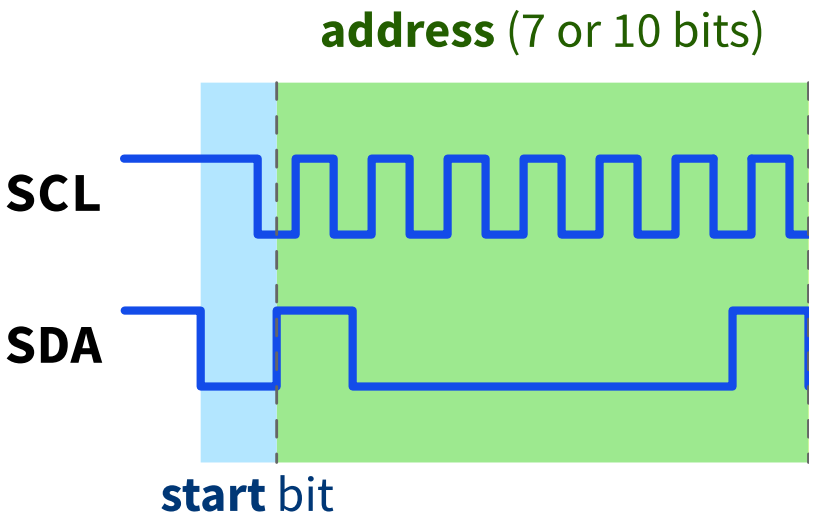
To start the communication, the **master**:

asserts the **start** bit (**SDA** 1→0 transition while **SCL** is **still 1**)

then, it starts generating the **SCL** clock

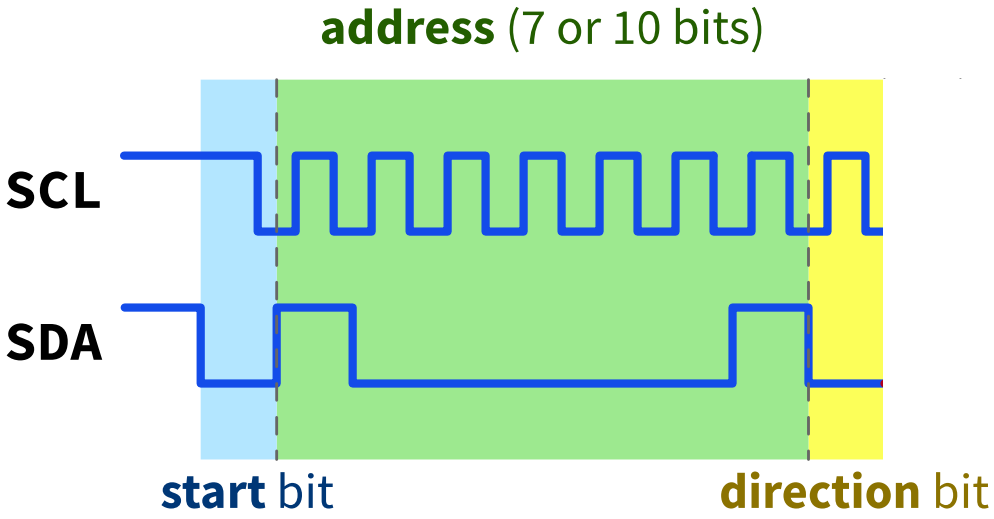
except for the start and stop bits, **SDA** transitions *only* when **SCL** is **0**

I²C: Interface Protocol



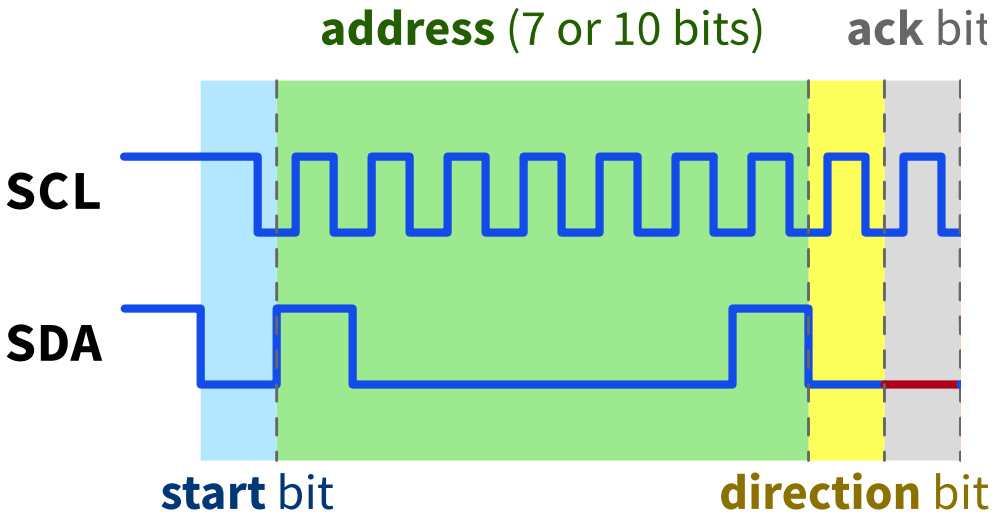
2. The **master** transmits the **slave** **address**:
   - **broadcasted** to all devices on the I²C bus
   - used to select the target **slave**
   - either **7 bits** or **10 bits** (newer devices – 7 bits address space is small!)
   - *in the example, the address is 7'b1000001*

I²C: Interface Protocol



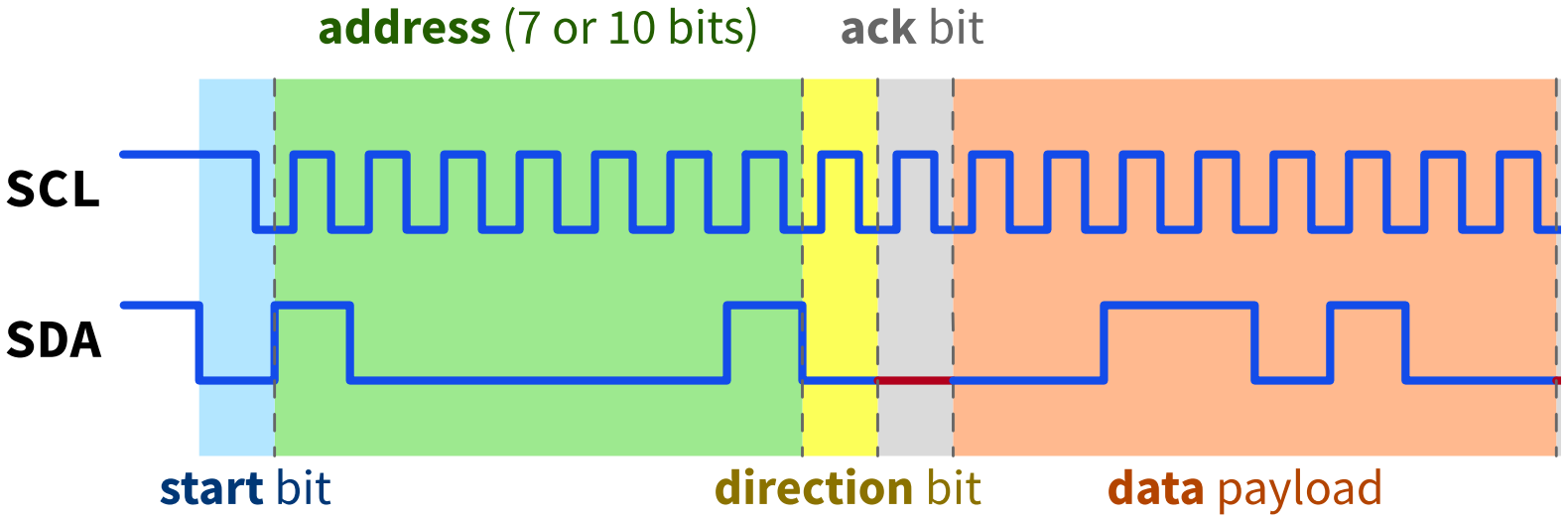
3. The **master** transmits a **direction** bit:
   - a **0** for **master → slave** (write) transfer
   - a **1** for **slave → master** (read) transfer
   - *in the example, suppose a write transfer*

I²C: Interface Protocol



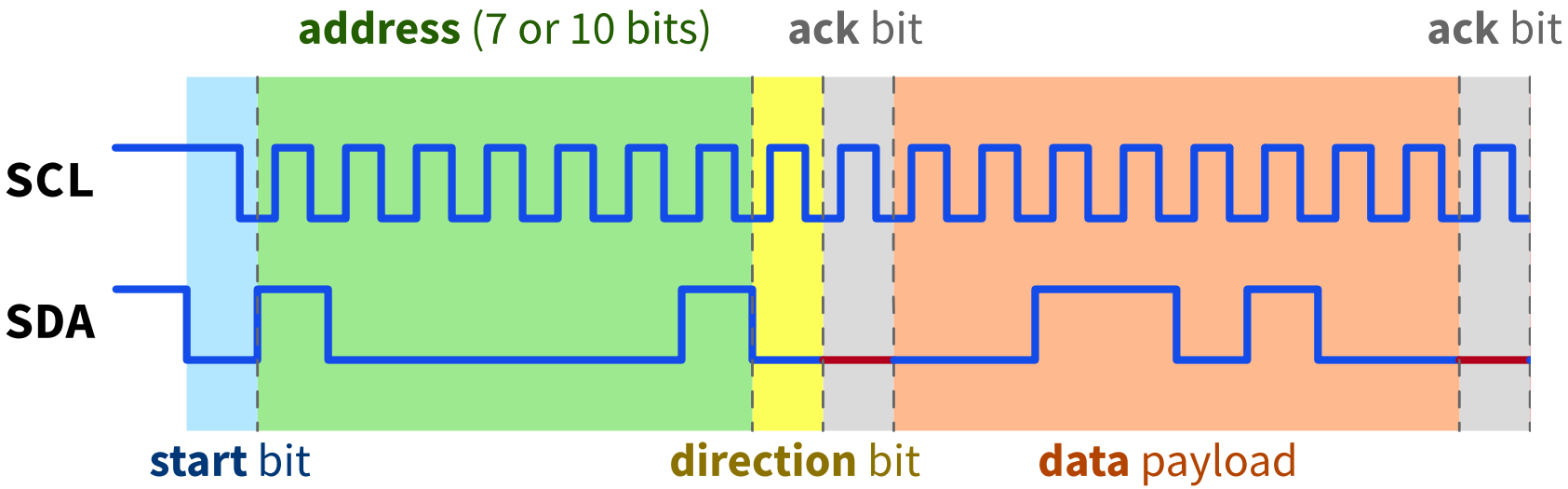
4. The **slave** then **acknowledges** reception:
   - by driving **SDA** to **0**
   - if not acknowledged, the transaction must be repeated by the master

I²C: Interface Protocol



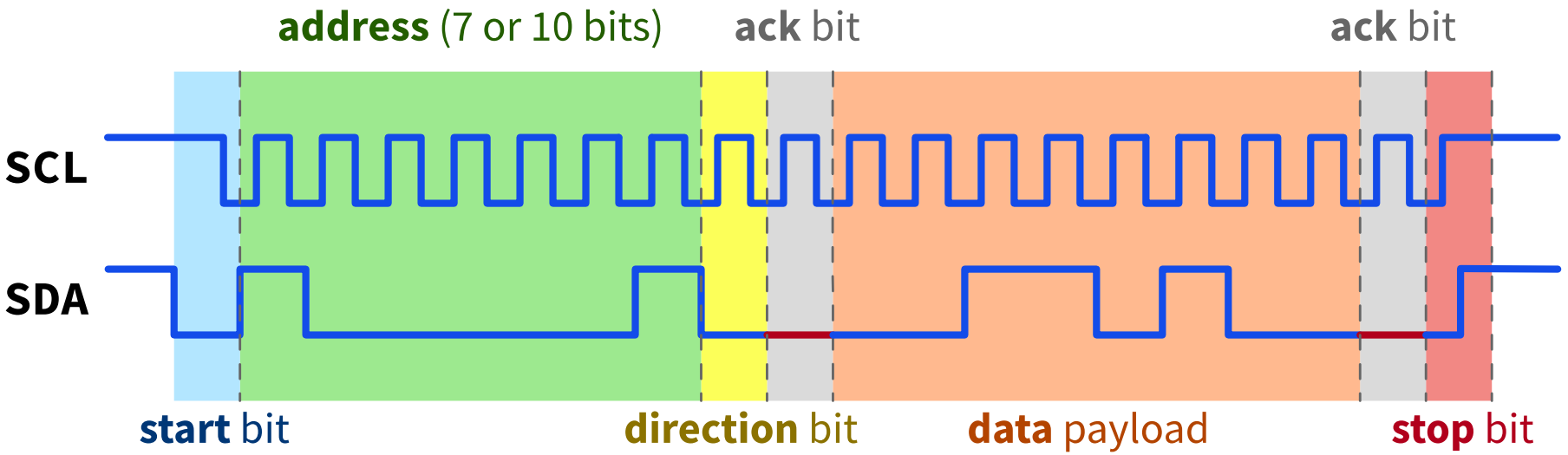
5. The **master** transmits its data payload:
   - each payload packet is **8 bits**
   - there might be more than one packet, depending on application
   - *in the example, data payload is 8'b00110100*

I²C: Interface Protocol



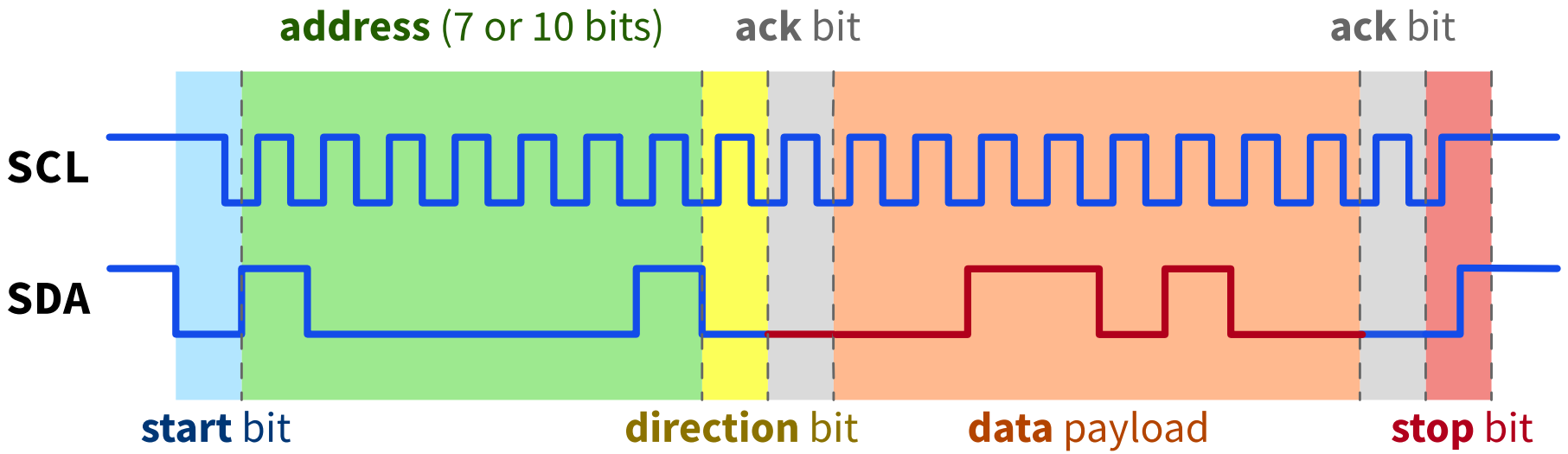
6. The **slave** acknowledges reception of the data packet:
   - **1 ack bit** every **8 payload bits**
   - slave must acknowledge each packet

I²C: Interface Protocol



7. At the end of the transfer, the **master** transmits a **stop bit**:
   - 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**

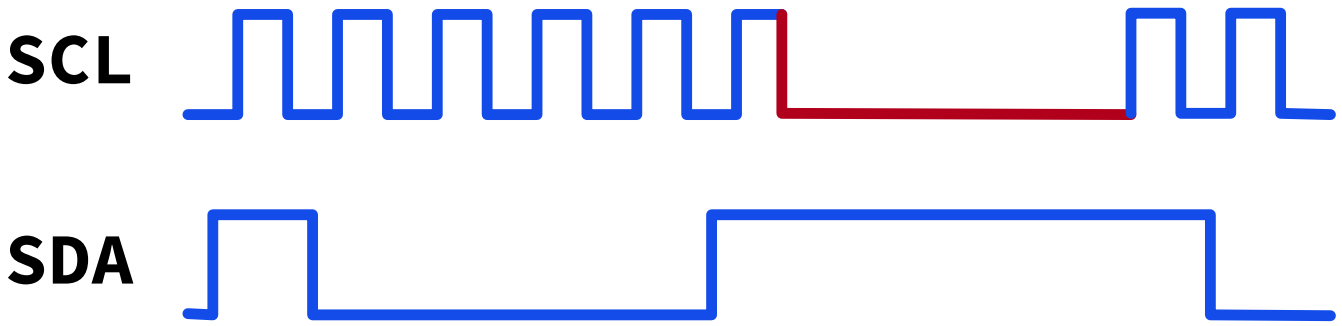
I²C: Interface Protocol



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

- the **slave** drives **SDA** when transmitting the data byte
- the **master** acknowledges the transfer

I²C: Interface Protocol

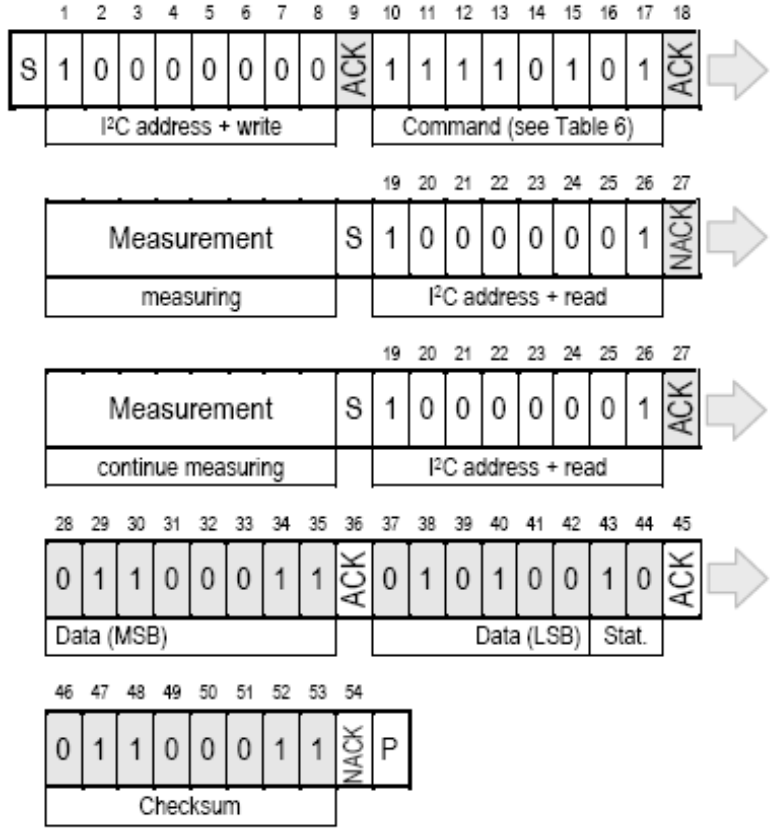


Slave can ask for more time to process a bit by **clock stretching**:

- drive **SCL** to **0** if in need of more processing time

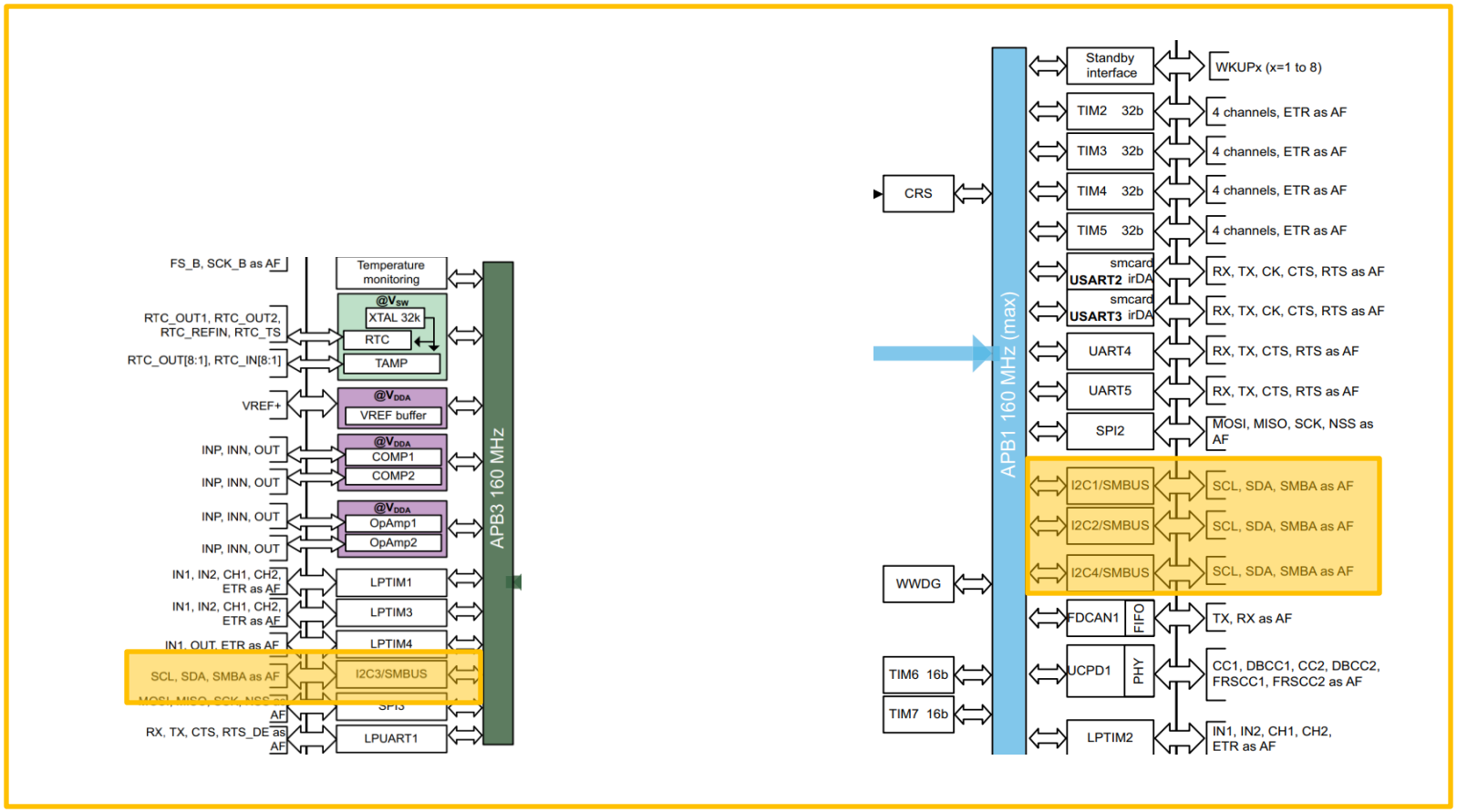
I²C: Interface Protocol

Example of **MCU** – **sensor** communication (data acquisition) via I²C bus

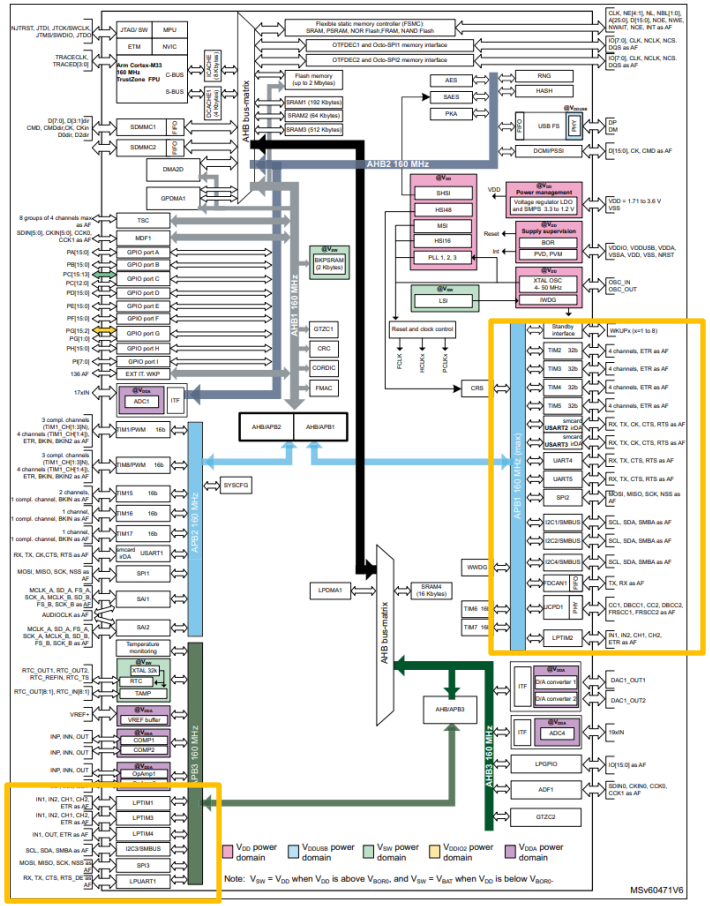


I²C – STM32U585xx

- 4 x I²C



STM32U585xx block diagram

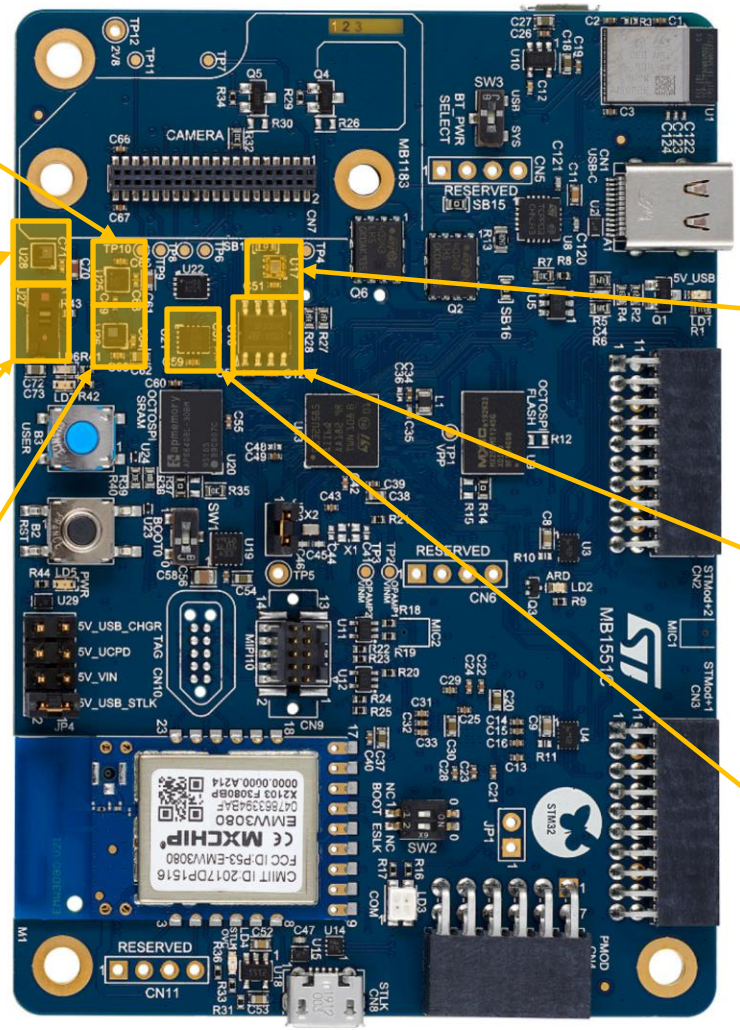


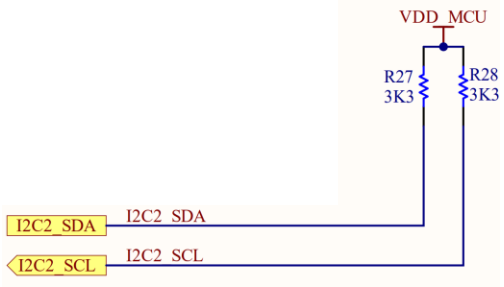
https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html

→ *Datasheet, page 21*

I²C – Connected Sensors

- 3-axis magnetometer
  - IIS2MDCTR
  - Read=00111101(3Dh)
  - Write=00111100(3Ch)
- Humidity and temperature
  - HTS221
  - Read=10111111 (BFh)
  - Write=10111110 (BEh)
- ToF, gesture-detection
  - VL53L5CXV0GC/1
  - Read=01010011 (53h)
  - Write=01010010 (52h)
- MEMS nano pressure sensor
  - LPS22HH
  - Read=10111011 (BBh)
  - Write=10111010 (BAh)

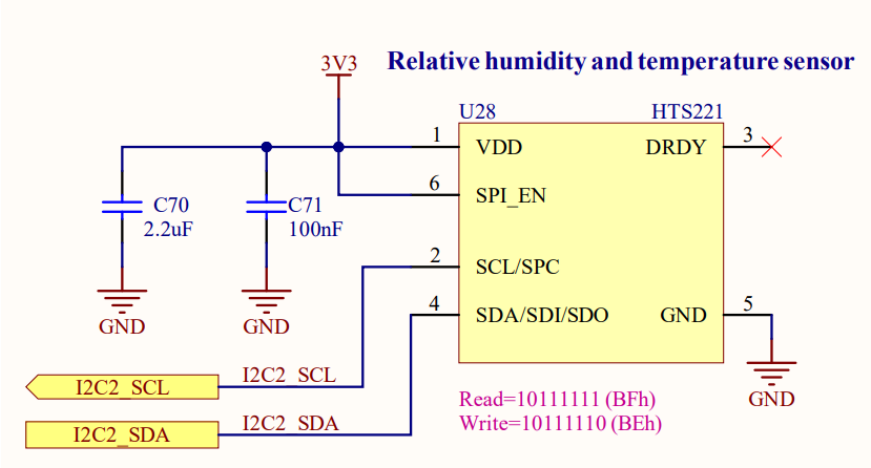


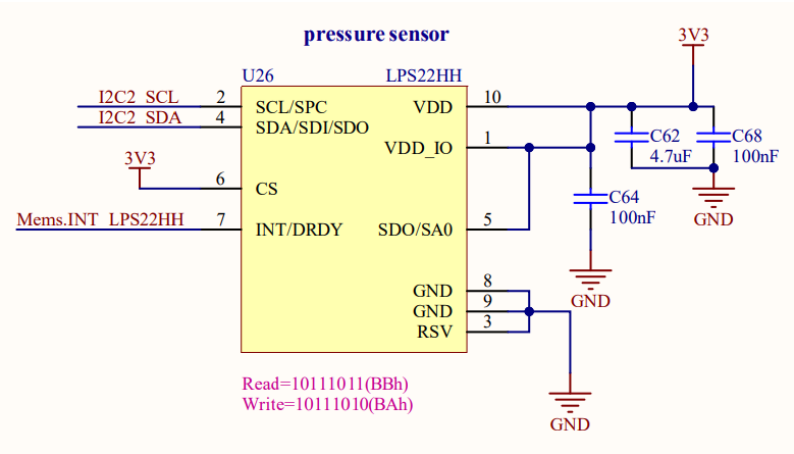


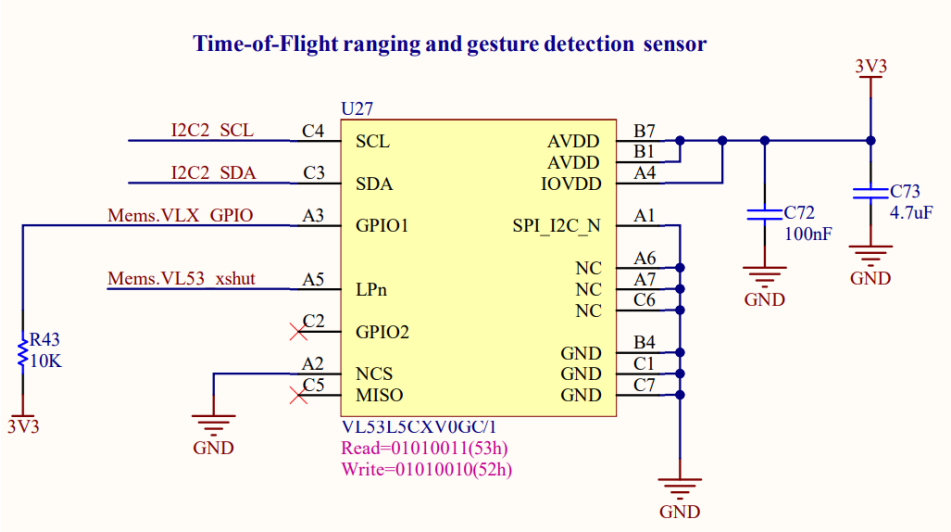
- I2C2 @ VMCU
- PH4 / PH5

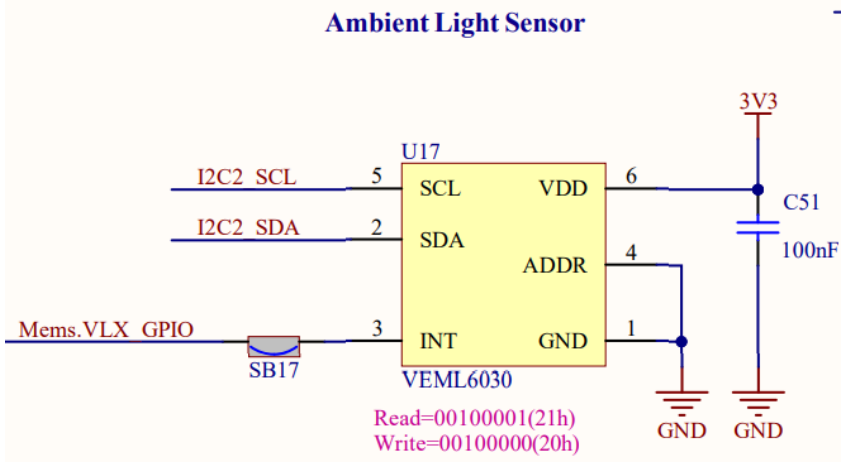
- Ambient light sensor
  - VEML6030
  - Read=00100001 (21h)
  - Write=00100000 (20h)
- Authentication and security
  - STSAFE-A110
  - Read=01000000 (40h)
  - Write=01000001 (41h)
- 3D accelerometer and 3D gyroscope
  - ISM330DHCX
  - Read=11010101 (D5h)
  - Write=11010100 (D4h)

I²C – Connected Sensors Schematic

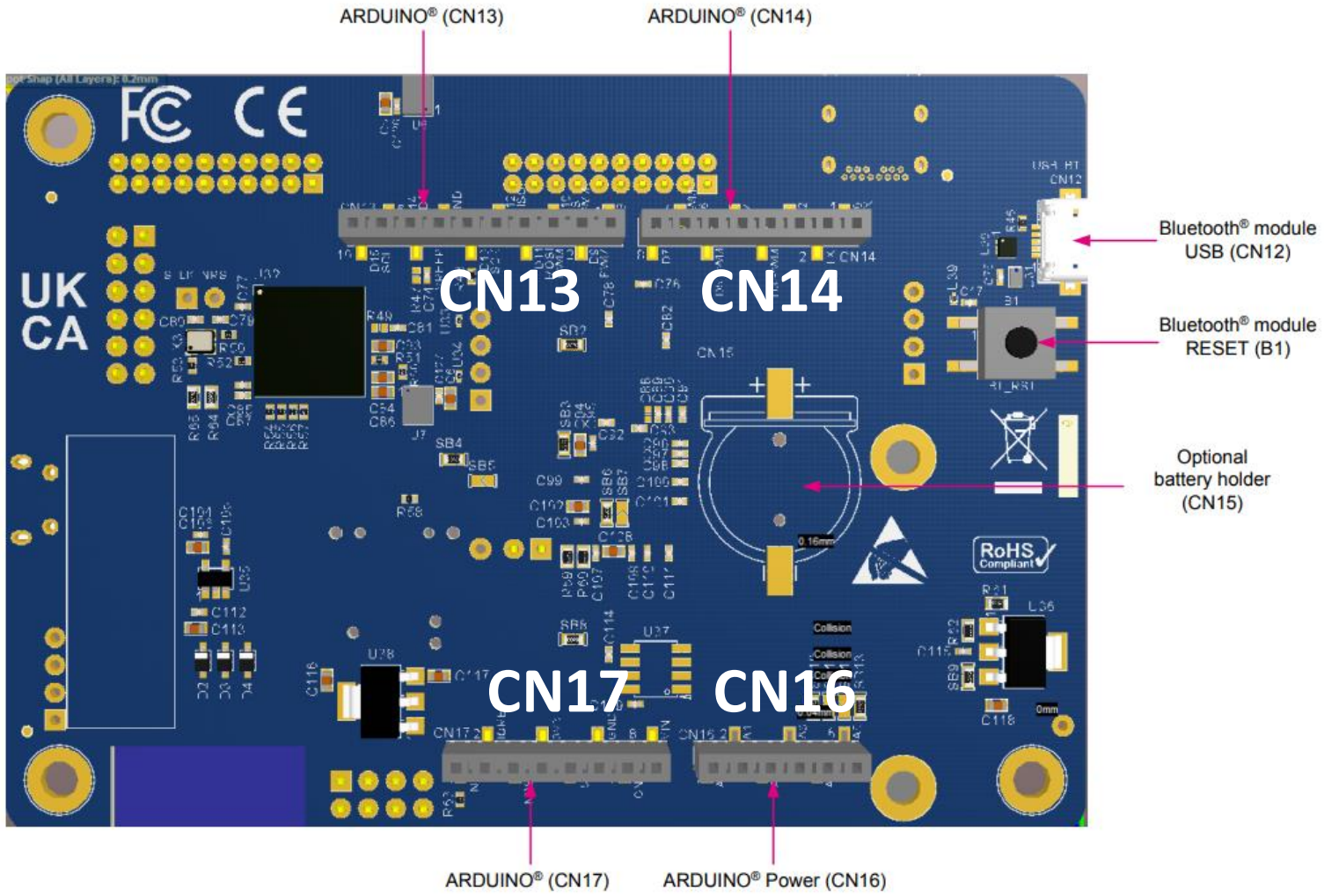








I²C – External Connection



Connector	Pin number	Pin name	Signal name	STM32L 4+ pin	Function
CN17	1	NC	-	-	-
	2	IOREF	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	3V3	-	-	3.3 V I/O
	5	5V	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
	8	VIN	-	-	Power input
CN16	1	A0	ARD_ADC_A0	PC0	ADC
	2	A1	ARD-ADC_A1	PC2	ADC
	3	A2	ARD-ADC_A2	PC4	ADC
	4	A3	ARD-ADC_A3	PC5	ADC
	5	A4	ARD-ADC_A4	PA7	ADC / I2C1_SDA
	6	A5	ARD-ADC_A5	PB0	ADC / I2C1_SCL
CN13	10	SCL/D15	I2C1_SCL	PB8	I2C1_SCL
	9	SDA/D14	I2C1_SDA	PB9	I2C1_SDA
	8	VREFP	VREFP	-	VREFP
	7	GND	GND	-	GND
	6	SCK/D13	ARD.D13_SPI1_SCK	PE13	SPI1_SCK / LD2
	5	MISO/D12	ARD.D12_SPI1_MISO	PE14	SPI1_MISO
	4	PWM/MOSI/D11	ARD.D11_SPI1_MOSI	PE15	SPI1_MOSI / TIM1_CH4N
	3	PWM/CS/D10	ARD.D10_TIM_SPI1_NSS	PE12	SPI1_NSS / TIM1_CH3N
	2	PWM/D9	ARD.D9_TIM	PA8	TIM1_CH1
	1	D8	ARD.D8_IO	PC1	GPIO
CN14	8	D7	ARD.D7_IO	PF13	GPIO
	7	PWM/D6	ARD.D6_TIM	PB6	TIM4_CH1
	6	PWM/D5	ARD.D5_TIM	PE0	TIM16_CH1
	5	D4	ARD.D4_INT	PE7	GPIO
	4	PWM/D3	ARD.D3_TIM	PB2	TIM8_CH4N
	3	D2	ARD.D2_IO	PD15	GPIO
	2	TX/D1	ARD.D1_TX	PD8	UART3_TX
	1	RX/D0	ARD.D0_RX	PD9	UART3_RX

https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html

→ *Documents → UM2839, page 10*

I²C – Typical Datasheet - HTS221

Figure 1. HTS221 block diagram

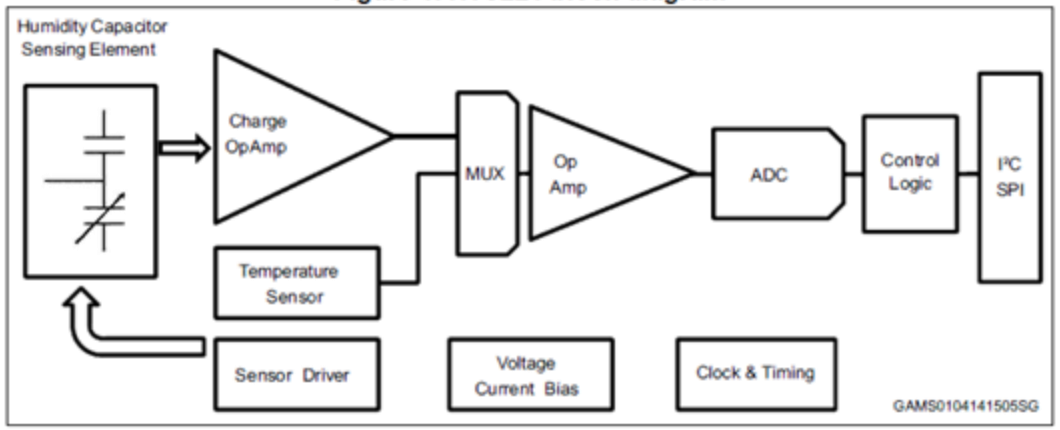


Figure 2. Pin configuration (bottom view)

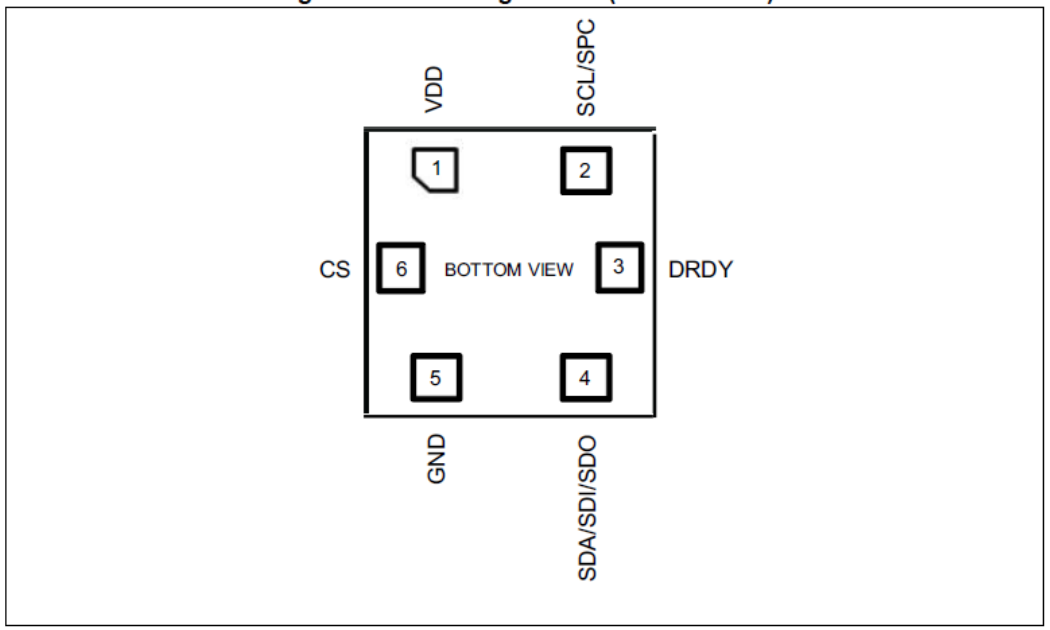


Table 10. SAD + Read/Write patterns

Command	SAD[6:0]	R/W	SAD+R/W
Read	1011111	1	10111111 (BFh)
Write	1011111	0	10111110 (BEh)

- Address sometimes hidden in the text.

- What happens if I want to use one sensor twice on the bus?

Table 2. Pin description

Pin n°	Name	Function
1	V_{DD}	Power supply
2	SCL/SPC	I²C serial clock (SCL) SPI serial port clock (SPC)
3	DRDY	Data Ready output signal
4	SDA/SDI/SDO	I²C serial data (SDA) 3-wire SPI serial data input /output (SDI/SDO)
5	GND	Ground
6	SPI enable	I²C/SPI mode selection (1: SPI idle mode / I²C communication enabled; 0: SPI communication mode / I²C disabled)

I²C – Typical Datasheet - HTS221

Table 15. Register address map

Name	Type	Register address (hex)	Default (hex)
Reserved		00-0E	Do not modify
WHO_AM_I	R	0F	BC
AV_CONF	R/W	10	1B
Reserved		11-1C	Do not modify
CTRL_REG1	R/W	20	0
CTRL_REG2	R/W	21	0
CTRL_REG3	R/W	22	0
Reserved		23-26	Do not modify
STATUS_REG	R	27	0
HUMIDITY_OUT_L	R	28	Output
HUMIDITY_OUT_H	R	29	Output
TEMP_OUT_L	R	2A	Output
TEMP_OUT_H	R	2B	Output
Reserved		2C-2F	Do not modify
CALIB_0..F	R/W	30-3F	Do not modify

WHO_AM_I (0Fh)

Device identification

7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	0

This read-only register contains the device identifier, set to BCh

Configuration

Data

Calibration

7.9 TEMP_OUT_L (2Ah)

Temperature data (LSB)

7	6	5	4	3	2	1	0
TOUT7	TOUT6	TOUT5	TOUT4	TOUT3	TOUT2	TOUT1	TOUT0

[7:0]	TOUT7 - TOUT0: Temperature data LSB (see TEMPERATURE_OUT_H)

7.10 TEMP_OUT_H (2Bh)

Temperature data (MSB)

15	14	13	12	11	10	9	8
TOUT15	TOUT14	TOUT13	TOUT12	TOUT11	TOUT10	TOUT9	TOUT8

[15:8]	TOUT15 - TOUT8: Temperature data MSB.

Temperature data are expressed as TEMP_OUT_H & TEMP_OUT_L as 2's complement numbers.

ETH zürich

SPI – Serial Peripherals Interface

SPI: Serial Peripheral Interface - 1

NXP – Qualcomm (almost) - NXP

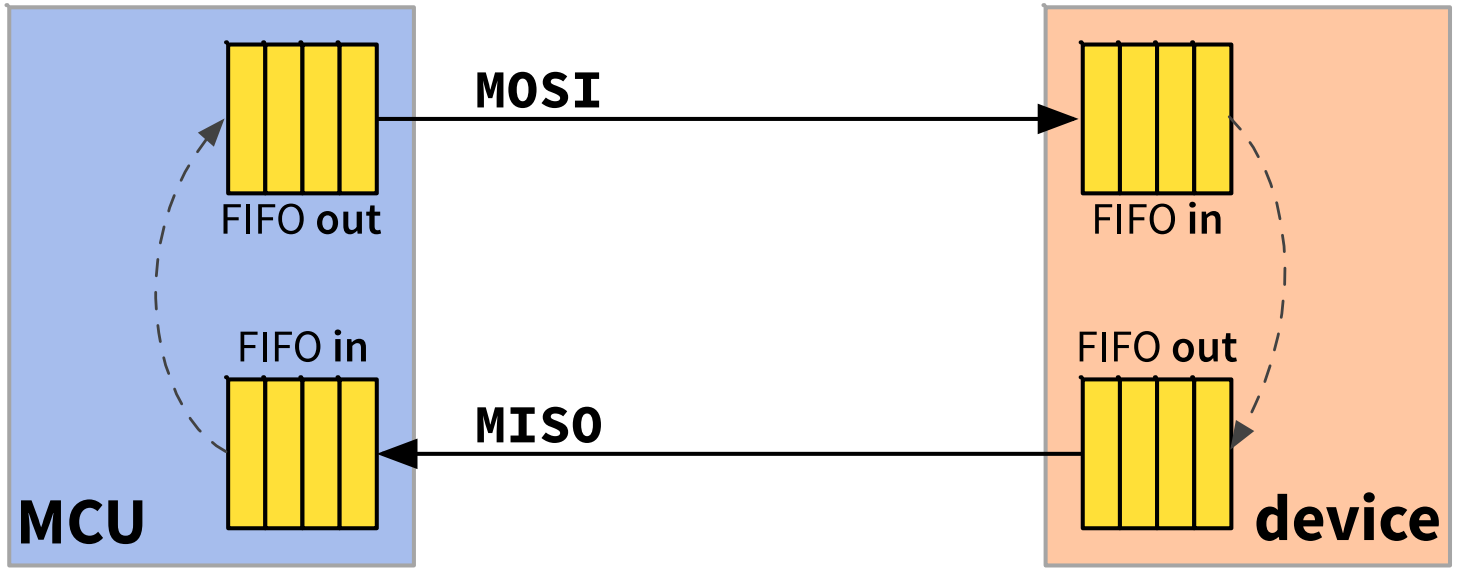
- Introduced by Motorola (now Freescale Semiconductors) for the MC68HCxx line of microcontrollers
- Use cases are generally similar to I2C
- Generally faster than I2C (up to several Mbit/s)
  - Short-distance (i.e. on printed circuit boards)
- Single-master, multiple slave
  - needs one chip select per slave device (no broadcast addressing)
- Full-duplex synchronous communication scheme
  - master drives the clock (SCLK or SCK)
  - clock polarity (i.e. write/read edges) and phase depend on specific application!

SPI: Serial Peripheral Interface - 2

- Based on two data and two control 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! Some possible alternatives:
  - SDI (SPI data in) instead of MISO
  - SDO (SPI data out) instead of MOSI
  - SCLK, CLK, SPC, … instead of SCK
  - CS, SS (slave select), SSN (slave select, active low) … instead of CSN

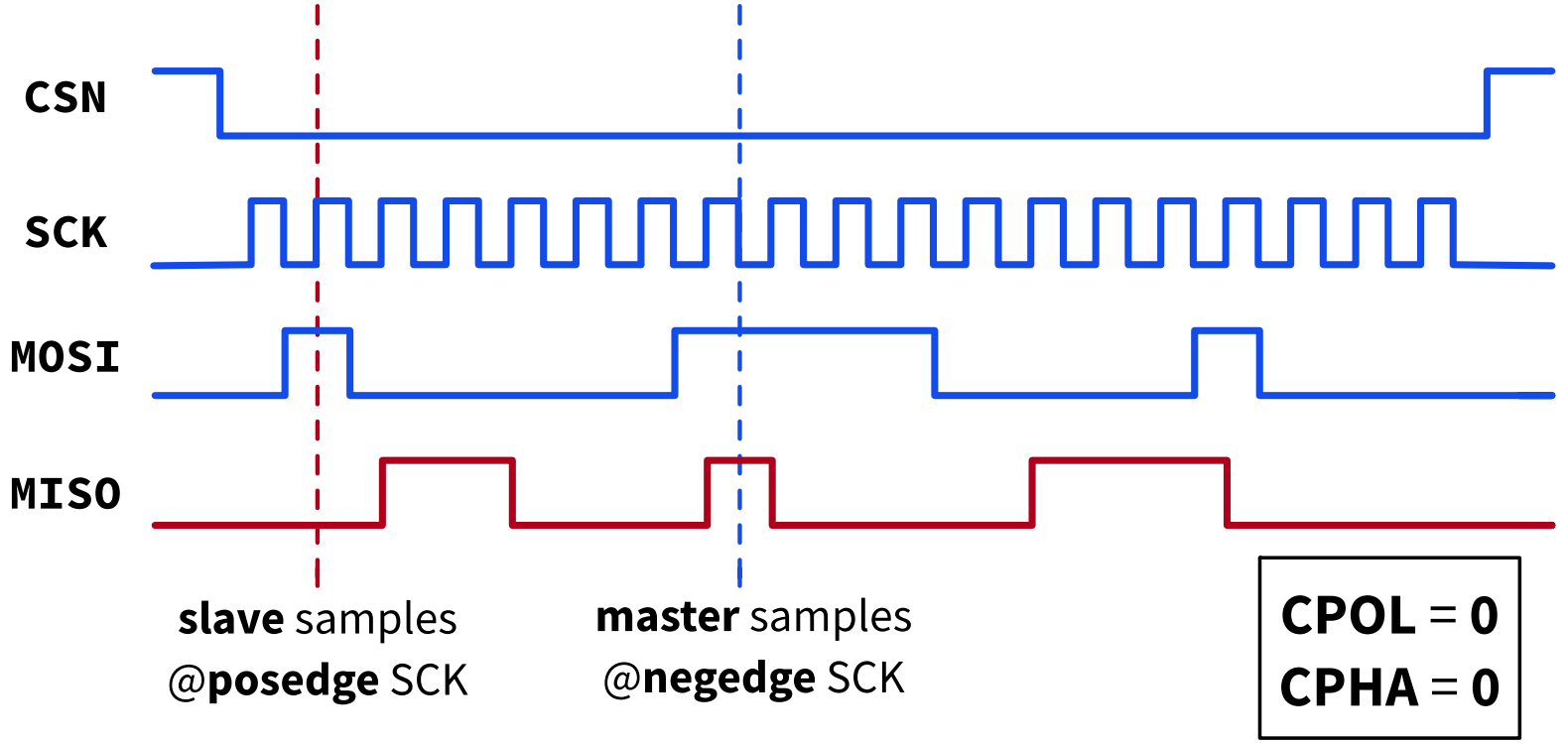
SPI: Serial Peripheral Interface - 3

- Full-duplex transfer: data is streamed between master and slave shift-registers / 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
- Processing / sensing / … happens in between (dashed line)



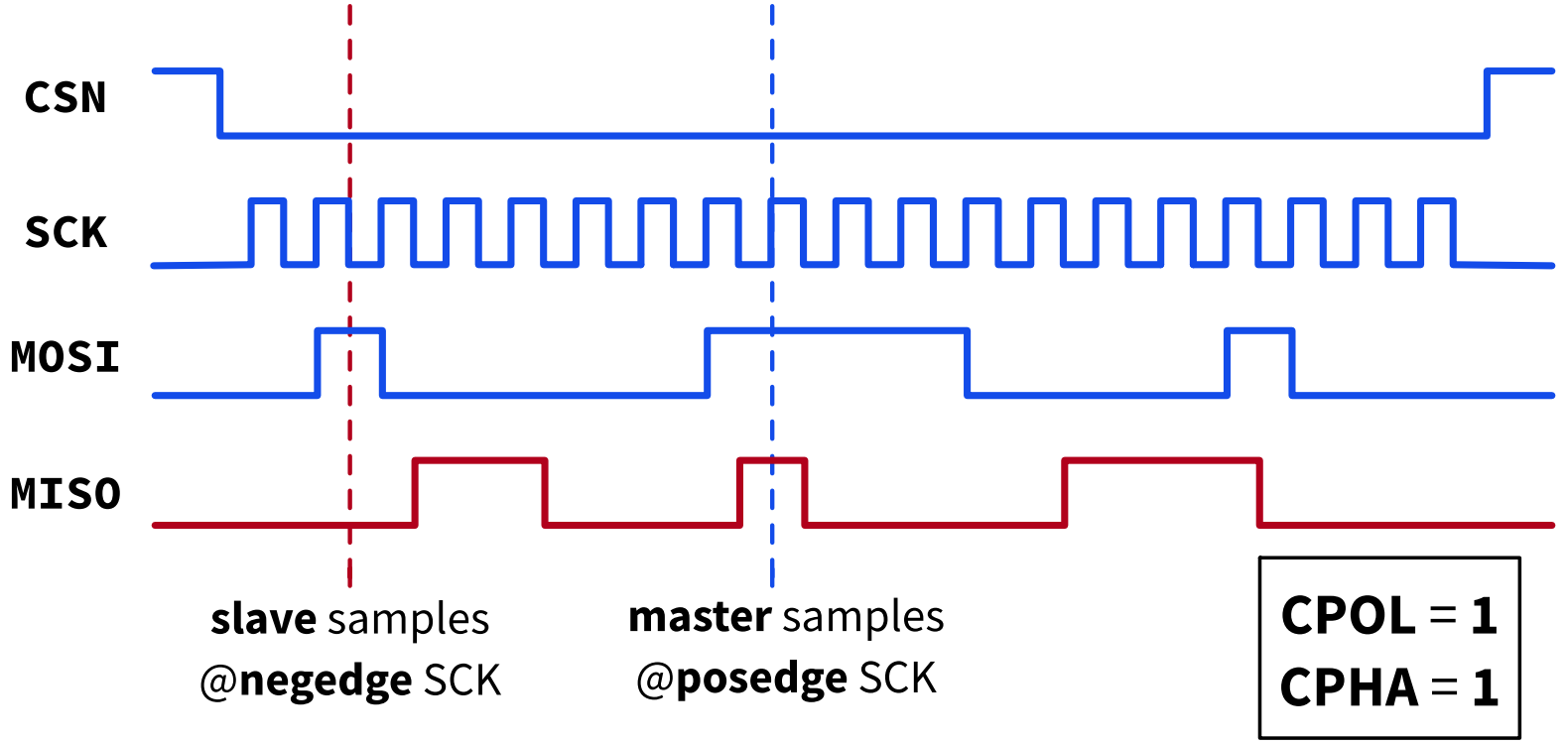
SPI: Interface Protocol - 1

- Four operating modes, varying by clock polarity (CPOL) and phase (CPHA):
  - polarity sets the initial value of the SPI clock signal
  - phase defines the edge at which MOSI is switched and the one at which MISO is sampled



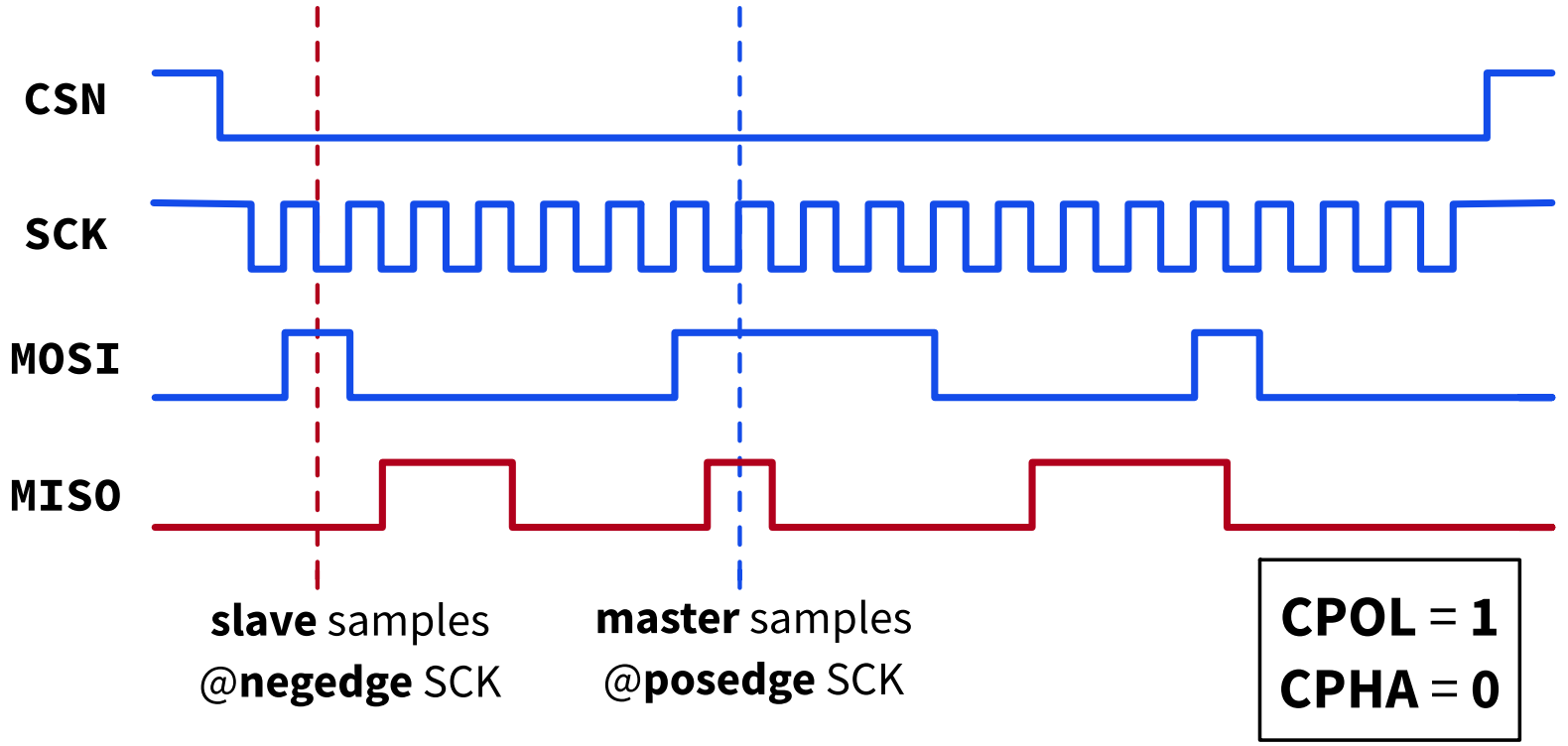
SPI: Interface Protocol - 1

- Four operating modes, varying by clock polarity (CPOL) and phase (CPHA):
  - polarity sets the initial value of the SPI clock signal
  - phase defines the edge at which MOSI is switched and the one at which MISO is sampled



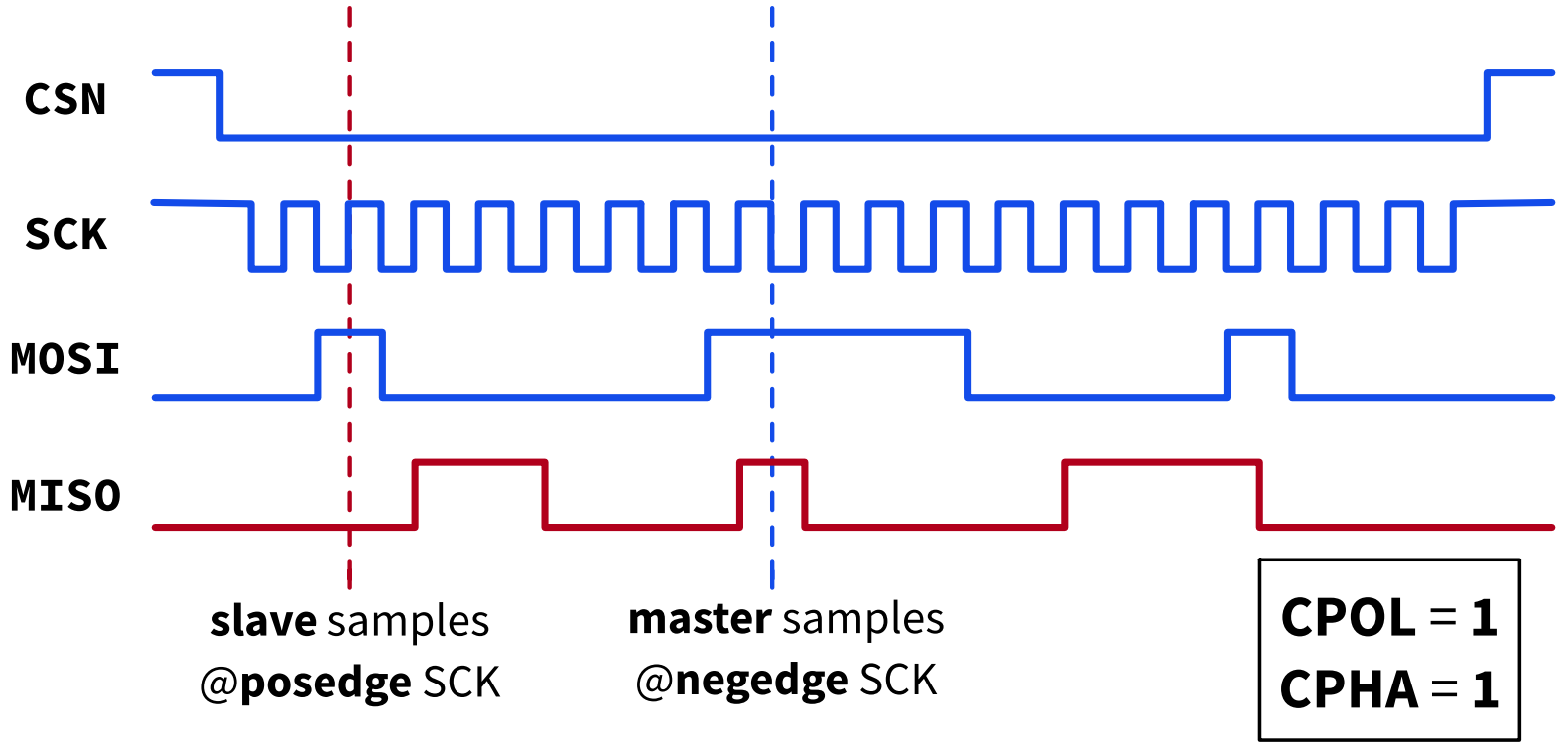
SPI: Interface Protocol - 1

- Four operating modes, varying by clock polarity (CPOL) and phase (CPHA):
  - polarity sets the initial value of the SPI clock signal
  - phase defines the edge at which MOSI is switched and the one at which MISO is sampled



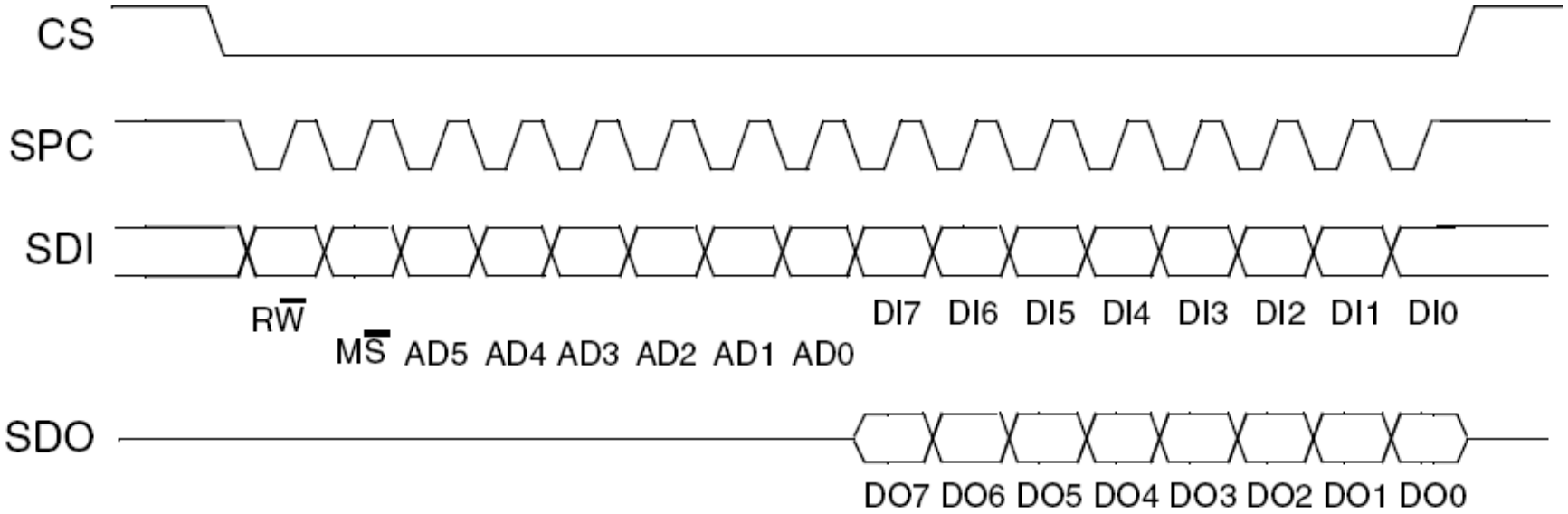
SPI: Interface Protocol - 1

- Four operating modes, varying by clock polarity (CPOL) and phase (CPHA):
  - polarity sets the initial value of the SPI clock signal
  - phase defines the edge at which MOSI is switched and the one at which MISO is sampled



SPI: Interface Protocol - 2

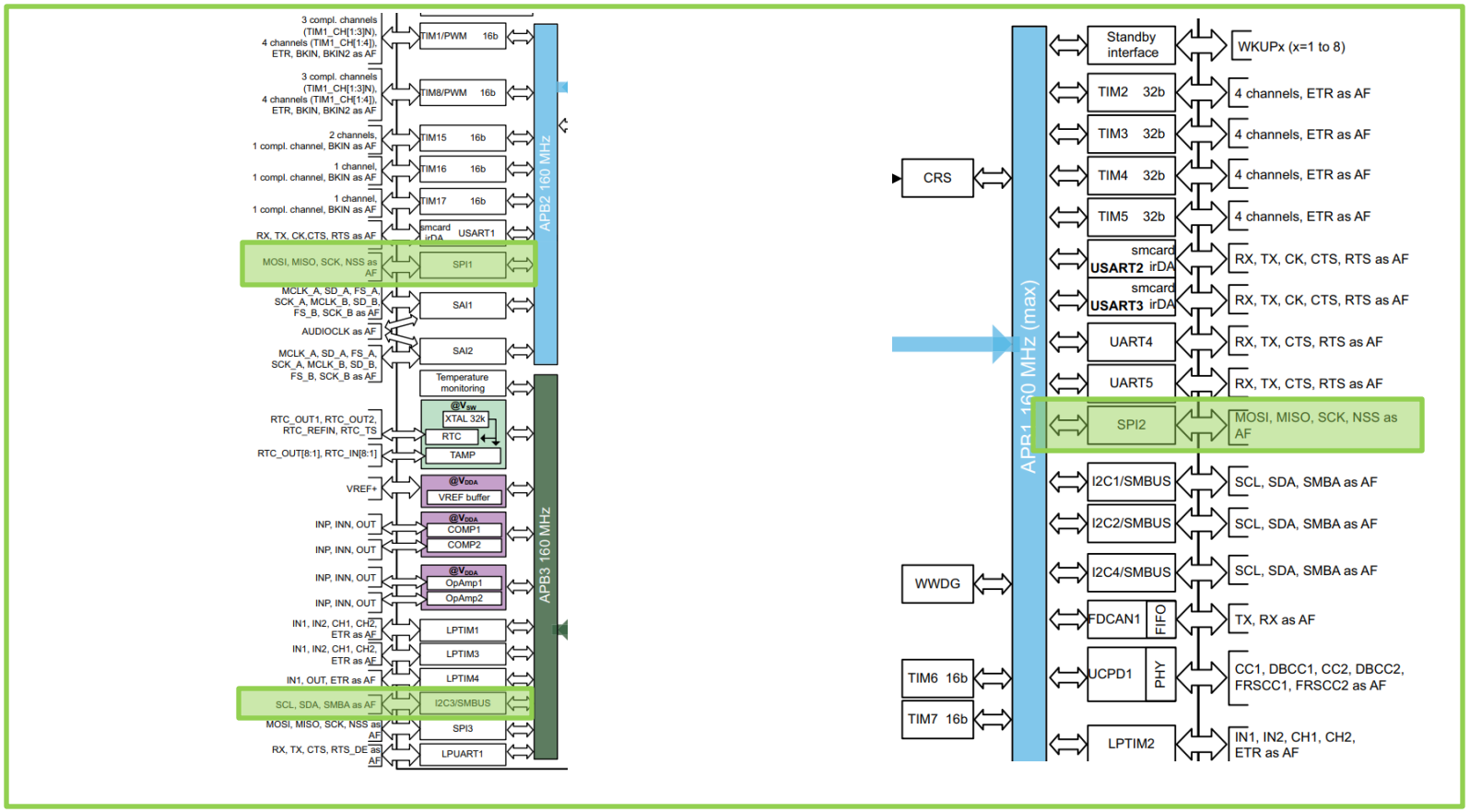
- Master completely in charge of transfer
  - no ack, no clock stretching contrarily to I2C
- More complex behavior than simple data streaming can be mapped on top of SPI protocol
  - e.g. command + address + data streaming



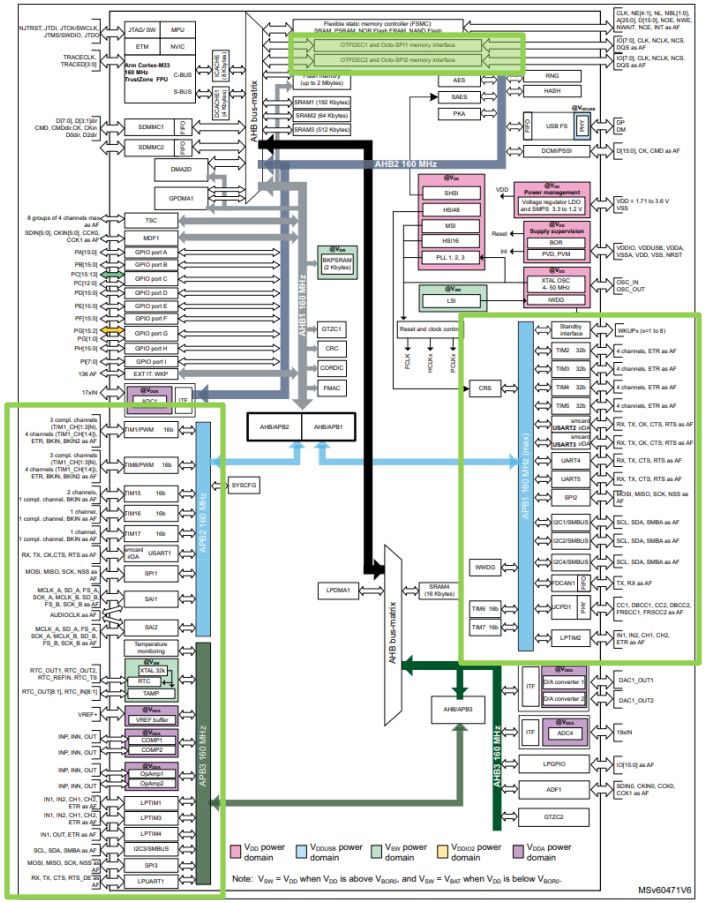
Saveral names!

SPI – STM32U585xx

- 3 x SPI
- 1 x SPI with dual OCTOSPI



STM32U585xx block diagram



https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html

→ *Datasheet, page 21*

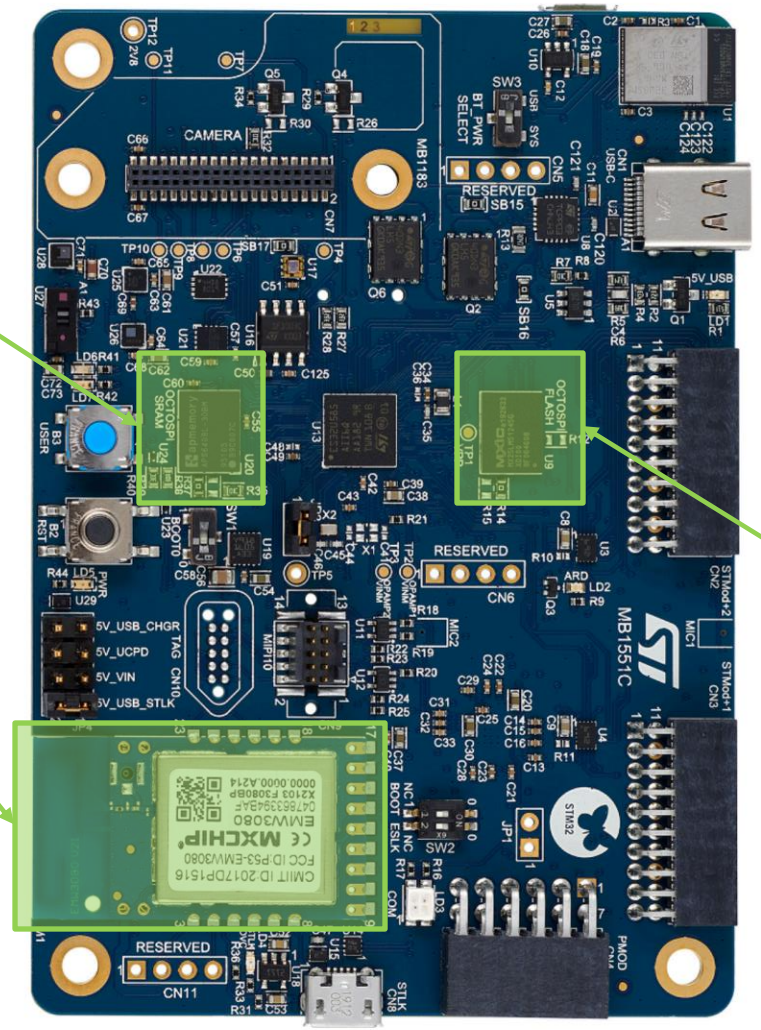
SPI – Connected Sensors

12-Mbit Octo-SPI Flash

- MX25LM51245GXDI005
- OCTOSPI2

Wifi Module

- EMW3080
- SPI2

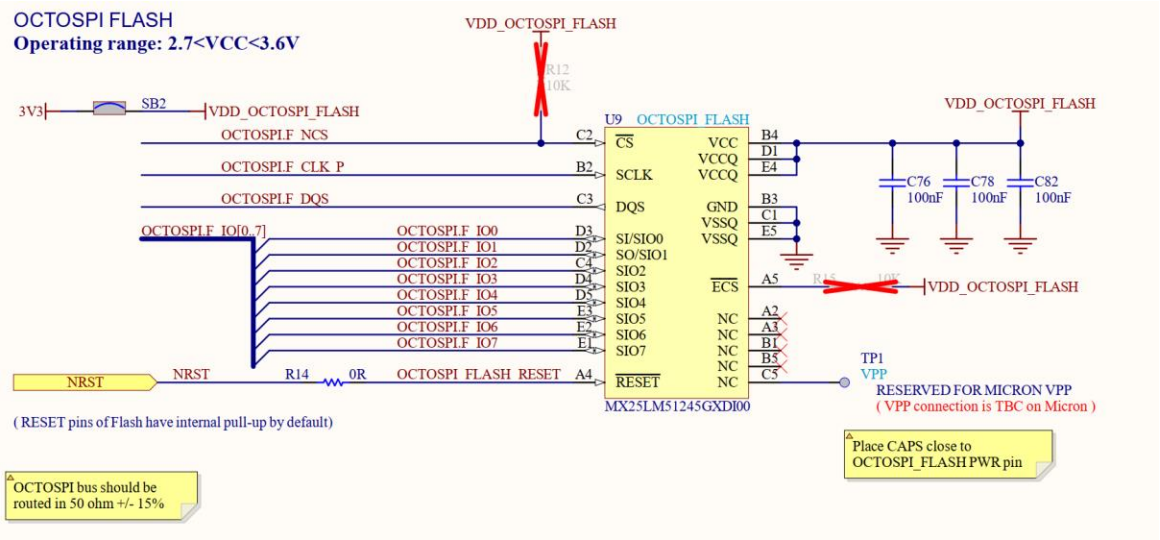


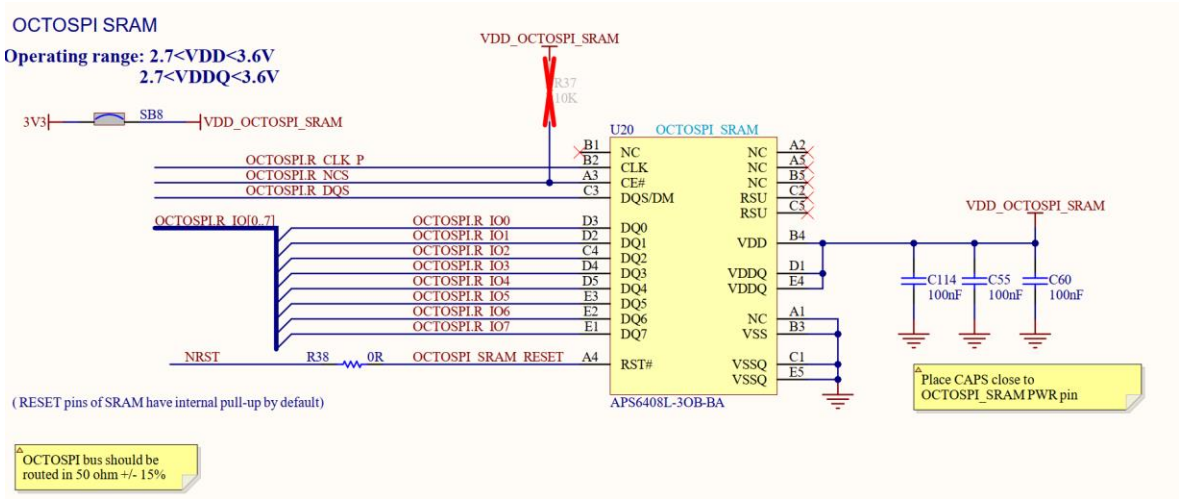
- SPI2 @ 3V3
- OCTOSPI1 @ 3V3
- OCTOSPI2 @ 3V3

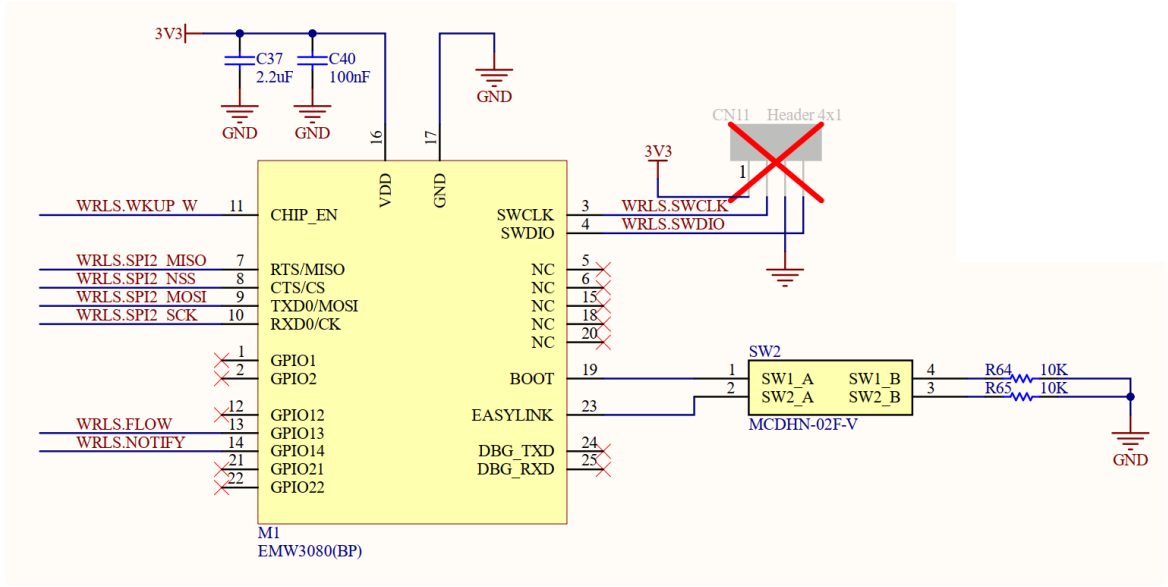
12-Mbit Octo-SPI RAM

- APS6408L-3OB-BA
- Octo-SPI

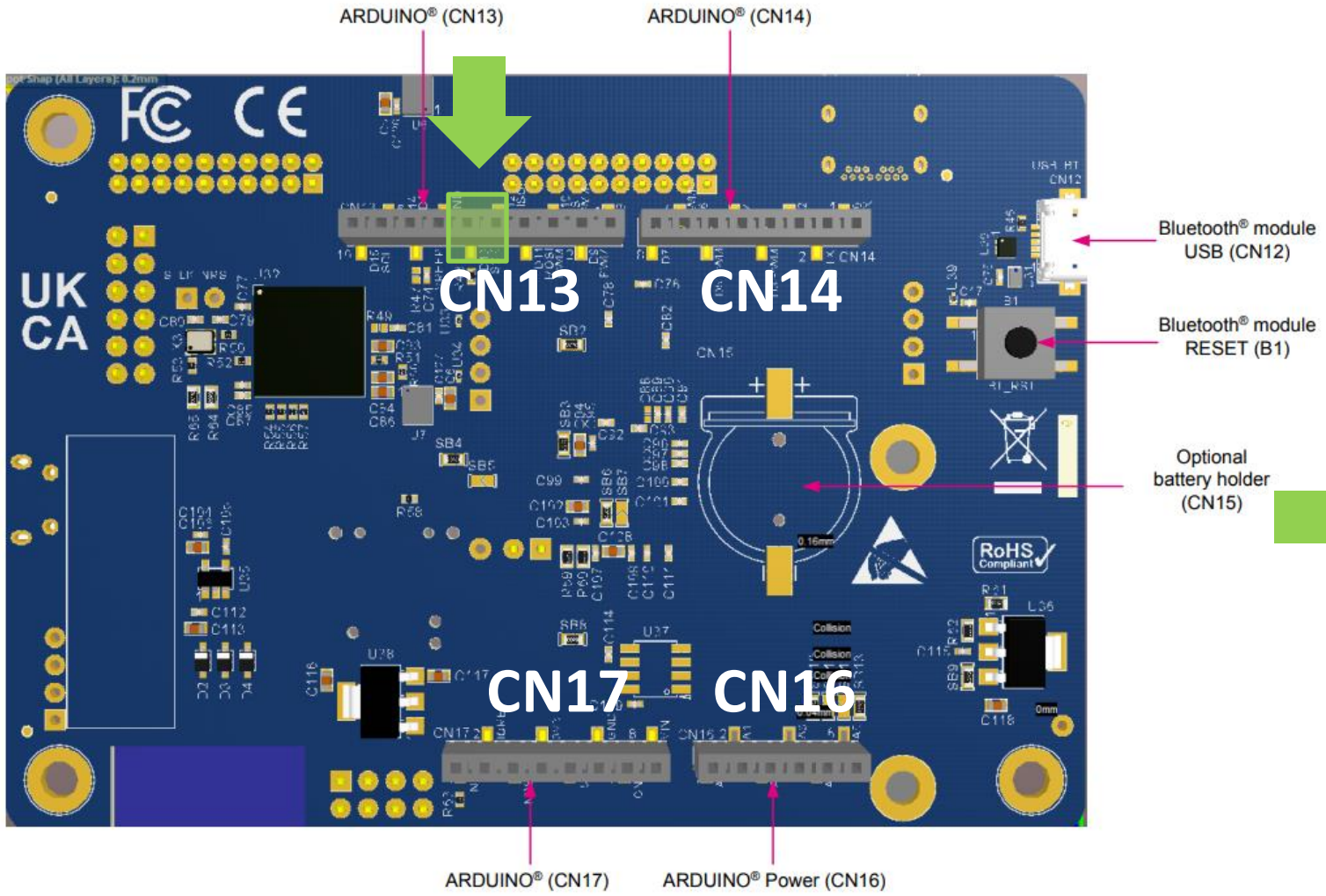
SPI – Connected Modules and Memory







SPI – External Connection



Connector	Pin number	Pin name	Signal name	STM32L 4+ pin	Function
CN17	1	NC	-	-	-
	2	IOREF	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	3V3	-	-	3.3 V I/O
	5	5V	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
	8	VIN	-	-	Power input
CN16	1	A0	ARD_ADC_A0	PC0	ADC
	2	A1	ARD-ADC_A1	PC2	ADC
	3	A2	ARD-ADC_A2	PC4	ADC
	4	A3	ARD-ADC_A3	PC5	ADC
	5	A4	ARD-ADC_A4	PA7	ADC / I2C1_SDA
	6	A5	ARD-ADC_A5	PB0	ADC / I2C1_SCL
CN13	10	SCL/D15	I2C1_SCL	PB8	I2C1_SCL
	9	SDA/D14	I2C1_SDA	PB9	I2C1_SDA
	8	VREFP	VREFP	-	VREFP
	7	GND	GND	-	GND
	6	SCK/D13	ARD.D13_SPI1_SCK	PE13	SPI1_SCK / LD2
	5	MISO/D12	ARD.D12_SPI1_MISO	PE14	SPI1_MISO
	4	PWM/MOSI/D11	ARD.D11_SPI1_MOSI	PE15	SPI1_MOSI / TIM1_CH4N
	3	PWM/CS/D10	ARD.D10_TIM_SPI1_NSS	PE12	SPI1_NSS / TIM1_CH3N
	2	PWM/D9	ARD.D9_TIM	PA8	TIM1_CH1
	1	D8	ARD.D8_IO	PC1	GPIO
CN14	8	D7	ARD.D7_IO	PF13	GPIO
	7	PWM/D6	ARD.D6_TIM	PB6	TIM4_CH1
	6	PWM/D5	ARD.D5_TIM	PE0	TIM16_CH1
	5	D4	ARD.D4_INT	PE7	GPIO
	4	PWM/D3	ARD.D3_TIM	PB2	TIM8_CH4N
	3	D2	ARD.D2_IO	PD15	GPIO
	2	TX/D1	ARD.D1_TX	PD8	UART3_TX
	1	RX/D0	ARD.D0_RX	PD9	UART3_RX

https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html

→ *Documents → UM2839, page 10*

SPI vs I²C

For **point-to-point**, SPI is simple and efficient

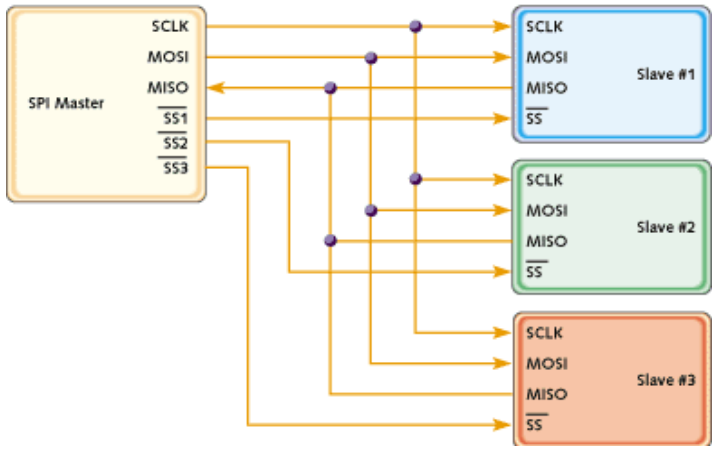
- Less overhead than I²C due to **lack of addressing**, plus SPI is **full-duplex**.

For multiple slaves, each slave needs **separate slave select** signal

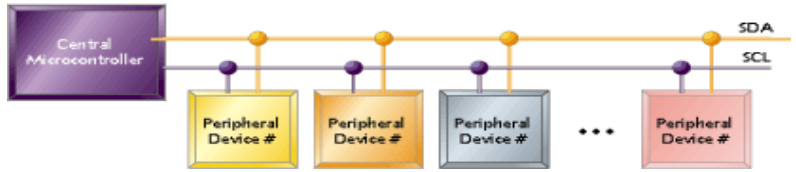
- SPI requires more effort and more hardware than I²C

Quad-SPI also exists

- 4x the bandwidth, often used by Flash drives



SPI



I²C

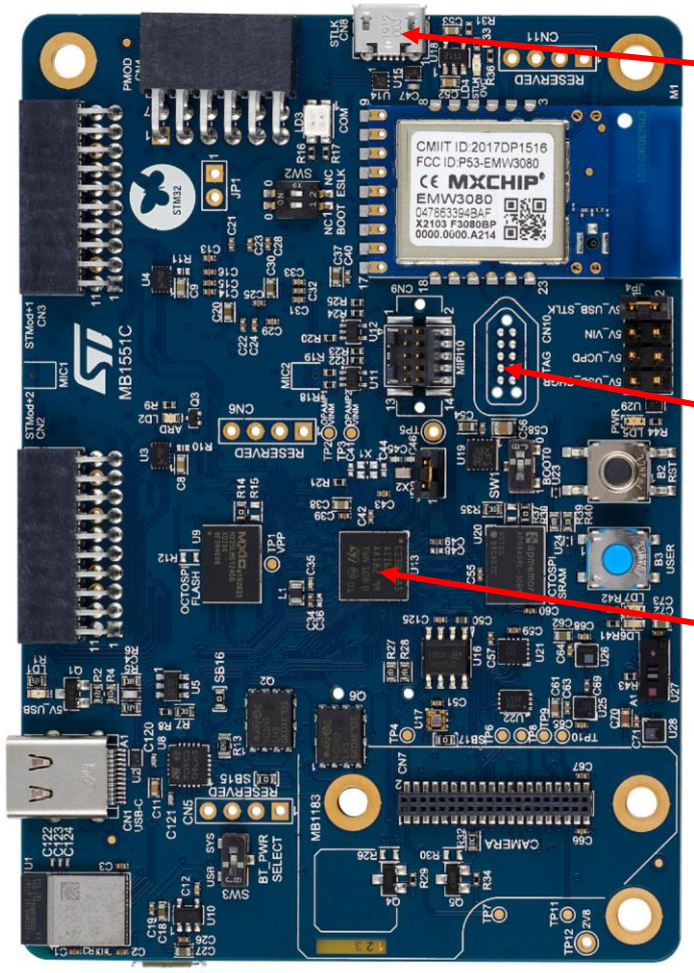
ETH zürich

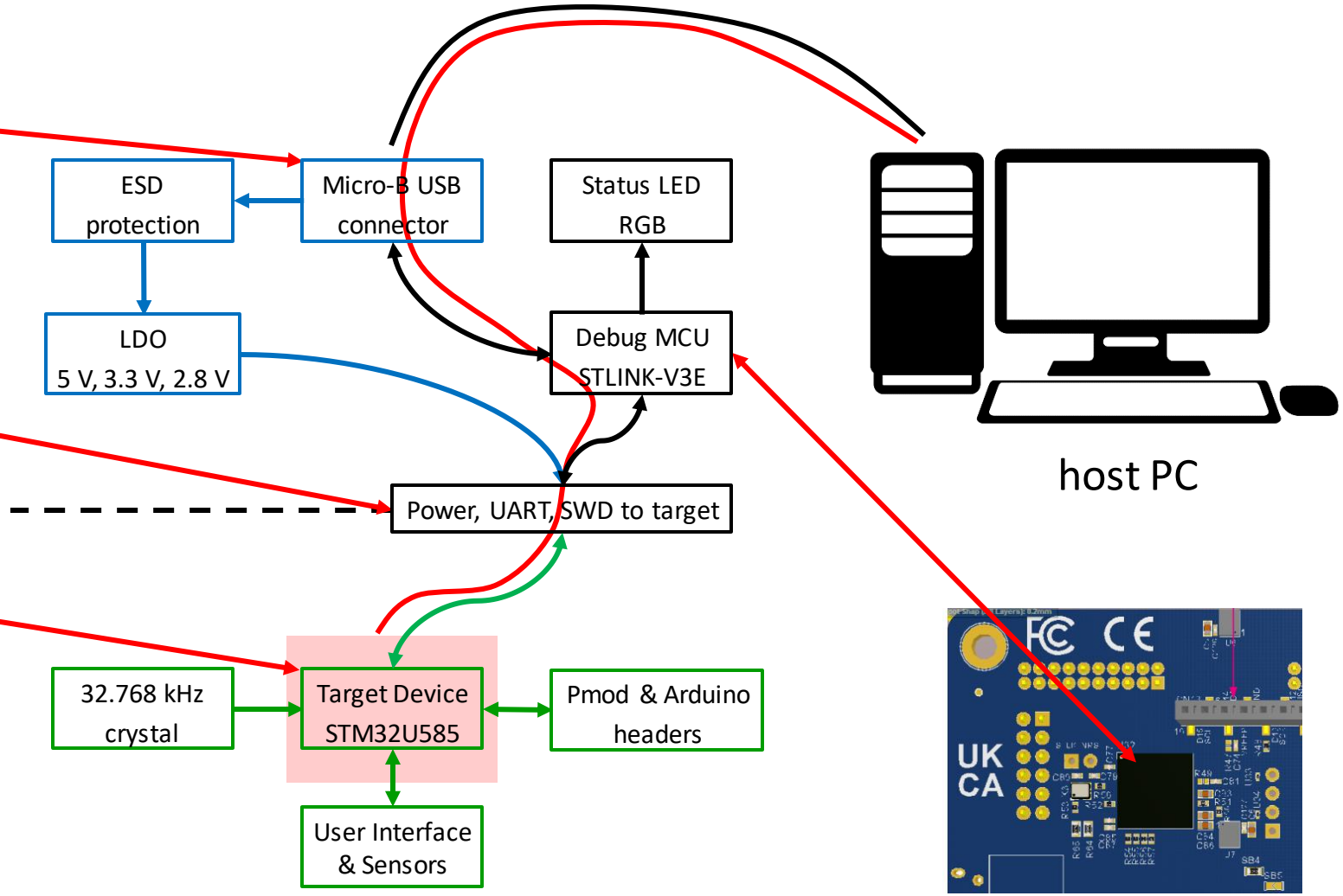
UART - Universal Asynchronous Receiver-Transmitter

UART - 1

- Stands for Universal Asynchronous Receiver-Transmitter
  - sometimes also found as USART (Universal Synchronous-Asynchronous Receiver Transmitter)
- Used to interface MCUs with other computing devices:
  - Communication with other processors, a PC (e.g. a serial terminal)
  - Used to interface the microcontroller with others transmission bus as: RS232, RS485, USB, CAN BUS, KNX, LonWorks ecc.
  - Used to connect MCUs with modems and transceivers as telephone modems, Bluetooth, Wi-Fi, GSM/GPRS/HDPSA

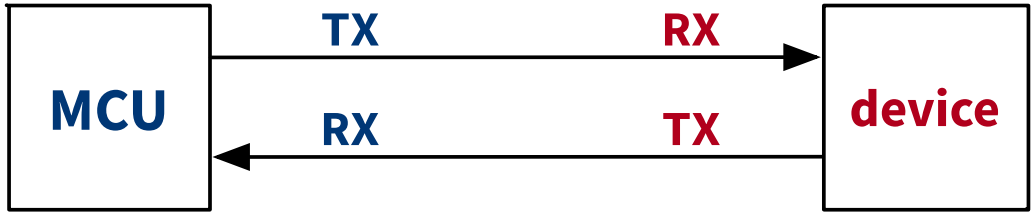
Software Development with STM32U585xx (ES-Lab)





UART - 2

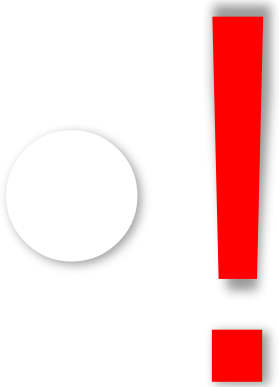
- Essentially a parallel2serial (TX), serial2parallel (RX) converter couple
  - e.g. using shift registers for P2S conversion
- Asynchronous: no common clock shared
  - Each device has its own local clock, typically running faster than the bit rate (e.g. 8x faster)
  - The phase of the receiver clock is locked onto the edge of the transmitted data



- Highly configurable
  - parity / no parity
  - data framing (e.g number of stop bits, number of payload bits)
  - simplex, full-duplex or half-duplex

UART - 2

- Essentially a parallel2serial (TX), serial2parallel (RX) converter couple
  - e.g. using shift registers for P2S conversion
- Asynchronous: no common clock shared
  - Each device has its own local clock, typically running faster than the bit rate (e.g. 8x faster)
  - The phase of the receiver clock is locked onto the edge of the transmitted data



- Highly configurable
  - parity / no parity
  - data framing (e.g number of stop bits, number of payload bits)
  - simplex, full-duplex or half-duplex

UART: “baud rate” vs “bit rate”

- UART communication speed is defined by its symbol rate measured in baud:
  - 1 baud = 1 **symbol per second**
  - in UART, a **symbol** has two values (0/1) -> **1 bit**
  - this number includes both data payload and protocol bits (e.g. parity, framing) – this number is also called “physical” or “gross” bit rate
- This can cause some confusion
  - Some people use “bit rate” for UART when referring only to payload bits
  - In some devices (e.g. modems) one symbol
  - might correspond to more bits -> baud rate is not the same as gross bit rate
  - Bottom line: to be 100% clear, **always talk of baud rate when referring to UART**, and remember that in UART **1 symbol = 1 bit**

Gross bit rate [edit]

In digital communication systems, the physical layer *gross bitrate*,[5] *raw bitrate*,[6] *data signaling rate*,[7] *gross data transfer rate*[8] or *uncoded transmission rate*[6] (sometimes written as a variable R_b[5][6] or f_b[9]) is the total number of physically transferred bits per second over a communication link, including useful data as well as protocol overhead.

In case of serial communications, the gross bit rate is related to the bit transmission time T_b as:

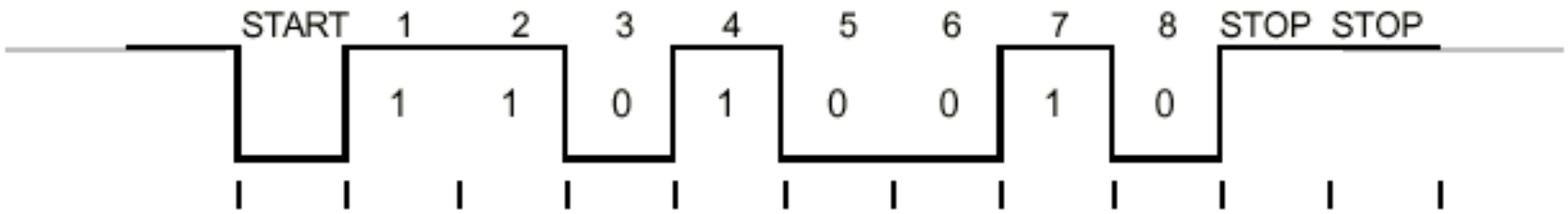
$$R_b = \frac{1}{T_b},$$

The gross bit rate is related to the symbol rate or modulation rate, which is expressed in bauds or symbols per second. However, the gross bit rate and the baud value are equal *only* when there are only two levels per symbol, representing 0 and 1, meaning that each symbol of a data transmission system carries exactly one bit of data; for example, this is not the case for modern modulation systems used in modems and LAN equipment.[10]

Ref. Wikipedia “Bit rate” page

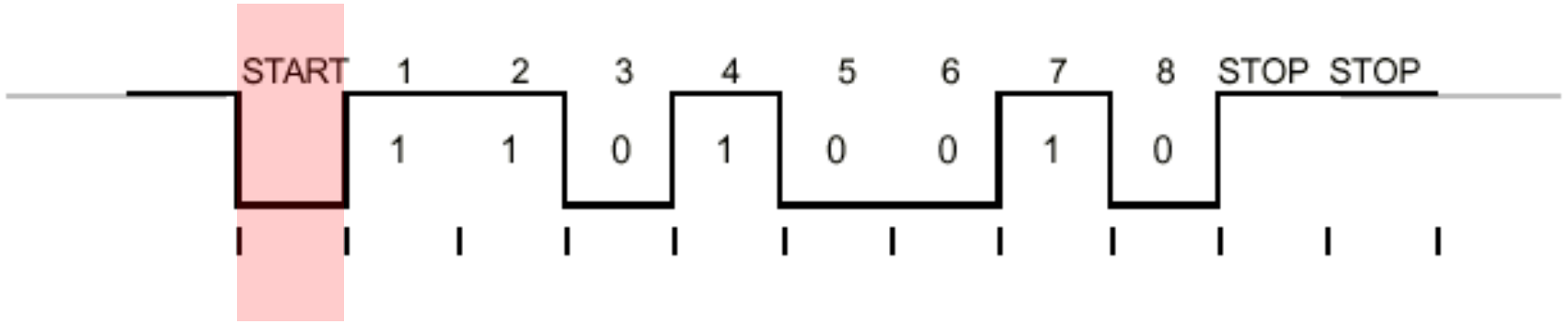
UART: Interface Protocol

- In idle, the transmission line is driven to 1



UART: Interface Protocol

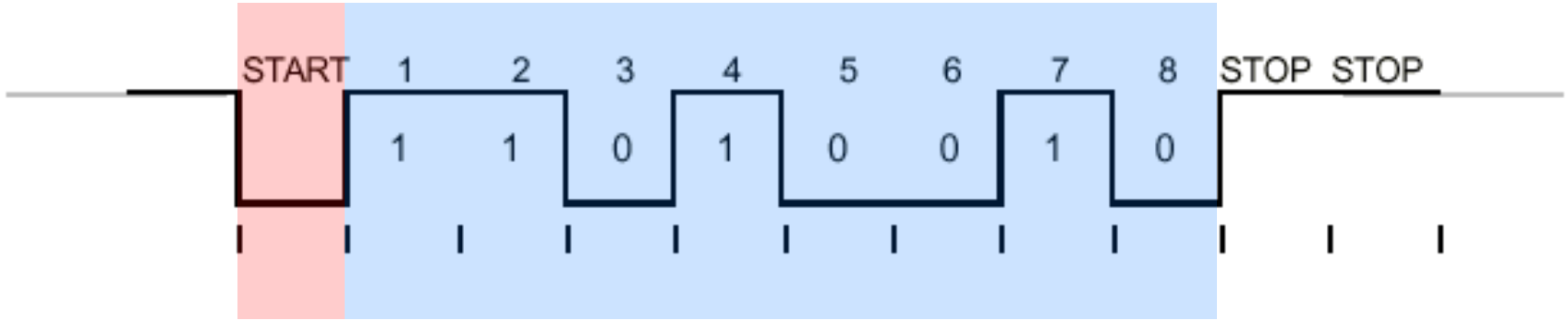
- The transfer begins with a start bit:
  - the transmission line is driven to 0



1 start bit

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

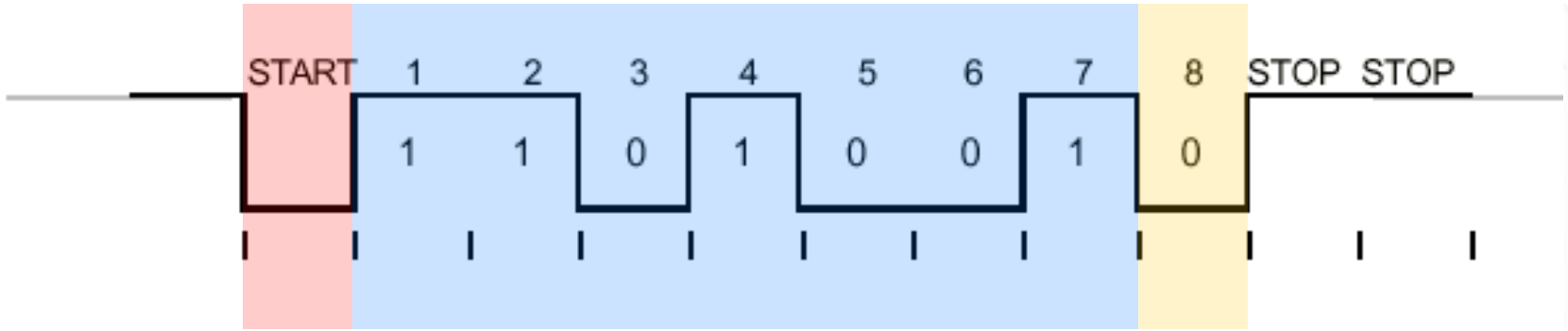


1 start bit

UART: Interface Protocol

- One of the data bits can be used for parity:
  - odd parity
  - even parity
  - in this case, 4-8 bits can be used for data

0 in the parity bit and there is an even number of 1's in the data bits. T

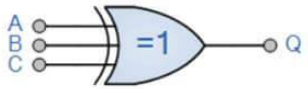


1 start bit **4-8 data bits** **1 parity bit**

$$p_{\text{odd}} = \underset{i=1}{\overset{N}{\mathbf{XOR}}}\ b_i$$

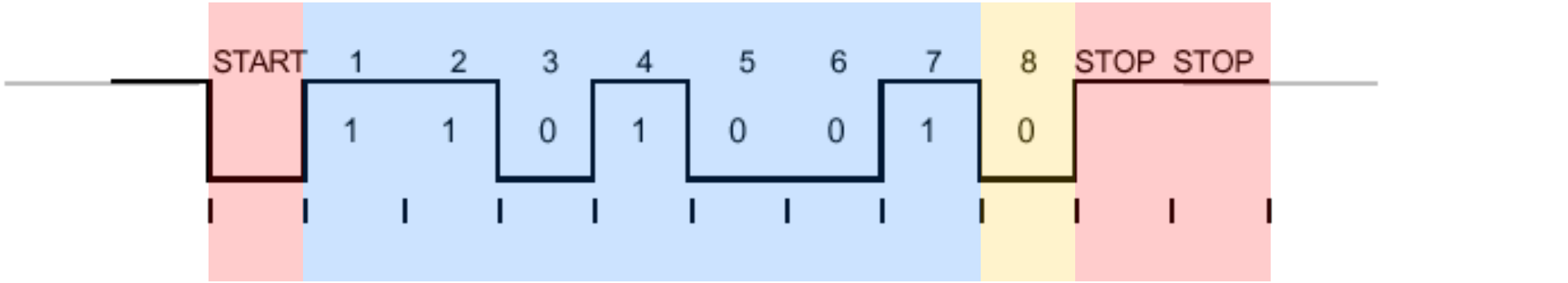
$$p_{\text{even}} = \underset{i=1}{\overset{N}{\mathbf{XNOR}}}\ b_i$$

Even parity where a 1-parity-bit is sent if there are an even number of 1-bits

Symbol	Truth Table			
 3-input Ex-OR Gate	C	B	A	Q
	0	0	0	0
	0	0	1	1
	0	1	0	1
	0	1	1	0
	1	0	0	1
	1	0	1	0
	1	1	0	0
	1	1	1	1
Boolean Expression Q = A ⊕ B ⊕ C	"Any **ODD** Number of Inputs" gives Q			

UART: Interface Protocol

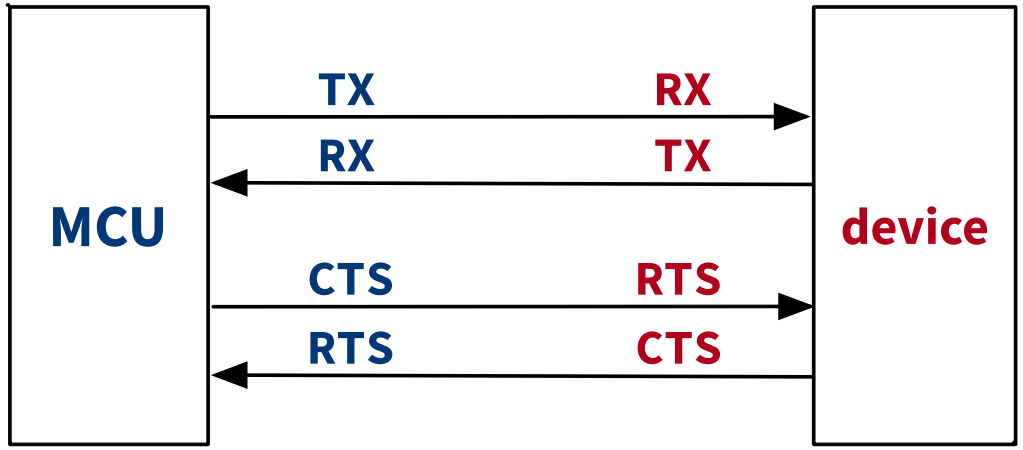
- Finally, 1-2 stop bits:
  - Transmission line brought back to 1
  - 1 or 2 stop bits depending on application



1 start bit **4-8 data bits** **1 parity bit** **1-2 stop bits**

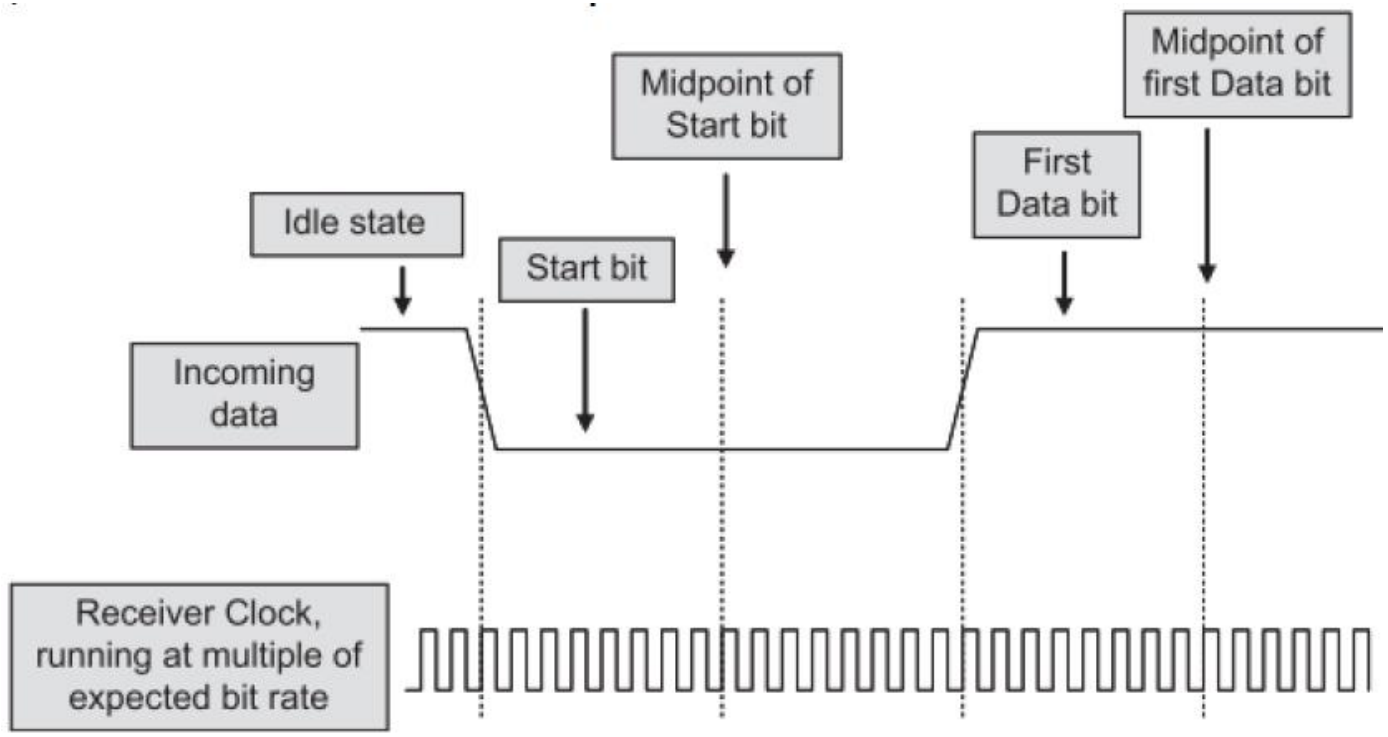
UART: Handshake

- The UART protocol can also include a handshake:
- request-to-send (RTS) signal from the MCU to the device means that the MCU can accept new data
- clear-to-send (CTS) signal from the device to the MCU means that the device can send new data
- signals have dual meaning if seen from the other point of view
- exchange happens when CTS and RTS are both asserted



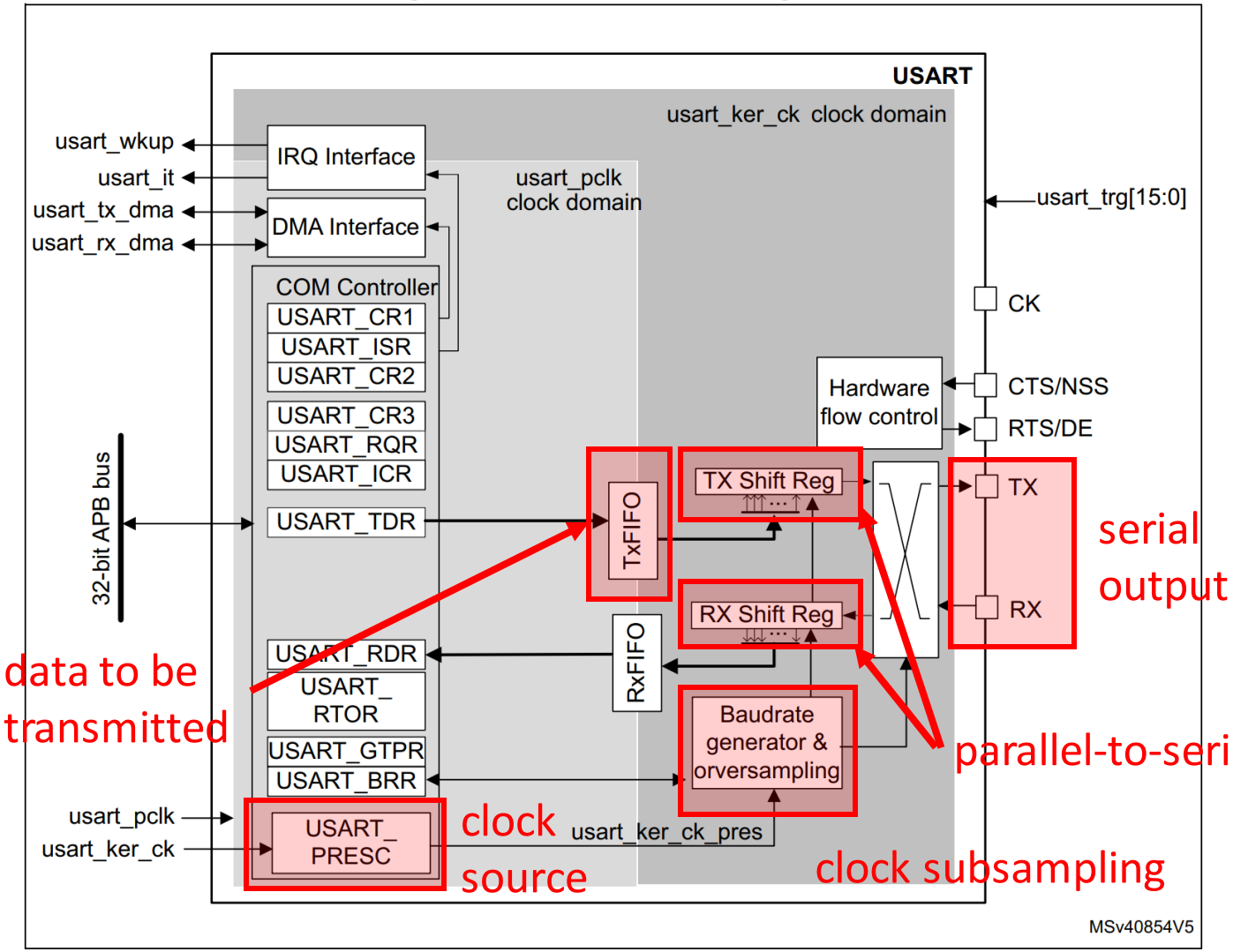
UART

- The receiver runs an *internal clock* whose frequency is an exact multiple of the expected bit rate.
- When a *Start bit* is detected, a counter begins to count clock cycles e.g. 8 cycles until the midpoint of the anticipated Start bit is reached.
- The clock counter counts a further 16 cycles, to the middle of the first *Data bit*, and so on until the *Stop bit*.



Transmission Rate

Figure 656. USART block diagram



Clock subsampling:

- The clock subsampling block is complex, as one tries to match a large set of transmission rates with a fixed input frequency.

Clock Source:

- USART_PCLK in the lab is then dived

to have the baud rate generator

- ***Example:***
- Will be done in the LAB!

https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html

→ *Reference Manual RM0456, page 2258*

Memory-Mapped Device Access

Bus	Secure boundary address	Non-secure boundary address	Size (bytes)	Peripheral	Peripheral register map
	0x5000 5000 - 0x5000 53FF	0x4000 5000 - 0x4000 53FF	1 K	UART5	
	0x5000 4C00 - 0x5000 4FFF	0x4000 4C00 - 0x4000 4FFF	1 K	UART4	*USART register map*
	0x5000 4800 - 0x5000 4BFF	0x4000 4800 - 0x4000 4BFF	1 K	USART3	
	0x5000 4400 - 0x5000 47FF	0x4000 4400 - 0x4000 47FF	1 K	USART2	

57.8.12 USART receive data register (USART_RDR)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	RDR[8:0]								
							r	r	r	r	r	r	r	r	r

Bits 31:9 Reserved, must be kept at reset value.

Bits 8:0 **RDR[8:0]**: Receive data value

57.8.13 USART transmit data register (USART_TDR)

Address offset: 0x28

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	TDR[8:0]								
							rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:9 Reserved, must be kept at reset value.

Bits 8:0 **TDR[8:0]**: Transmit data value

- *Configuration of Transmitter and Receiver must match*; otherwise, they can not communicate.
- Examples of configuration parameters:
  - transmission rate (baud rate, i.e., symbols/s)
  - LSB or MSB first
  - number of bits per packet
  - parity bit
  - number of stop bits
  - interrupt-based communication
  - clock source

in our case: bit/s

buffer for received bits and bits that should be transmitted

https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html

→ *Reference Manual RM0456, page 2337*

Software Interface

Part of C program that *prints one byte to a UART* terminal on the host PC:

```
void uart_init()
{
  __USART2_CLK_ENABLE();          // Enable clocksource for USART2

  /* Configure USART2 */
  huart2.Instance =               USART2;
  huart2.Init.BaudRate =          115200;
  huart2.Init.WordLength =        UART_WORDLENGTH_8B;
  huart2.Init.StopBits =          UART_STOPBITS_1;
  huart2.Init.Parity =            UART_PARITY_NONE;
  huart2.Init.Mode =              UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl =         UART_HWCONTROL_NONE;
  huart2.Init.OverSampling =      UART_OVERSAMPLING_16;

  HAL_UART_Init(&huart2);         // Init USART2 peripheral
}

int main(void)
{
  HAL_Init();             // Initialize HAL

  uart_gpio_init();       // Init UART GPIO (RX and TX pins)
  uart_init();            // Init UART peripheral

  /* Transmit one byte with 100 ms timeout */
  HAL_UART_Transmit(&huart2, &byte, 1, 100)

  /* Code */
}
```

data structure uartConfig contains the configuration of the UART

USART2 defines the start address for the USART2 memory block:

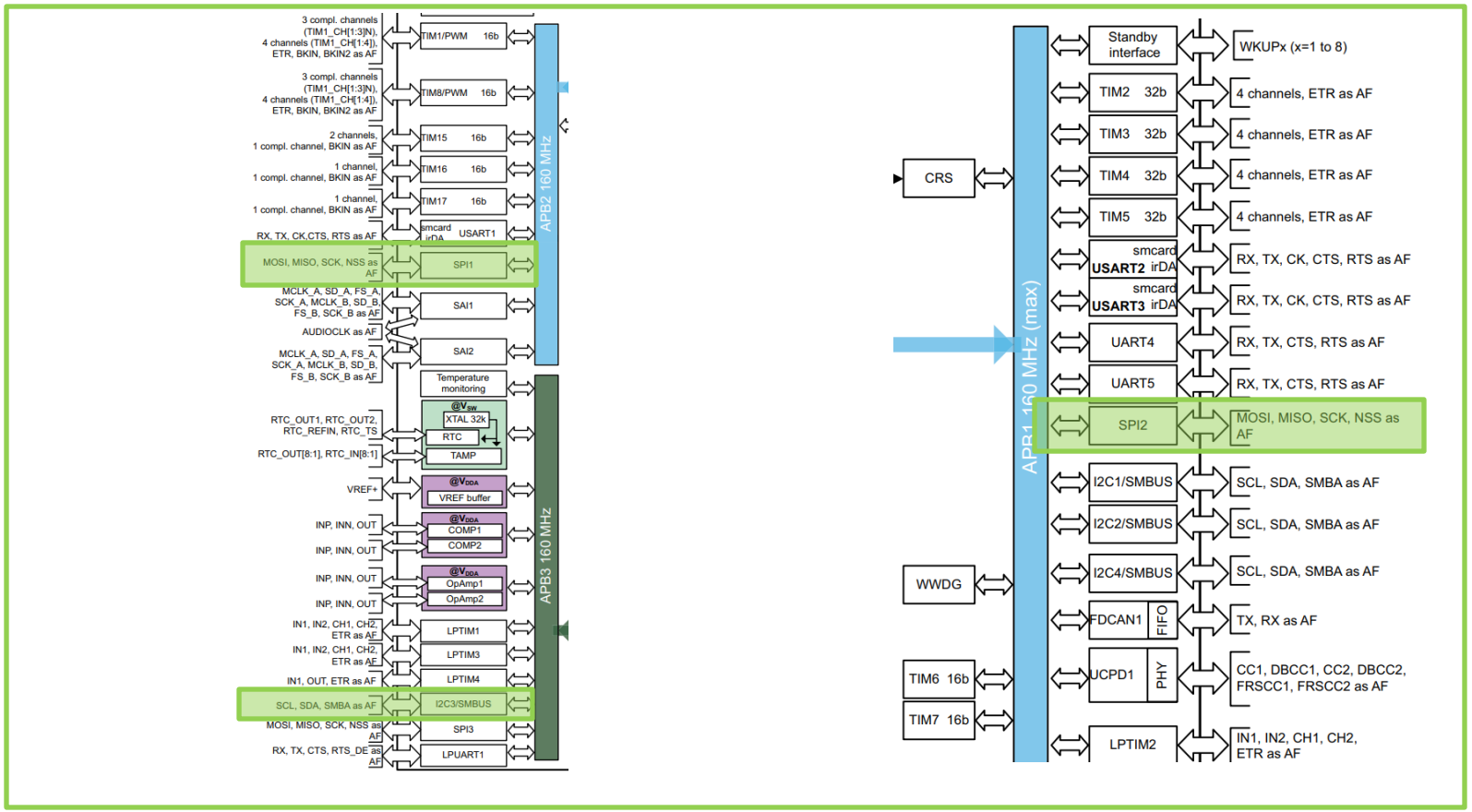
0x4000_4400 (for non-secure)

start UART

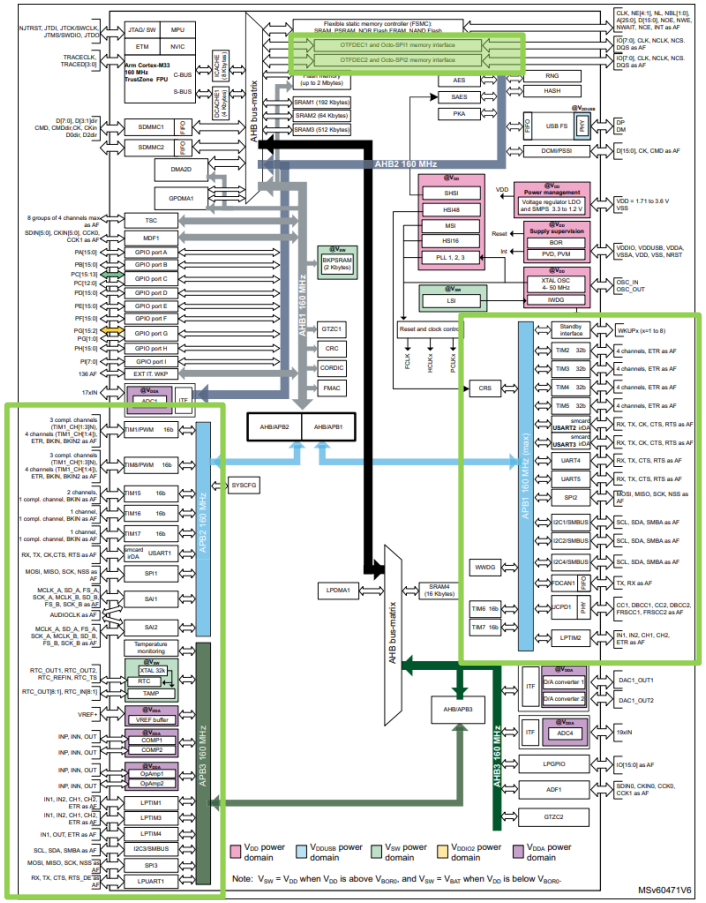
Use HAL_UART_Init to write the configuration registers of the instance USART2

SPI – STM32U585xx

- 3 x SPI
- 1 x SPI with dual OCTOSPI



STM32U585xx block diagram

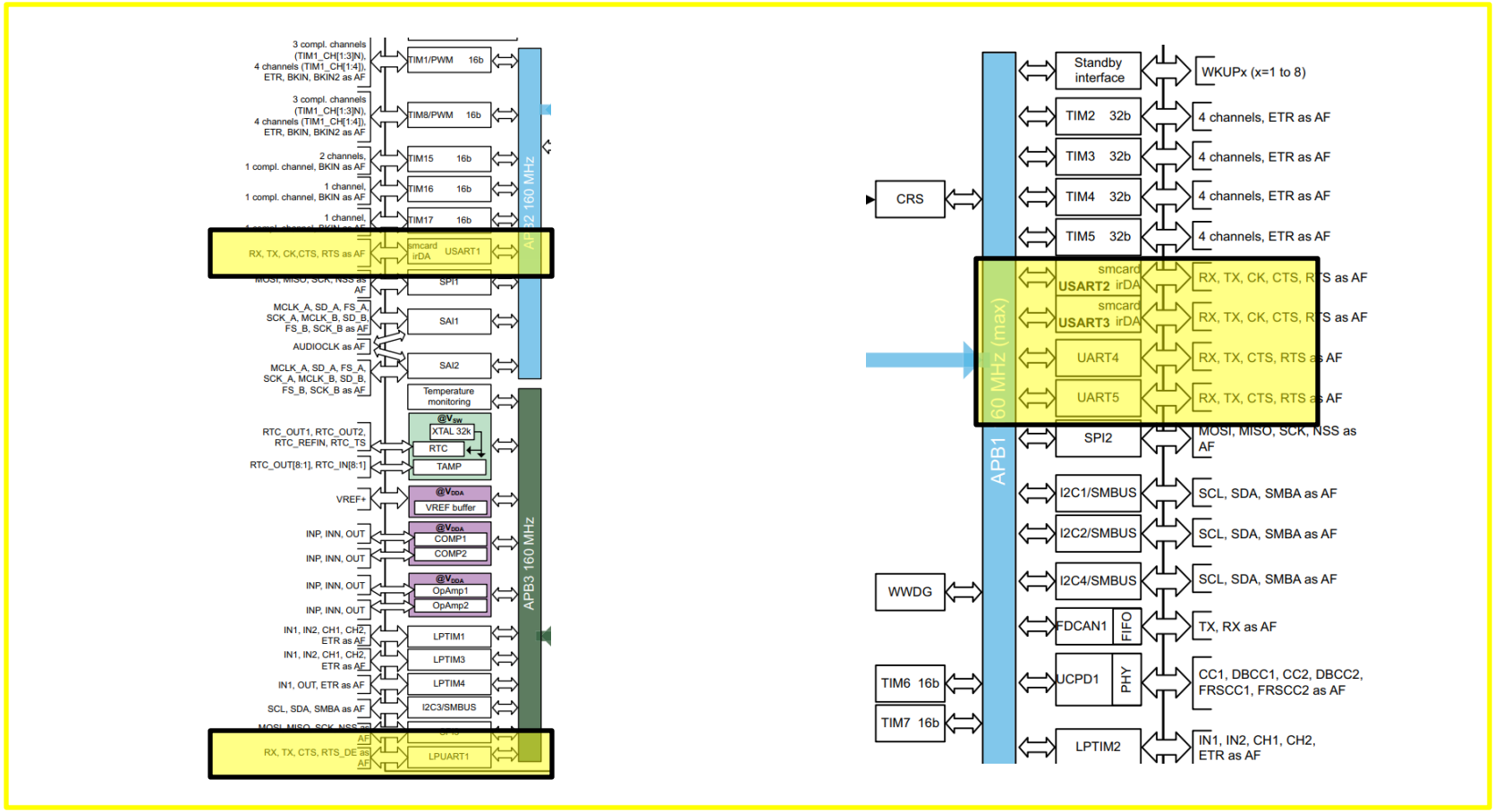


https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html

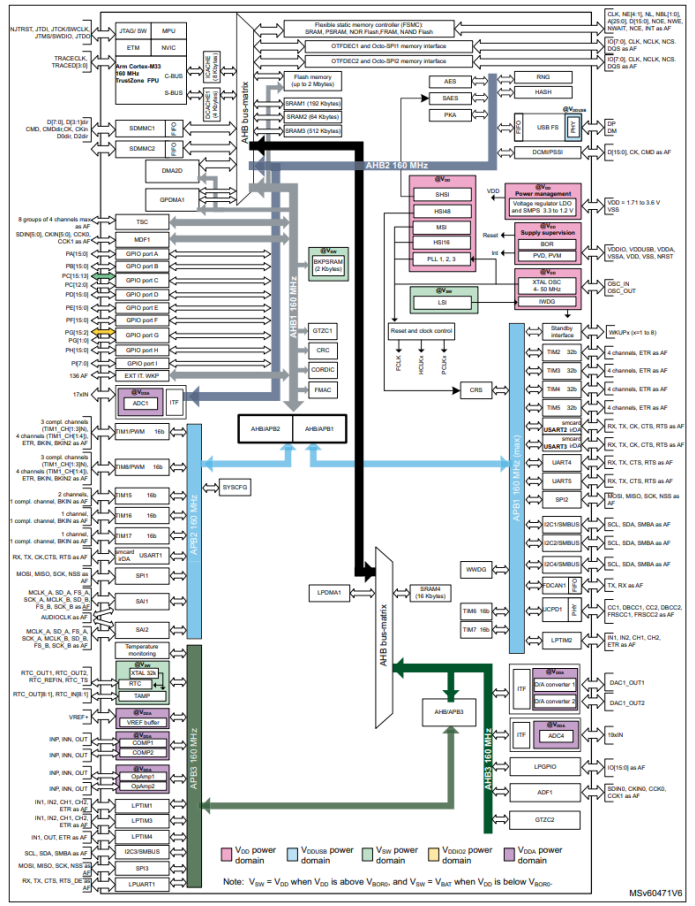
→ *Datasheet, page 21*

UART – STM32U5

- 3x USART 2x UART and 1x LPUART



STM32U585xx block diagram



https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html

→ *Datasheet, page 21*

UART – STM32U585xx

Table 22. USART, UART and LPUART features

USART modes/features(1)	USART1/2/3	UART4/5	LPUART1
Hardware flow control for modem	X	X	X
Continuous communication using DMA	X	X	X
Multiprocessor communication	X	X	X
Synchronous mode (master/slave)	X	-	-
Smartcard mode	X	-	-
Single-wire half-duplex communication	X	X	X
IrDA SIR ENDEC block	X	X	-
LIN mode	X	X	-
Dual-clock domain and wakeup from Stop mode	X(2)	X(2)	X(3)
Receiver timeout interrupt	X	X	-
Modbus communication	X	X	-
Auto-baud rate detection	X	X	-
Driver enable	X	X	X
USART data length	7, 8 and 9 bits		
Tx/Rx FIFO	X	X	X
Tx/Rx FIFO size	8 bytes		
Autonomous mode	X	X	X

1. X = supported.
2. Wakeup supported from Stop 0 and Stop 1 modes.
3. Wakeup supported from Stop 0, Stop 1 and Stop 2 modes.

Table 72. Typical dynamic current consumption of peripherals (continued)

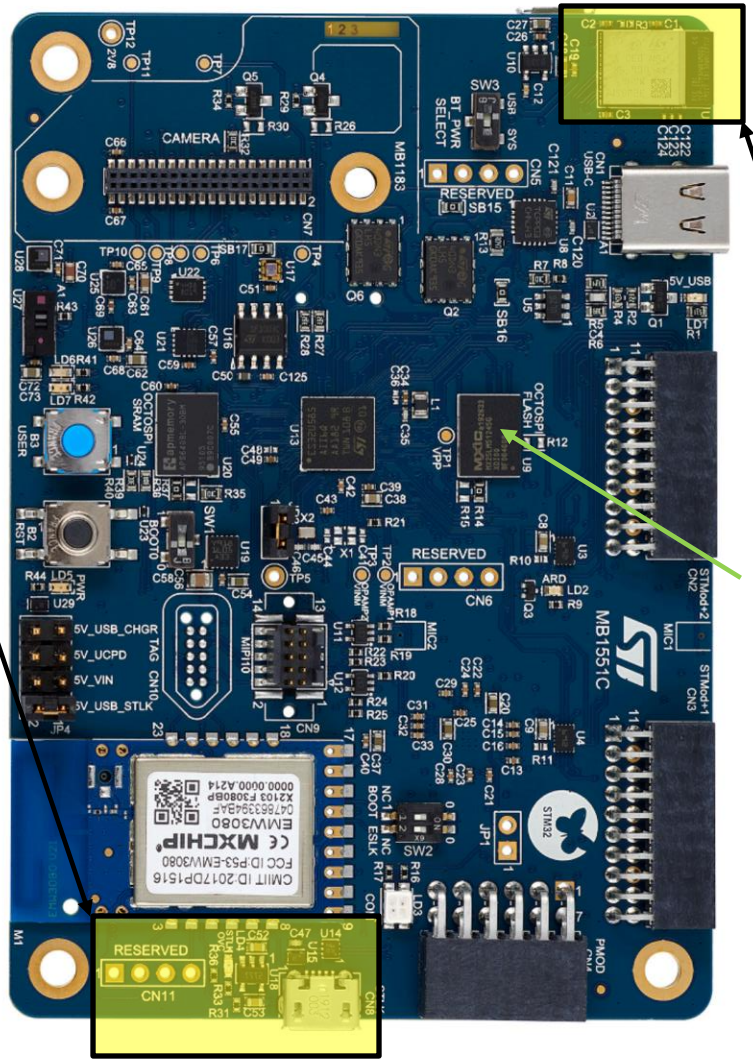
Bus	Peripheral	LDO					SMPS					Unit
		Range1	Range2	Range3	Range4	Stop 1 Stop 2	Range1	Range2	Range3	Range4	Stop 1 Stop 2	
	USART1	2.38	2.16	1.96	1.76	-	1.14	0.97	0.81	0.62	-	µA/ MHz
	USART1 indep(1)	4.48	4.09	3.71	3.35	-	2.17	1.84	1.54	1.19	-	
	LPUART1	1.18	1.07	0.97	0.87	0.88	0.57	0.48	0.41	0.31	0.31	
	LPUART1 indep(1)	1.96	1.78	1.62	1.45	1.46	0.95	0.80	0.67	0.52	0.52	
	UART4	1.86	1.70	1.54	1.39	-	0.90	0.76	0.64	0.50	-	
	UART4 indep(1)	3.47	3.17	2.87	2.60	-	1.68	1.42	1.19	0.93	-	
	UART5	1.93	1.76	1.60	1.44	-	0.94	0.79	0.66	0.51	-	
	UART5 indep(1)	3.57	3.25	2.95	2.67	-	1.72	1.46	1.23	0.95	-	
	UCPD1	1.60	1.46	1.33	1.17	-	0.78	0.66	0.55	0.43	-	
	USART2	5.53	5.04	4.57	4.12	-	2.67	2.26	1.91	1.46	-	
	USART2 indep(1)	3.57	3.24	2.95	2.65	-	1.72	1.46	1.22	0.94	-	
	USART3	2.10	1.91	1.73	1.57	-	1.02	0.86	0.72	0.56	-	
	USART3 indep(1)	4.24	3.86	3.5	3.17	-	2.05	1.73	1.45	1.12	-	

https://www.st.com/en/microcontrollers-microprocessors/stm32u585ai.html

→ *Datasheet, page 84*

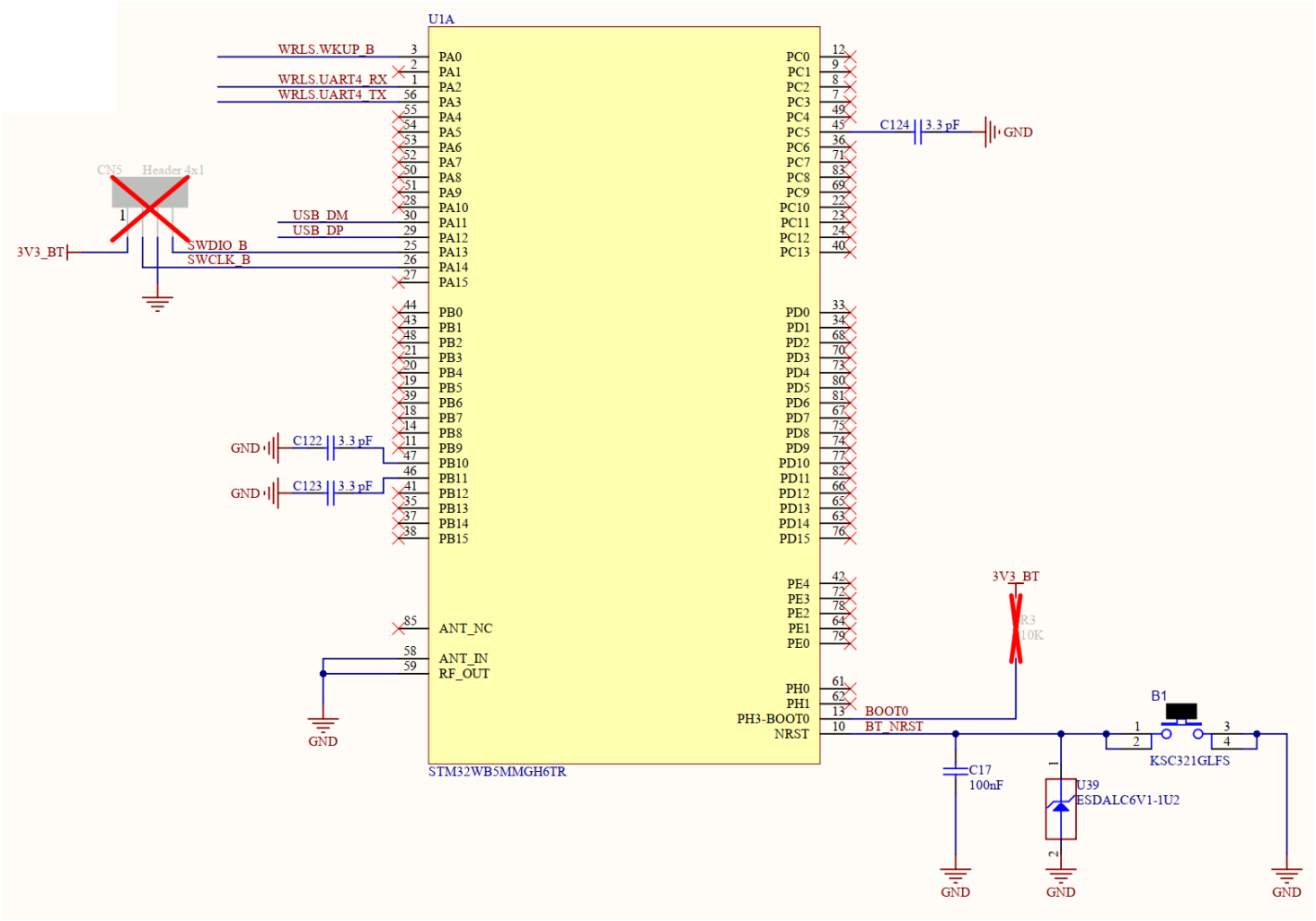
UART – Connected Modules

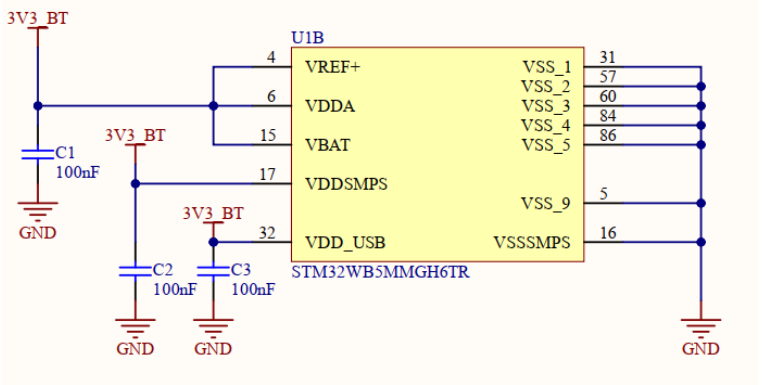
- DEBUGING CONNECTION
- ST-LINK
- UART1



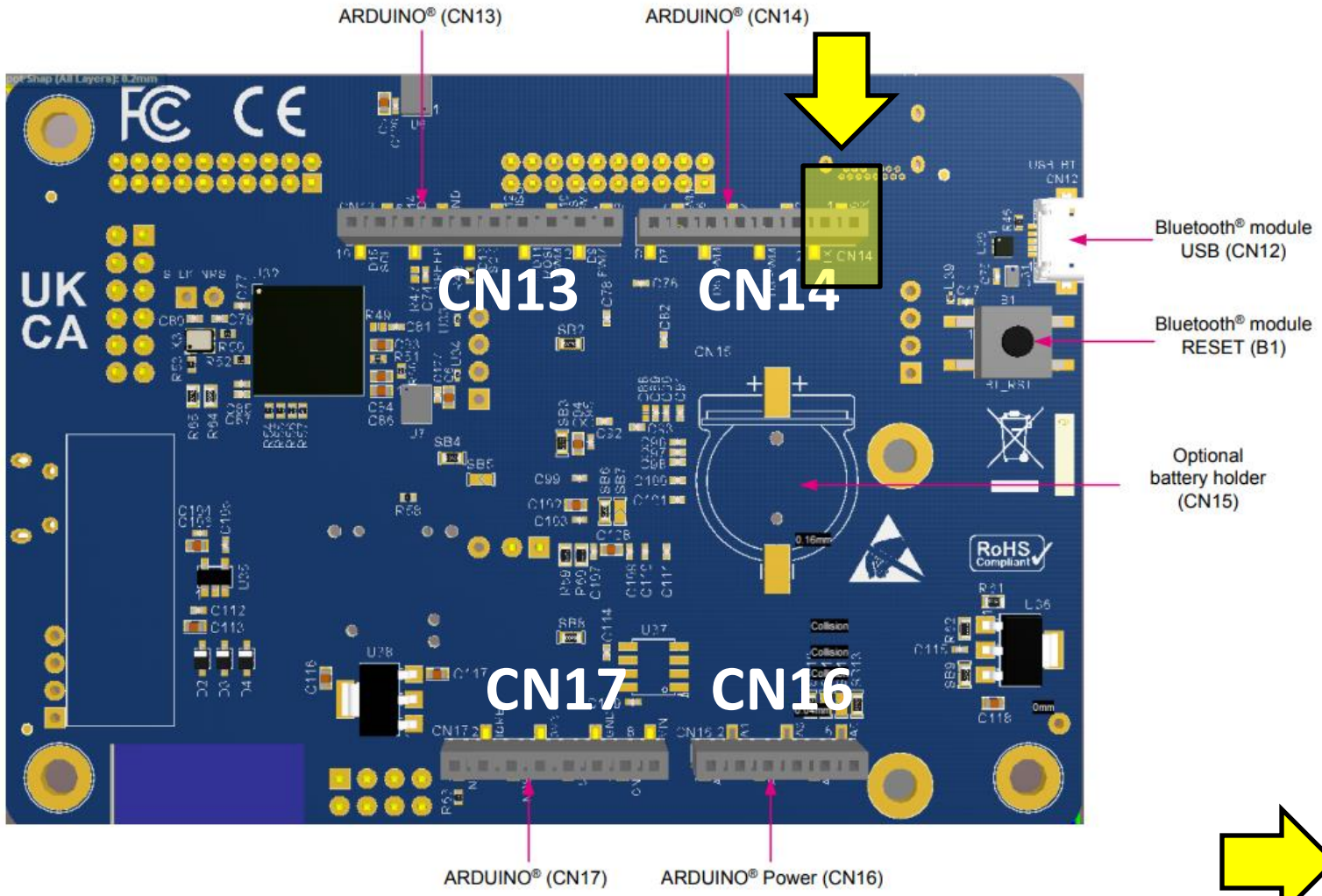
- Bluetooth Module
- STM32WB5MMGH6TR
- UART4

UART – Connected Module





UART – External Connection



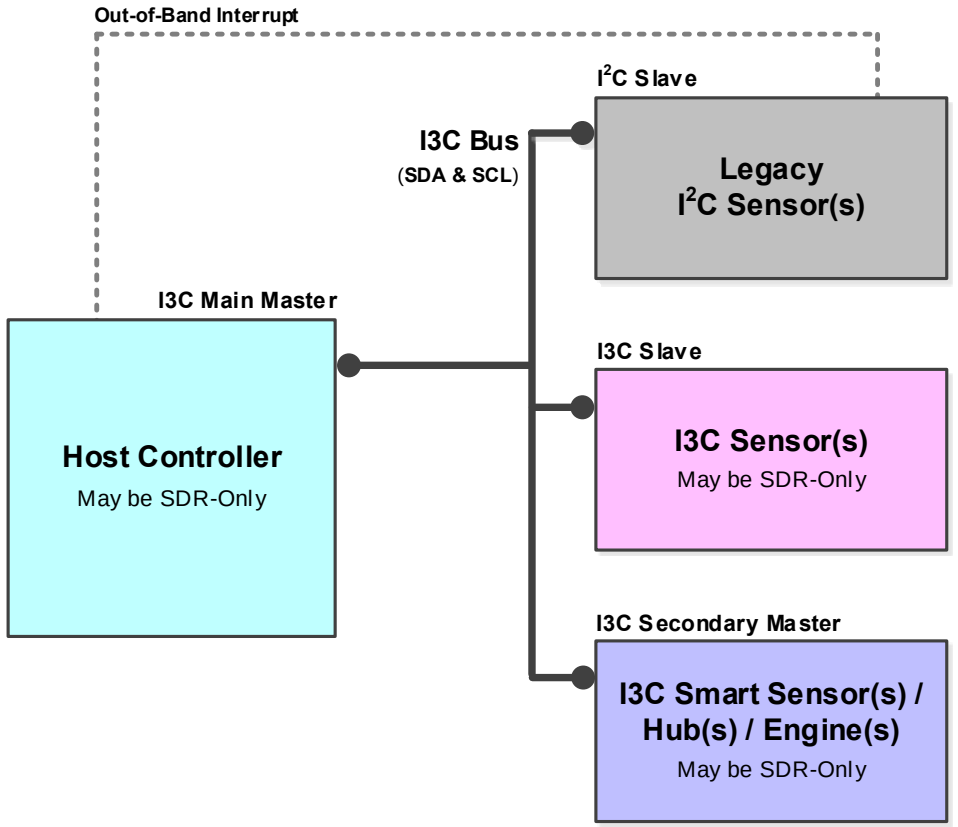
Connector	Pin number	Pin name	Signal name	STM32L 4+ pin	Function
CN17	1	NC	-	-	-
	2	IOREF	-	-	3.3 V reference
	3	NRST	NRST	NRST	Reset
	4	3V3	-	-	3.3 V I/O
	5	5V	-	-	5 V
	6	GND	-	-	GND
	7	GND	-	-	GND
	8	VIN	-	-	Power input
CN16	1	A0	ARD_ADC_A0	PC0	ADC
	2	A1	ARD-ADC_A1	PC2	ADC
	3	A2	ARD-ADC_A2	PC4	ADC
	4	A3	ARD-ADC_A3	PC5	ADC
	5	A4	ARD-ADC_A4	PA7	ADC / I2C1_SDA
	6	A5	ARD-ADC_A5	PB0	ADC / I2C1_SCL
CN13	10	SCL/D15	I2C1_SCL	PB8	I2C1_SCL
	9	SDA/D14	I2C1_SDA	PB9	I2C1_SDA
	8	VREFP	VREFP	-	VREFP
	7	GND	GND	-	GND
	6	SCK/D13	ARD.D13_SPI1_SCK	PE13	SPI1_SCK / LD2
	5	MISO/D12	ARD.D12_SPI1_MISO	PE14	SPI1_MISO
	4	PWM/MOSI/D11	ARD.D11_SPI1_MOSI	PE15	SPI1_MOSI / TIM1_CH4N
	3	PWM/CS/D10	ARD.D10_TIM_SPI1_NSS	PE12	SPI1_NSS / TIM1_CH3N
	2	PWM/D9	ARD.D9_TIM	PA8	TIM1_CH1
	1	D8	ARD.D8_IO	PC1	GPIO
CN14	8	D7	ARD.D7_IO	PF13	GPIO
	7	PWM/D6	ARD.D6_TIM	PB6	TIM4_CH1
	6	PWM/D5	ARD.D5_TIM	PE0	TIM16_CH1
	5	D4	ARD.D4_INT	PE7	GPIO
	4	PWM/D3	ARD.D3_TIM	PB2	TIM8_CH4N
	3	D2	ARD.D2_IO	PD15	GPIO
	2	TX/D1	ARD.D1_TX	PD8	UART3_TX
	1	RX/D0	ARD.D0_RX	PD9	UART3_RX

https://www.st.com/en/evaluation-tools/b-u585i-iot02a.html

→ *Documents → UM2839, page 10*

The next-generation: I3C example

- An evolution of I2C proposed by the MIPI alliance (2016/7)
- Designed to fit applications currently using I2C, SPI, UART
- Many operating modes, I2C backward-compatibility also supported
  - without some of the most «exotic» features such as SCL stretching
  - supporting also (mainly) push-pull drivers
- Targets high data rate and energy efficiency



What did you Learn?

- Interrupts are crucial for
  - Work in parallel
  - Duty cycling
  - Energy Efficiency

- External world Interface
  - Serial ports
  - SPI
  - I2C
  - UART

- This is today used in real embedded systems