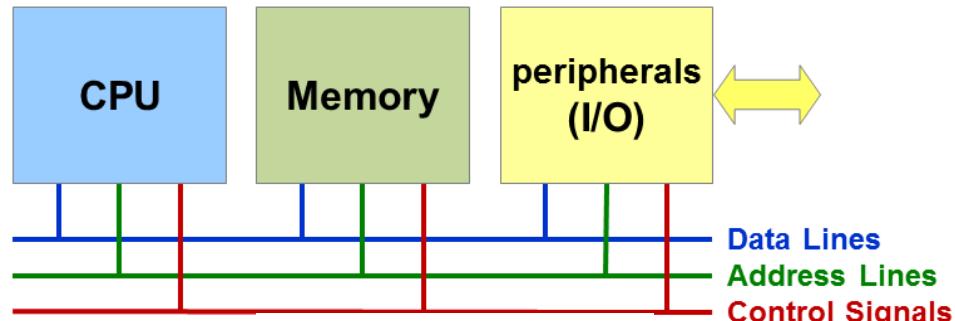


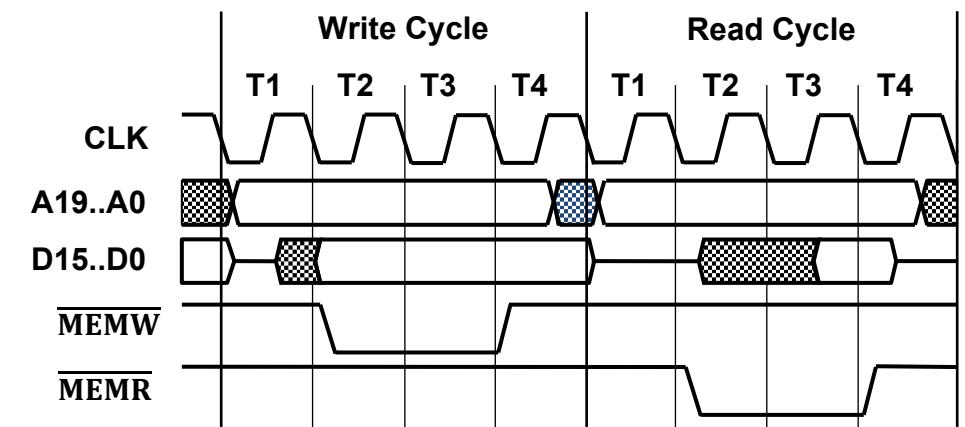
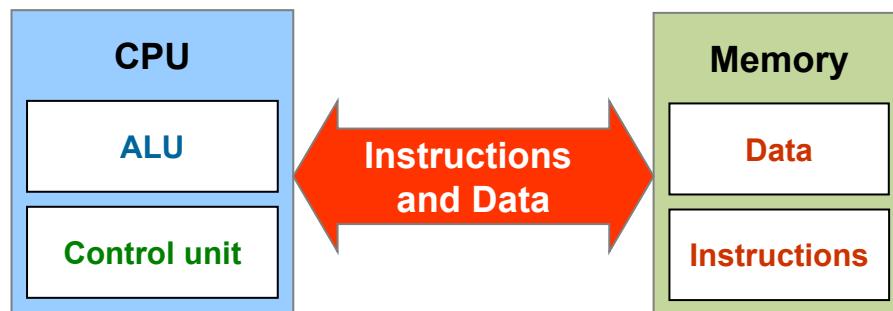
Microcontroller Basics

Computer Engineering 2

■ Connecting a CPU to the Outside World



von Neumann Architecture



- **Microcontroller**
- **System Bus**
- **Digital Logic Basics**
- **Synchronous Bus**
- **Control and Status Registers**
- **Address Decoding**
- **Slow Slaves (Peripherals)**
- **Bus Hierarchies**
- **Accessing Control Registers in C**
- **Conclusions**

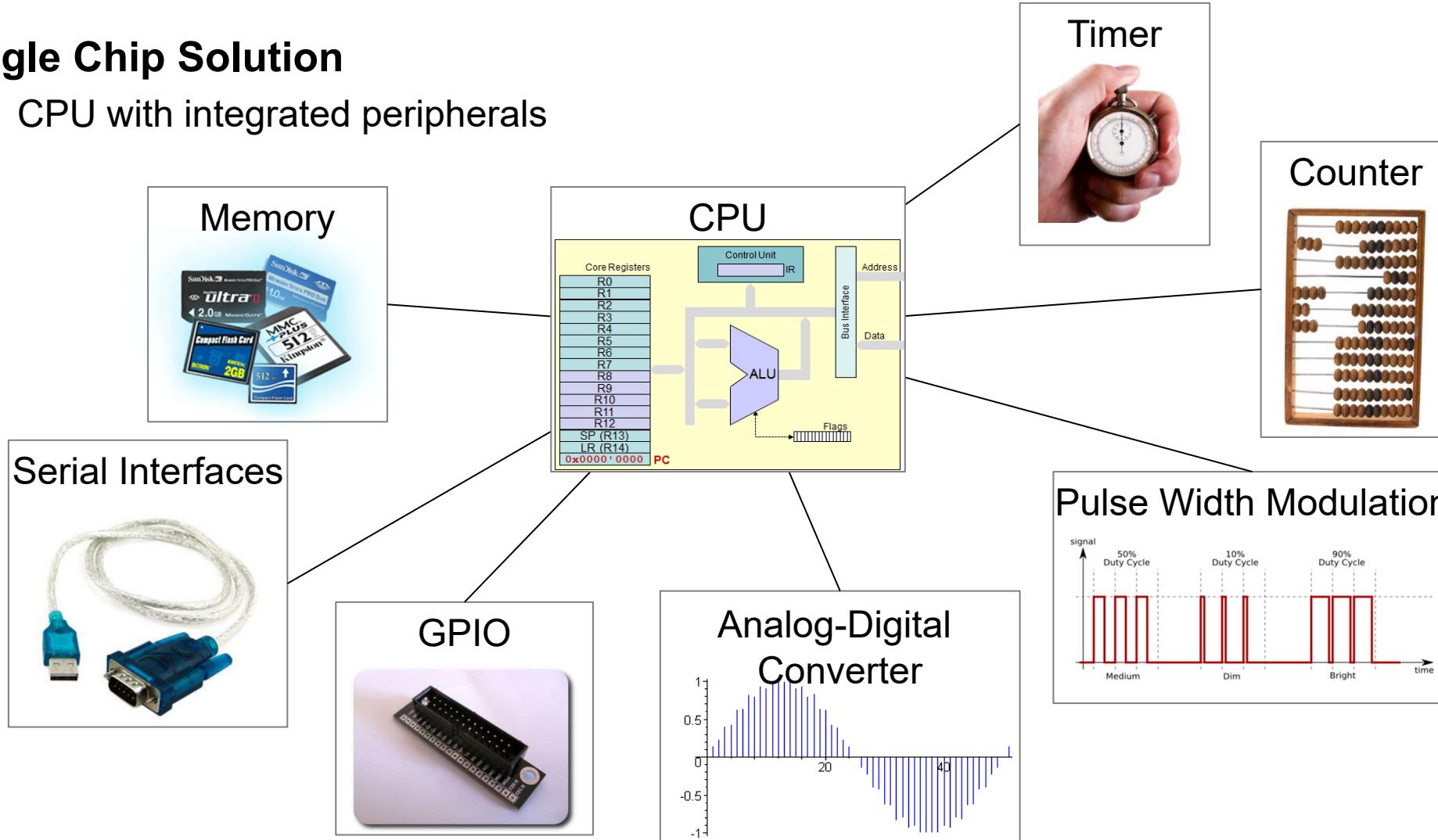
Learning Objectives

At the end of this lesson you will be able

- to enumerate the signal groups of a system bus
- to distinguish between 'synchronous' and 'asynchronous' bus timing
- to know what the term 'tri-state' means
- to interpret simple bus timing diagrams
- to describe the concept of a bus and how data is transferred on a bus
- to describe the function and purpose of control and status registers
- to explain the terms 'full address decoding' and 'partial address decoding'
- to access control registers from C
- to explain the meaning of the qualifier 'volatile' in C
- to analyze address decoding logic and to derive the applicable addresses or address ranges
- to derive an address decoding logic for a given address (range)
- to explain the function of wait states

■ Single Chip Solution

- CPU with integrated peripherals



■ Embedded Systems

LOW COST

USB stick, consumer,
power supplies



medical instruments

Real Time

anti-lock braking system,
process control

extreme environment



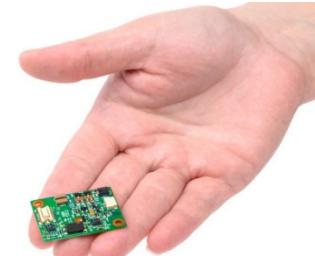
automotive, satellites

low power



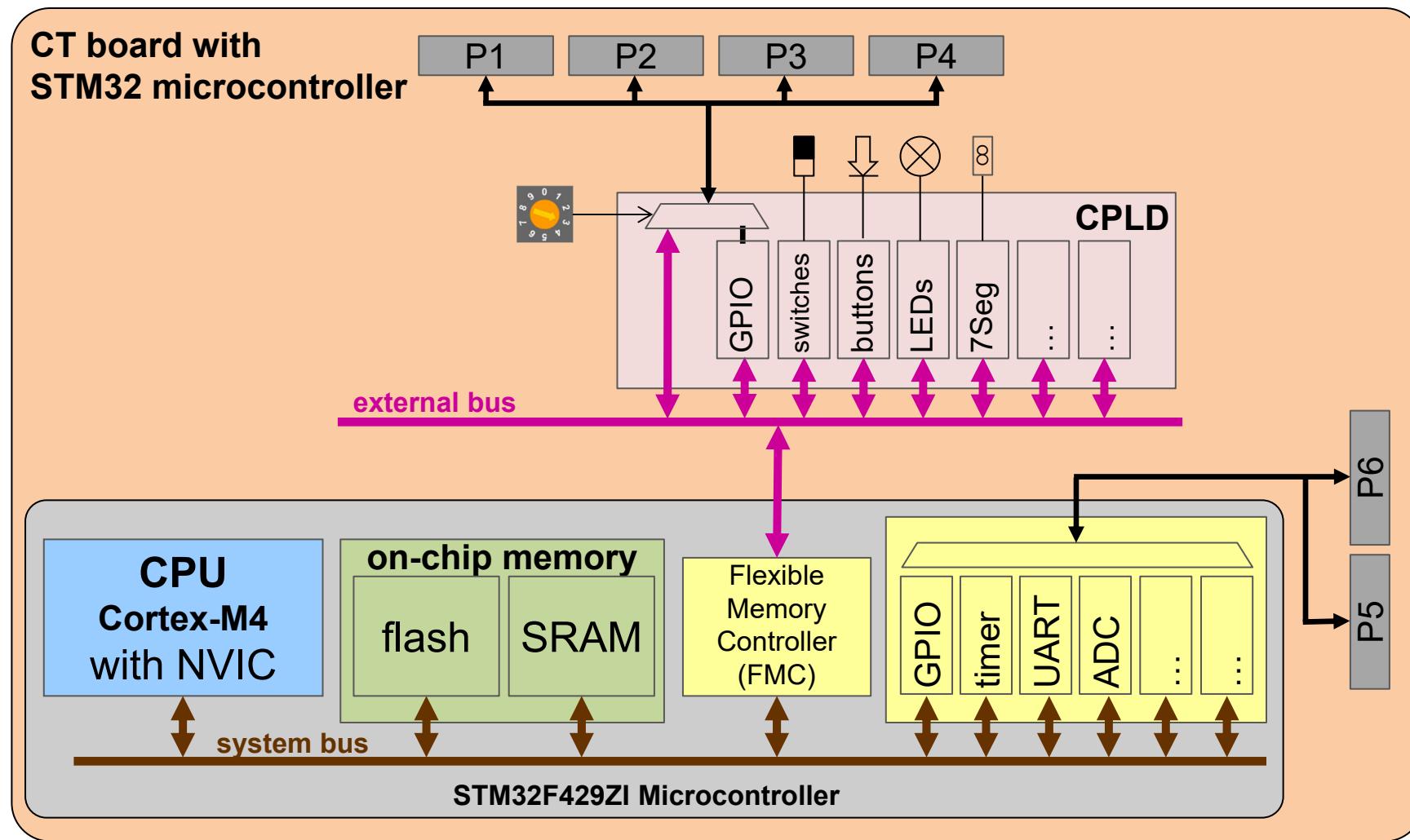
sensor networks,
autarkic systems

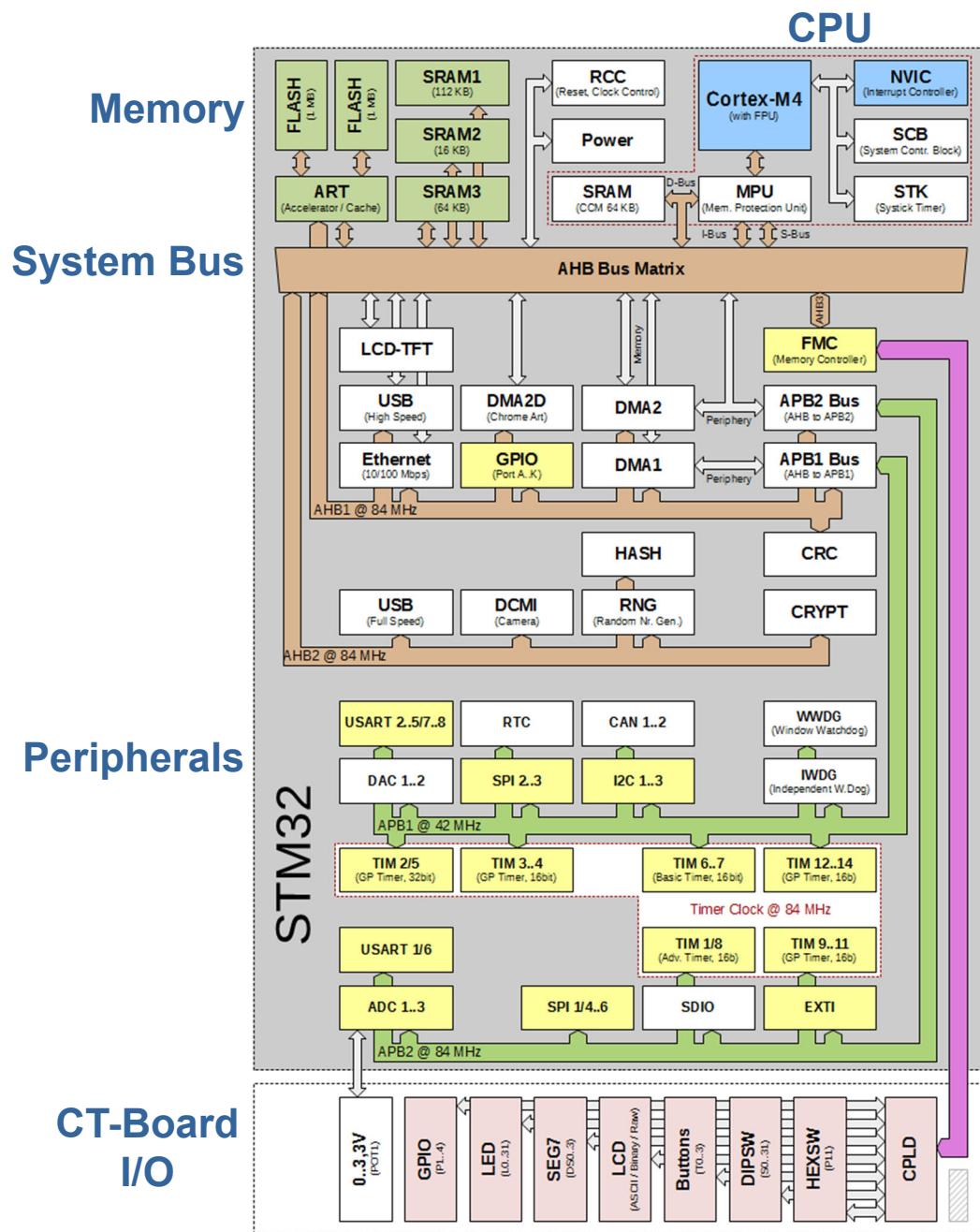
miniaturized



wearables

CT Board with STM32 Microcontroller



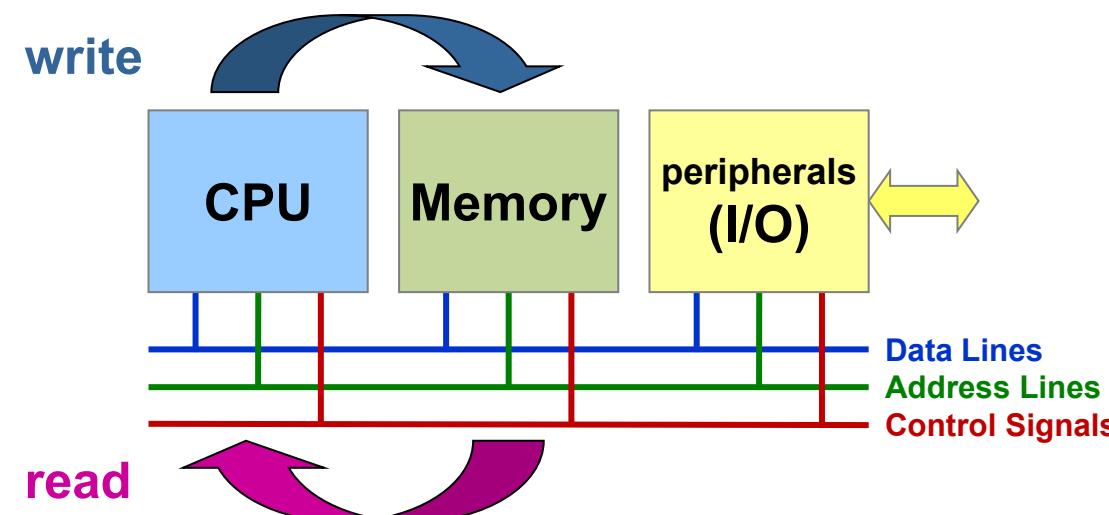


STM32 microcontroller

■ System Bus

- Interconnects CPU with memory and peripherals
- CPU acts as master¹⁾
 - Initiating and controlling all transfers
- Peripherals and memory act as slaves
 - Responding to requests

Etymology
*Bus from Latin *omnibus*, i.e. "for all"*



1) multi-master systems are not covered here

■ Bus Specification

Protocol and operations

Signals

- Number of signals
- Signal descriptions

Timing

- Frequency
- Setup and hold times

Electrical properties ¹⁾

- Drive strength
- Load

Mechanical
requirements ¹⁾

1) not covered in this course

■ Signal Groups

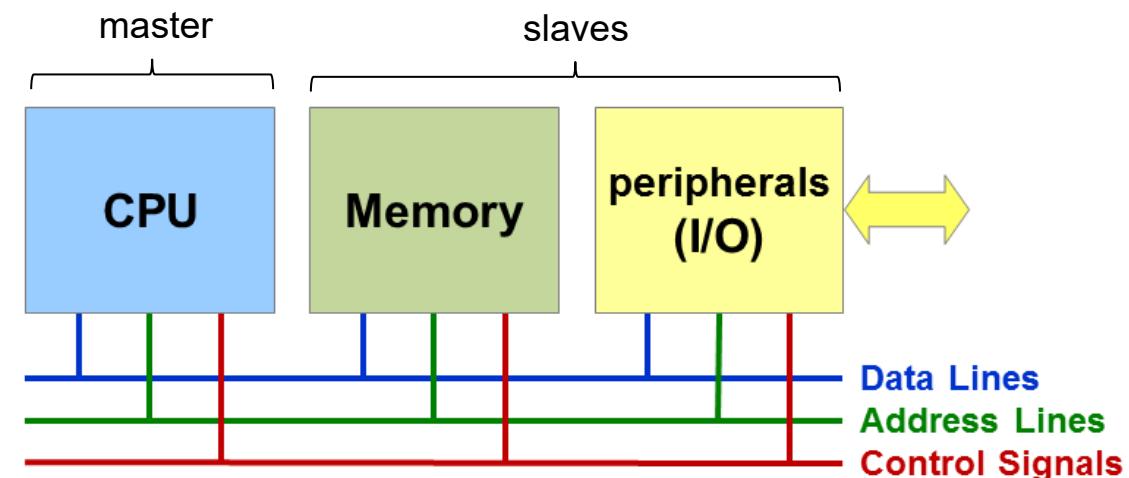
- Address lines
 - Unidirectional: From master to slave
 - Number of lines → size of address space
- Data lines
 - 8, 16, 32 or 64 parallel lines of data
 - bidirectional (read/write)
- Control signals
 - Control read/write direction
 - Provide timing information

Cortex-M

32 address lines

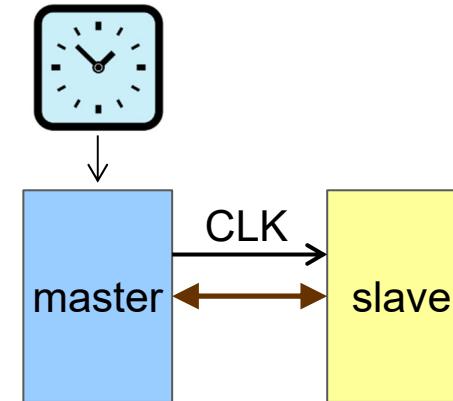
→ $2^{32} = 4$ Giga addresses

0x0000'0000 – 0xFFFF'FFFF

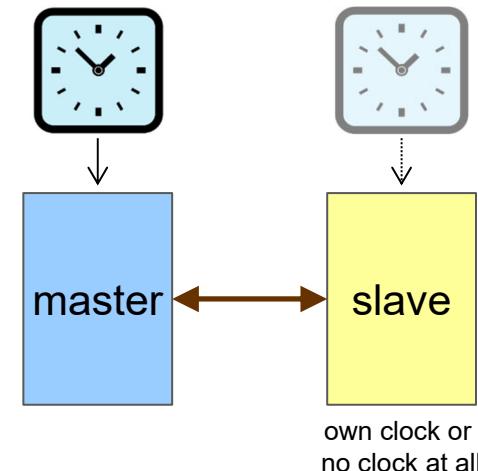


■ Bus Timing Options

- Synchronous
 - Master and slaves use a common clock¹⁾
 - Clock edges control bus transfer on both sides
 - Used by most on-chip busses
 - Off-chip: DDR and synchronous RAM



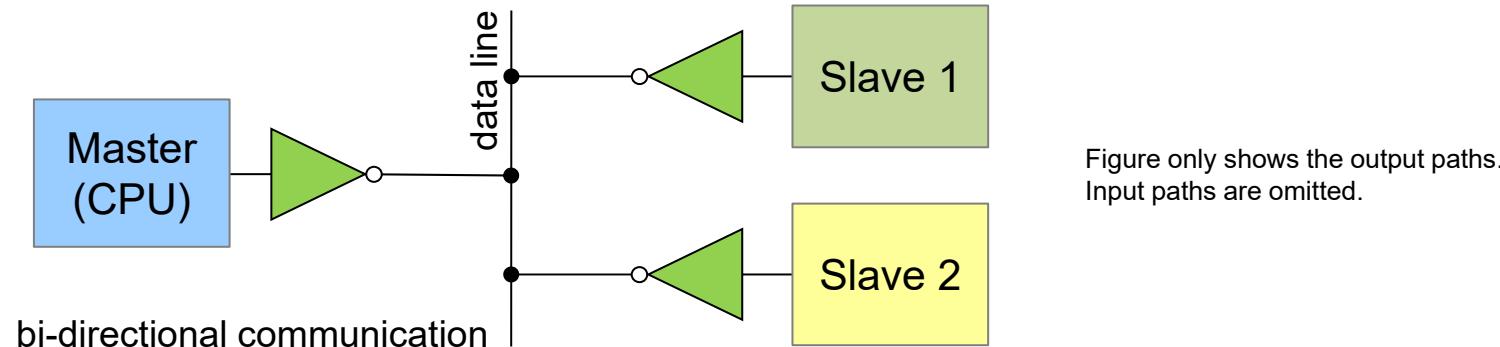
- Asynchronous
 - Slaves have no access to the clock of the master
 - Control signals carry timing information to allow synchronization
 - Widely used for low data-rate off-chip memories
→ parallel flash memories and asynchronous RAM



1) Often this is a dedicated clock signal from master to slave but the clock can also be encoded in a data signal

■ Multiple Devices Driving the Same Data Line

- What if one device drives a logic '1' (Vcc) and the other a logic '0' (Gnd)?
 - Electrical short circuit → bus contention (dt. "Streitigkeit")

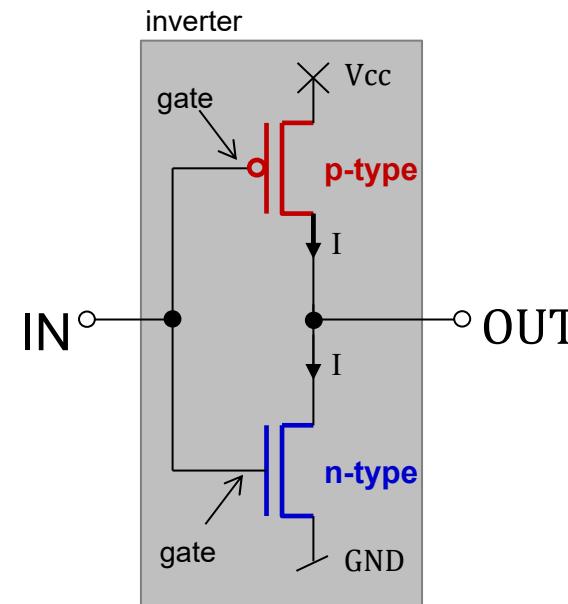


- CPU defines who drives the data bus at which moment in time
 - Write CPU drives bus all slave drivers disconnected
 - Read CPU driver disconnected single slave drives bus selected through values on address lines other slave drivers disconnected

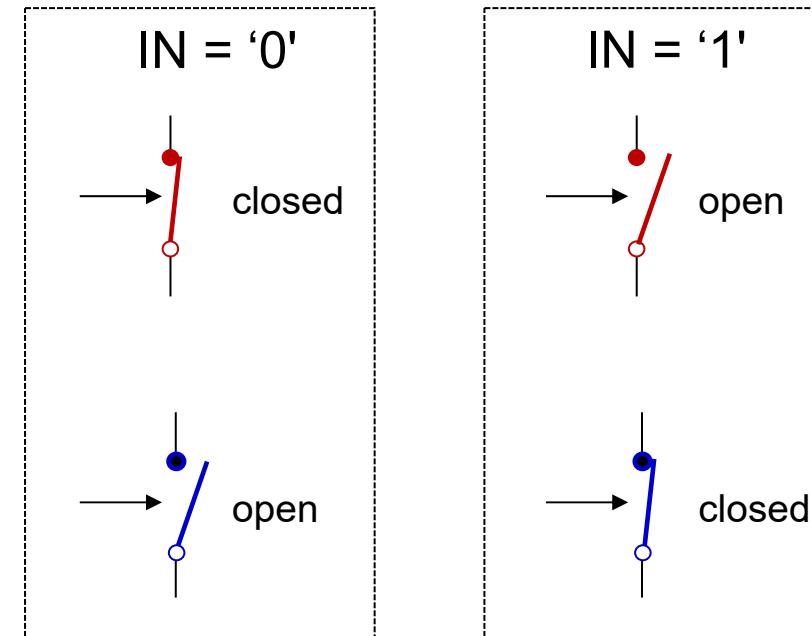
But, how can a driver be disconnected electrically?

■ CMOS¹⁾ Inverter

- Complementary switches (transistors)
 - **p-type** and **n-type** have opposite open-close behavior



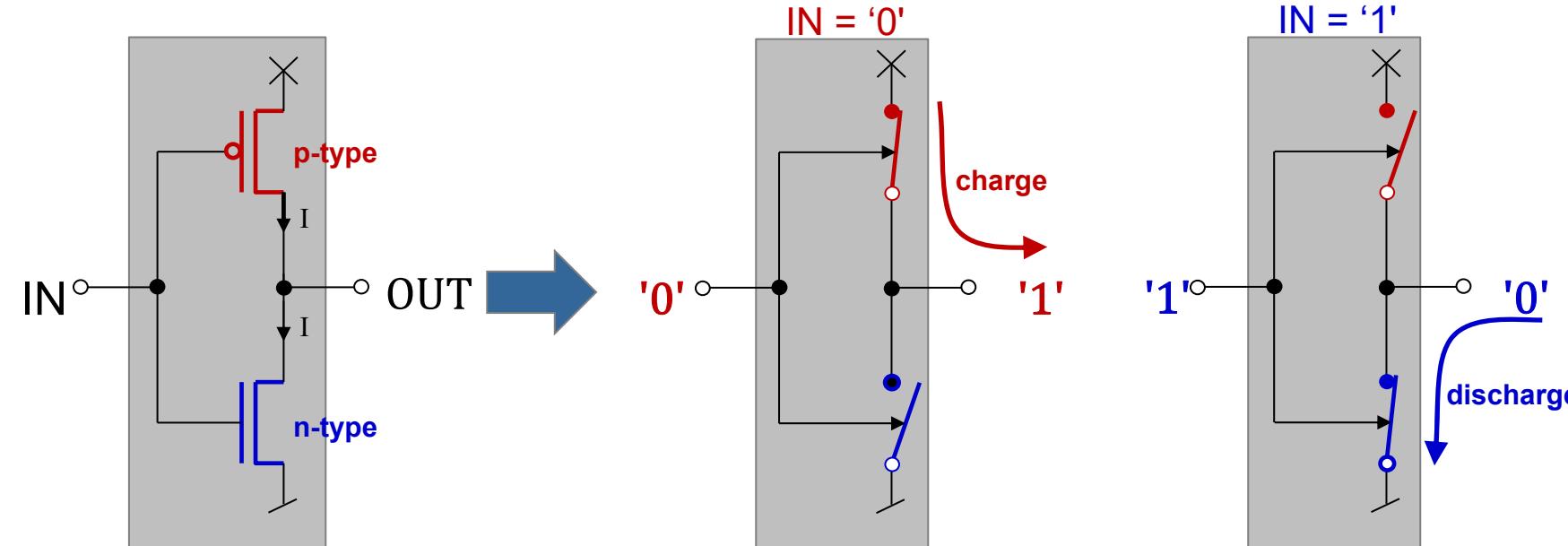
e.g. $V_{cc} = 3V = '1'$
 $GND = 0V = '0'$



1) CMOS: Complementary Metal-Oxide-Semiconductor

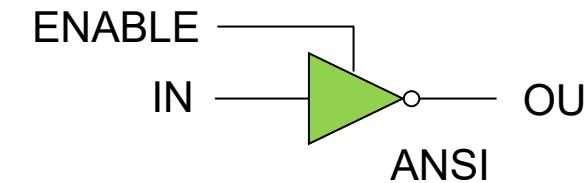
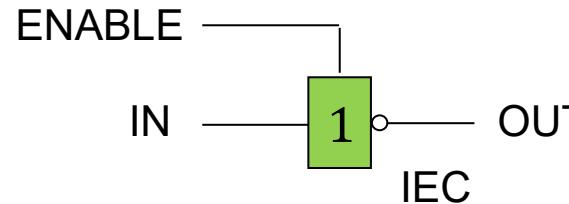
■ CMOS Inverter

- Complementary switches (transistors)



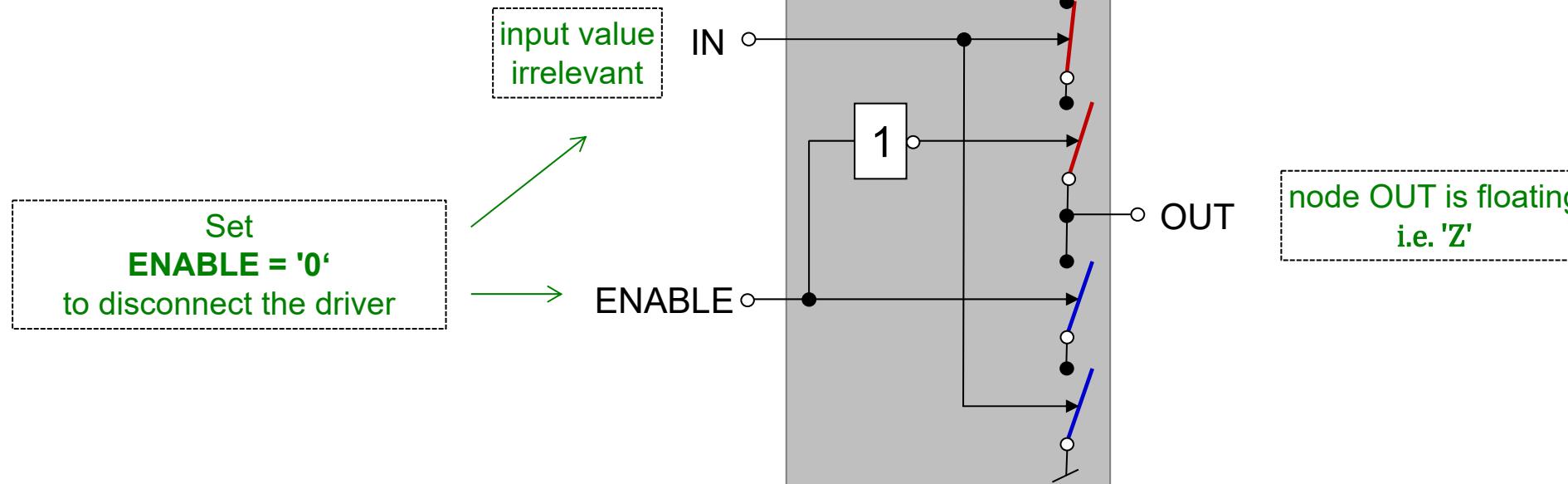
- A buffer is built by connecting two inverters in series

■ CMOS Tri-State Inverter



ENABLE	OUT
'1'	! IN
'0'	'Z'

'Z' = high impedance



■ CMOS Tri-State Inverter - Implementation

ENABLE

IN

IEC

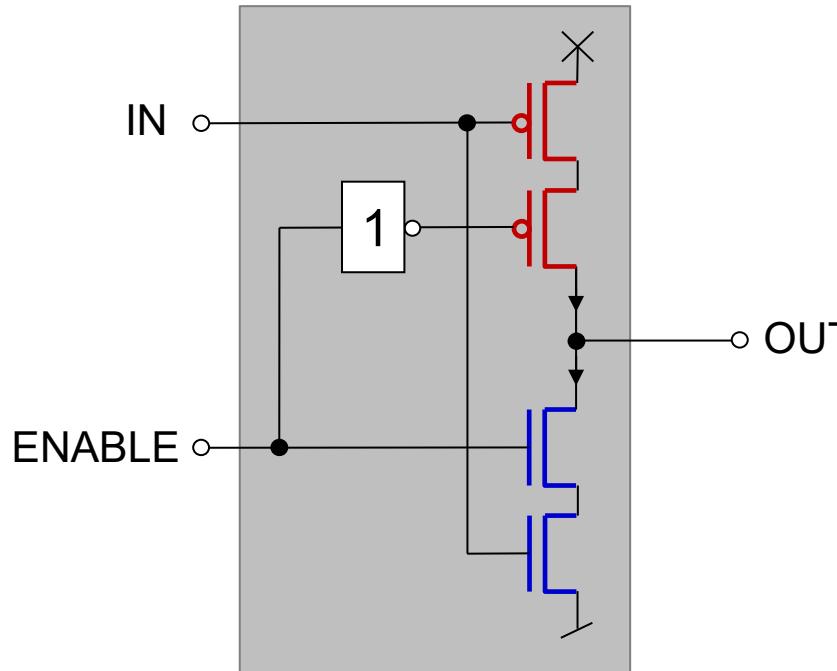
ENABLE

IN

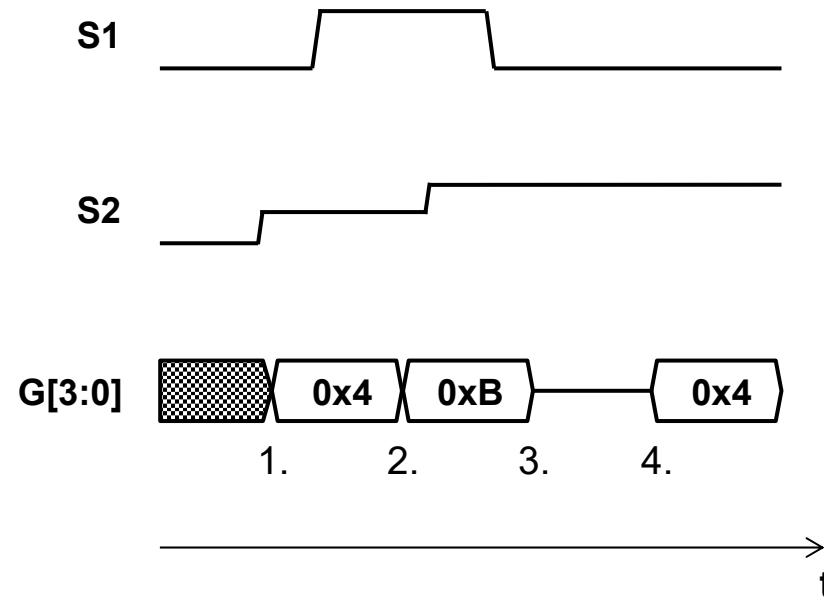
OUT

ANSI

ENABLE	OUT
'1'	! IN
'0'	'Z'



■ Timing Diagrams: Notations



Signal S1 changing from '0' to '1' (rising edge) and back from '1' to '0' (falling edge)

Signal S2 changing from '0' to high-impedance ('Z', tri-state) and then to '1'

- Group G of 4 signals changing
1. from an unknown value to 0x4
i.e. $G[3] = '0'$, $G[2] = '1'$, $G[1] = '0'$, $G[0] = '0'$
 2. from 0x4 to 0xB
 3. from 0xB to all signals high-impedance ('Z', tri-state)
 4. from high-impedance back to 0x4

■ Example Uses External Bus from ST Microelectronics

- Reason: Internal workings of the system bus are not disclosed by STM
- Signal names, bus protocol and timing based on external synchronous STM32F429xx mode instead
 - *For details see*
 - ▶ *Chapter 37, Flexible memory controller (FMC) in ST Reference Manual RM0090*
 - ▶ *Datasheet STM32F429xx*
Figure 60 Synchronous non-multiplexed NOR/PSRAM read timings
Figure 61 Synchronous non-multiplexed PSRAM write timings

■ Naming Convention

- Letter 'N' prefix in signal name (Nxxxx) means active-low signal
 - E.g. NOE means 'NOT OUTPUT ENABLE'
 - NOE = '0' → output enabled
 - NOE = '1' → output disabled

Synchronous Bus

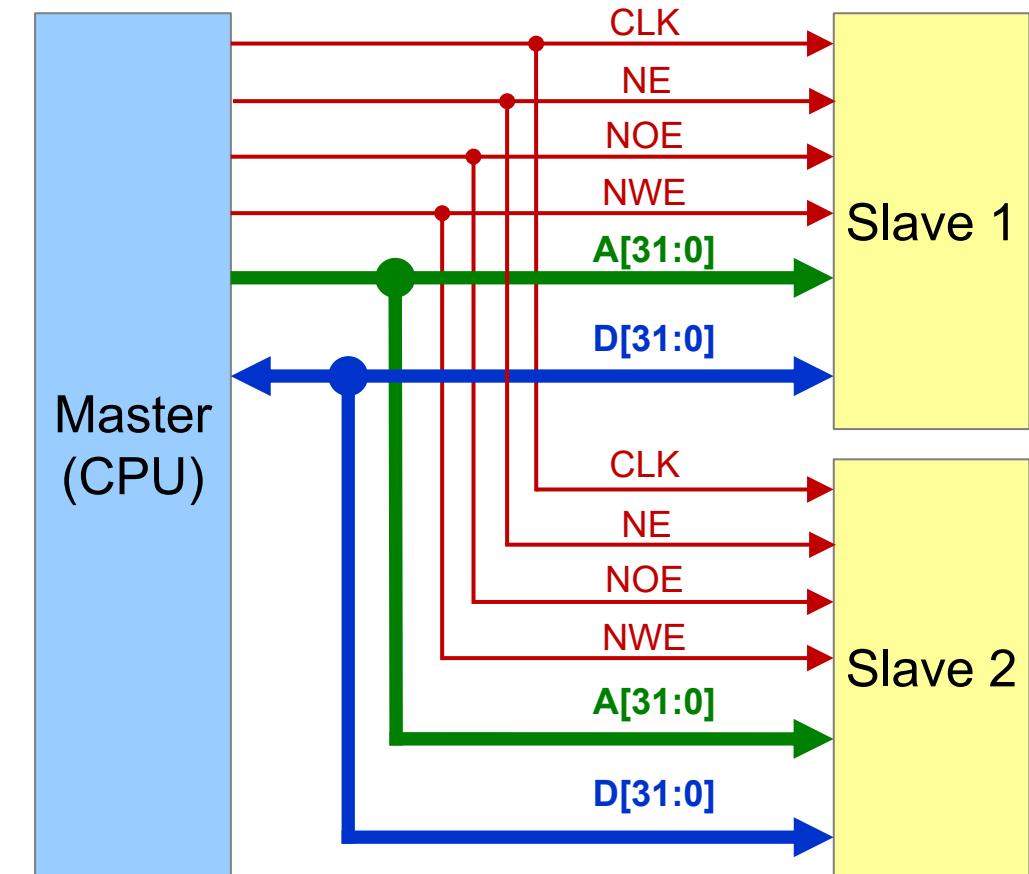
■ Block Diagram

- Address lines A[31:0]
- Data lines D[31:0]
- Control
 - CLK
 - NE
 - NWE
 - NOE

Not Enable
indicates start and end
of cycle, active-low

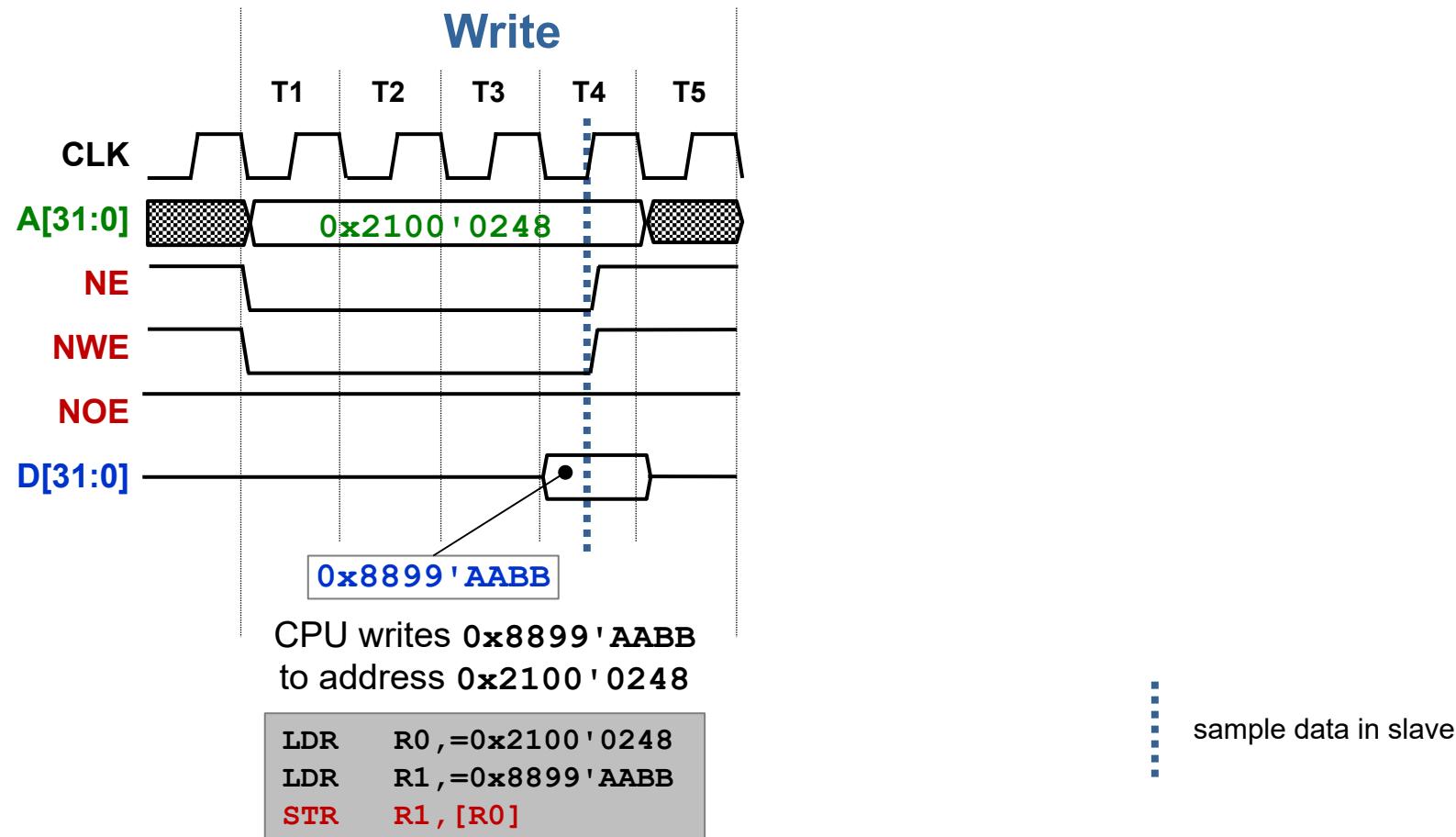
Not Write Enable
active-low

Not Output Enable
(read), active-low



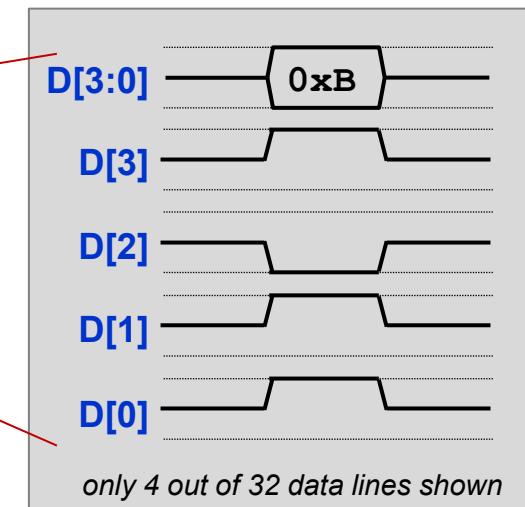
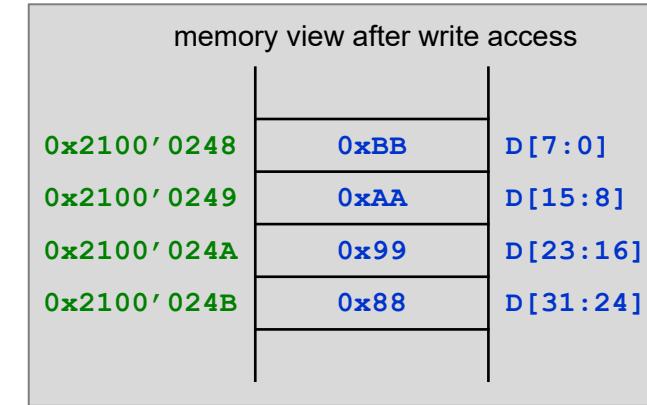
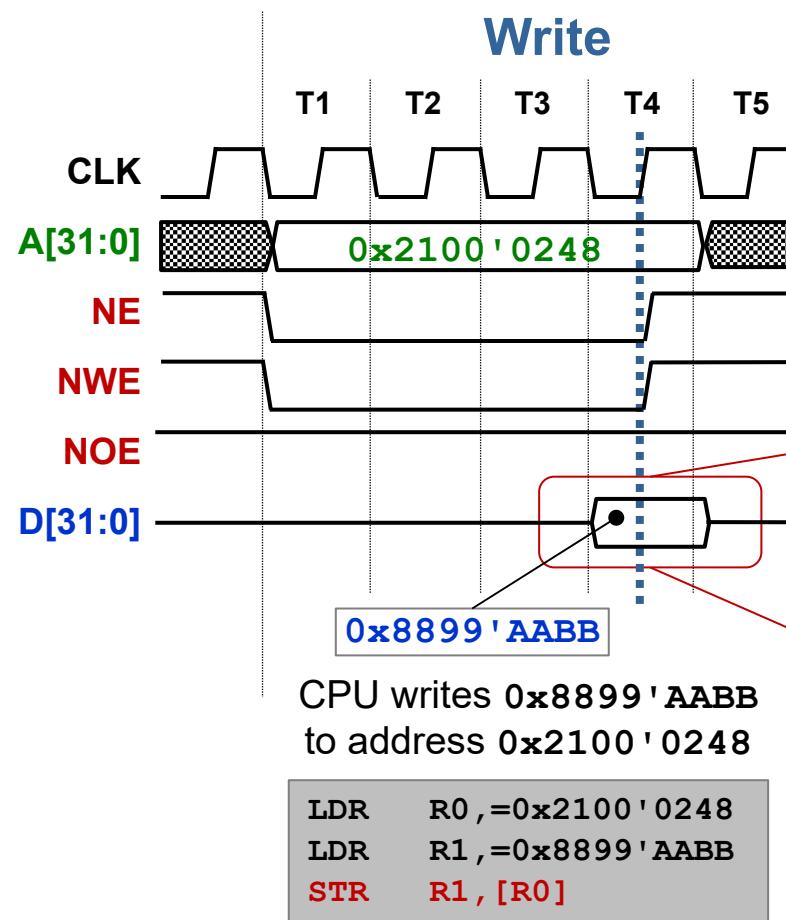
→ single line
→ multiple lines

■ Timing Diagram



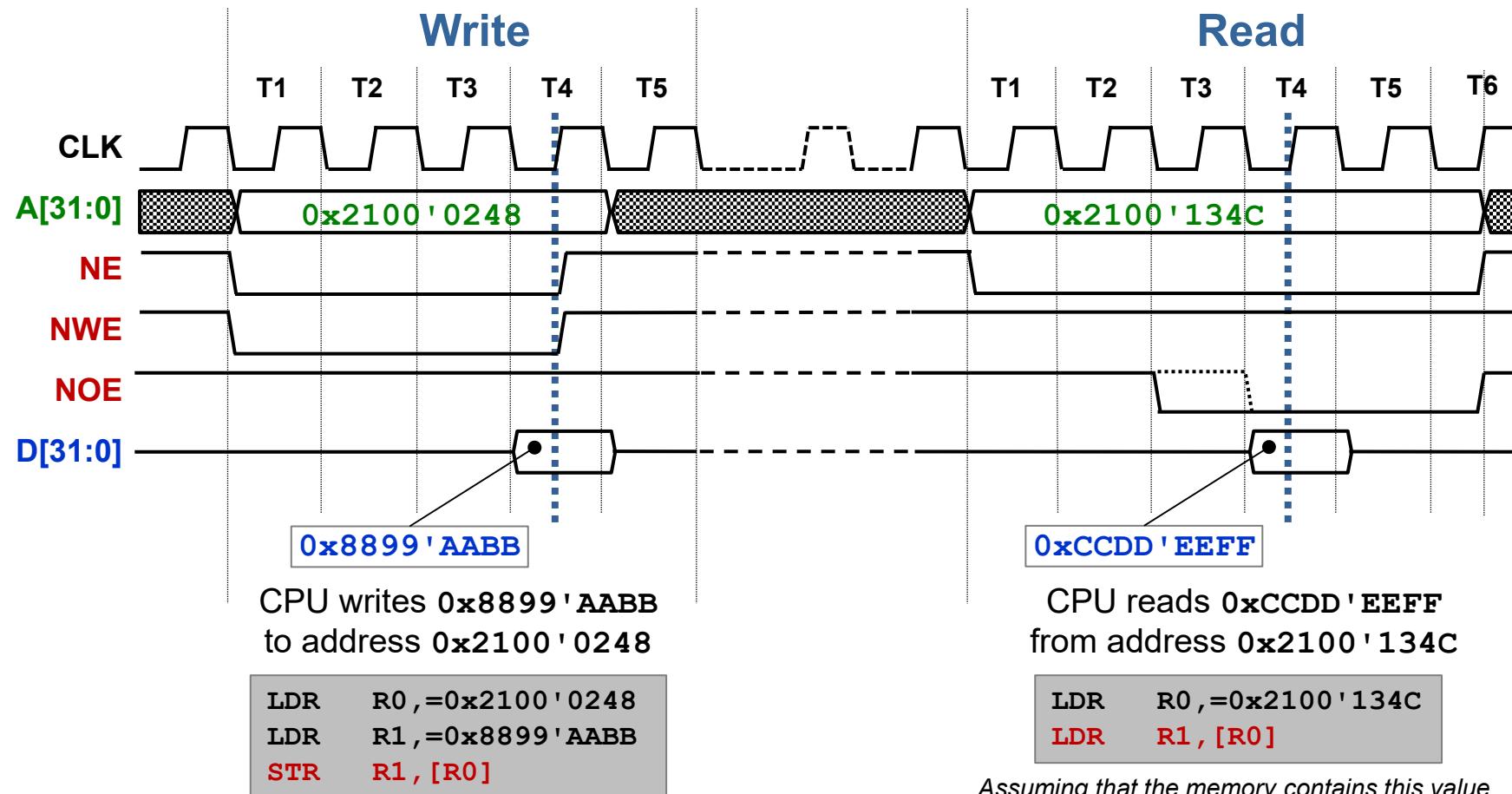
Synchronous Bus

■ Timing Diagram



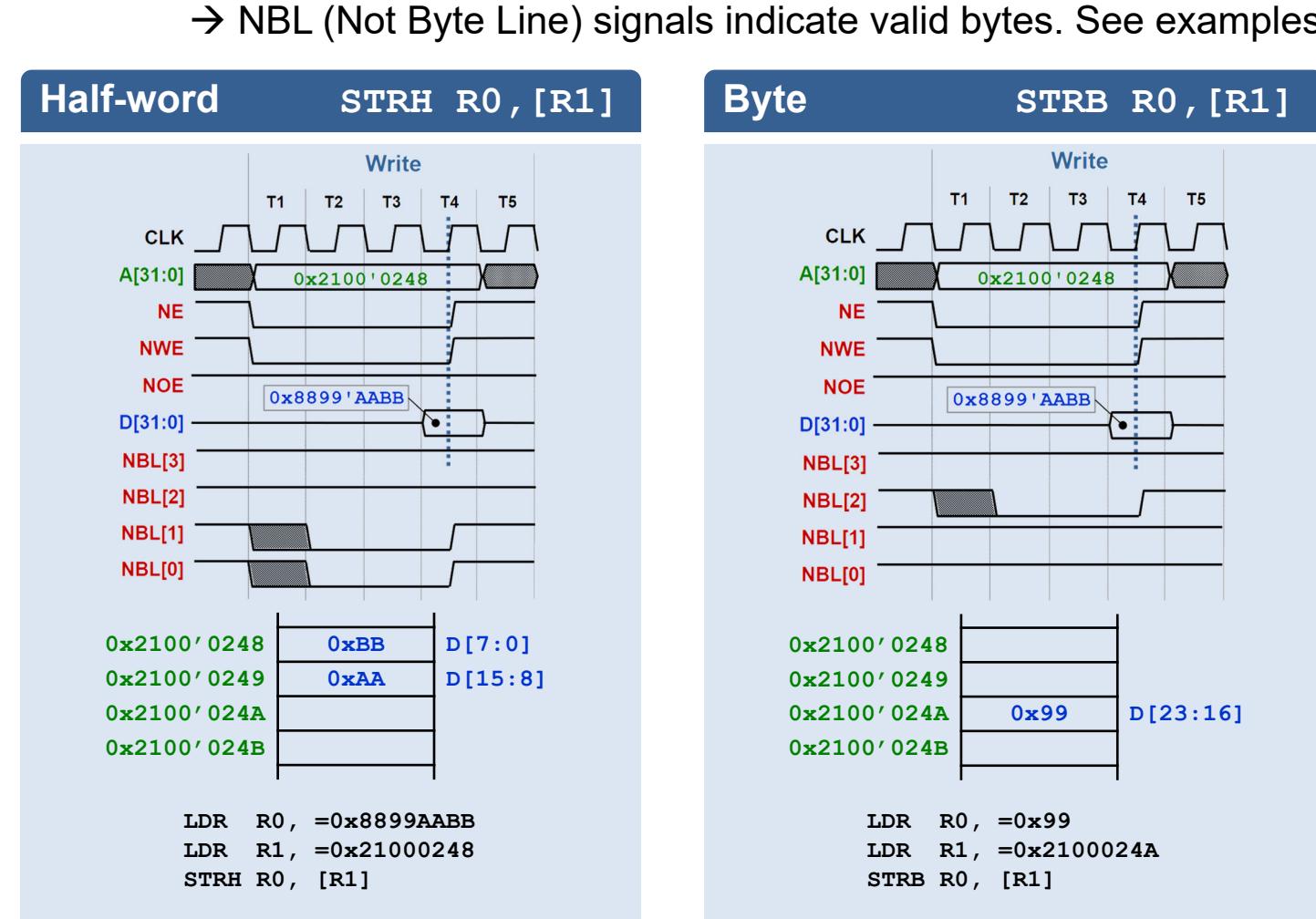
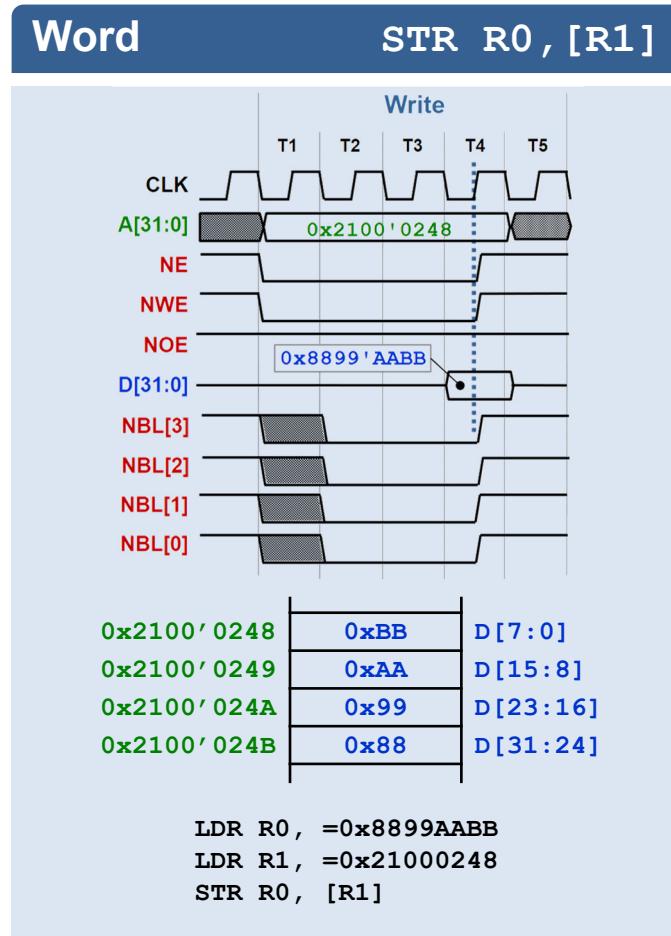
■ Timing Diagram

Note: The given timing is based on many design decisions by ST



Synchronous Bus

■ Bus Access Size

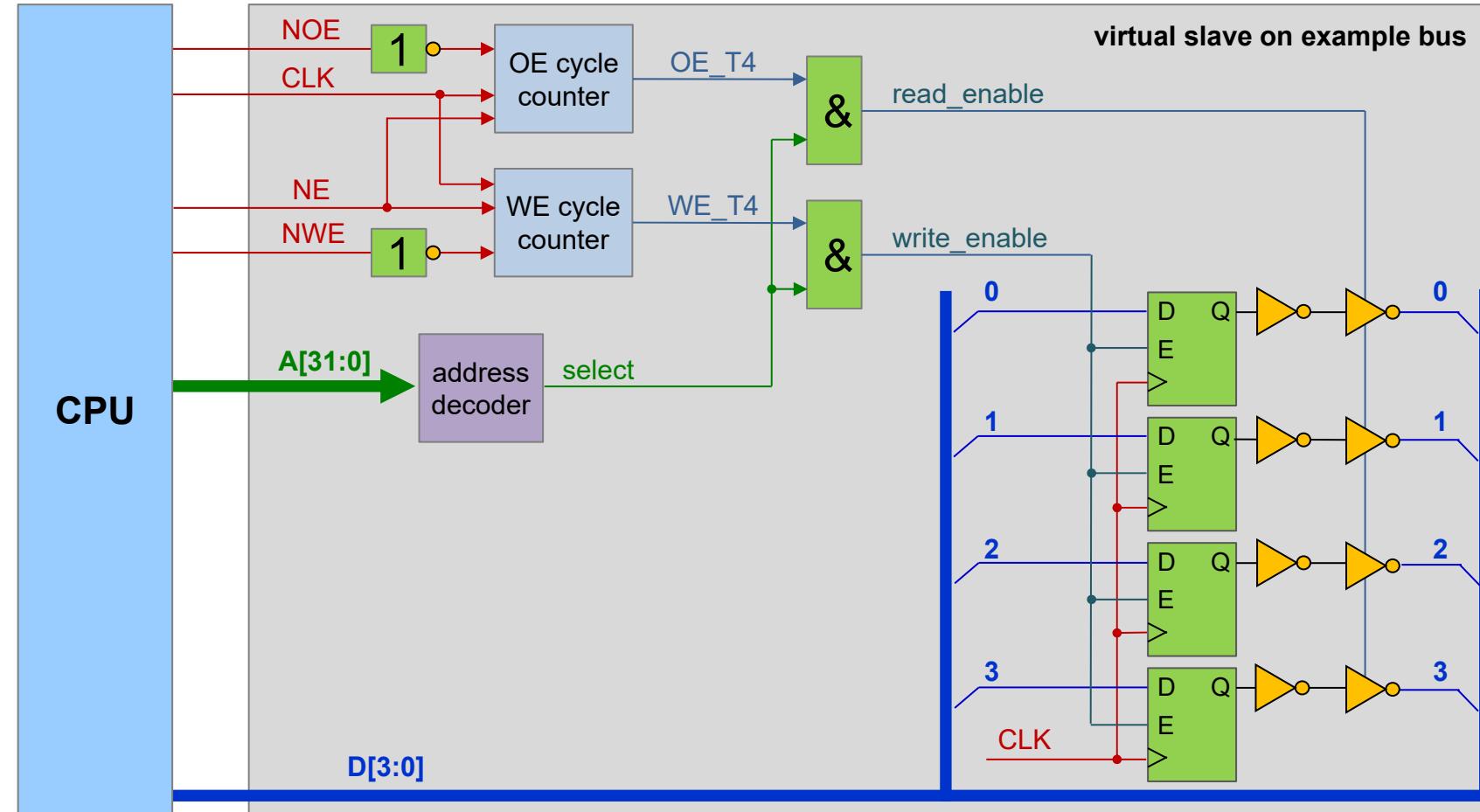


- Exact position of falling edge on NBL varies with chip version
- Value on unused data lines are unknown, figures show assumptions

Control and Status Registers

■ Hardware Slave (Peripheral)

Figure only shows lower 4 data bits



Control and Status Registers

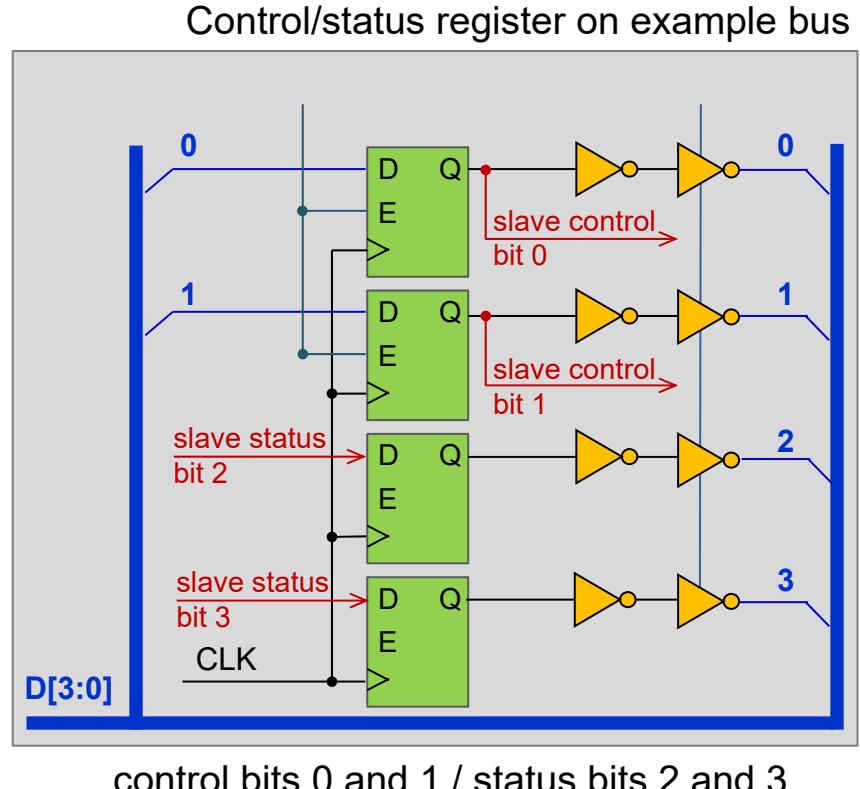
■ Control Bits

- Allow CPU to configure slave
- CPU (software) writes to register bit
- Slave hardware uses output of register bit
- Usually read/write

■ Status Bits

- Allow CPU to monitor a slave
- Slave writes status into register bit
- CPU (software) reads register bit
- Usually read-only

Same register may contain control and status bits



Examples

Status

Switch, "Data byte received on serial interface"

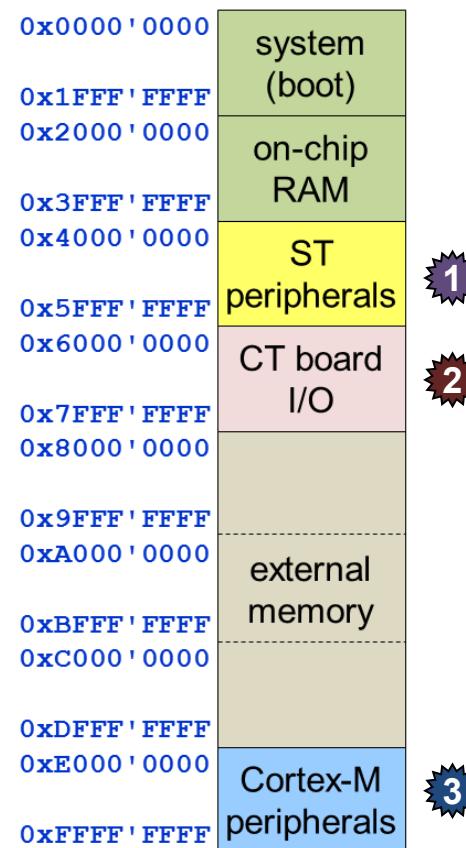
Control

LED, "enable interface"

Control and Status Registers

■ Control and Status Registers on CT Board

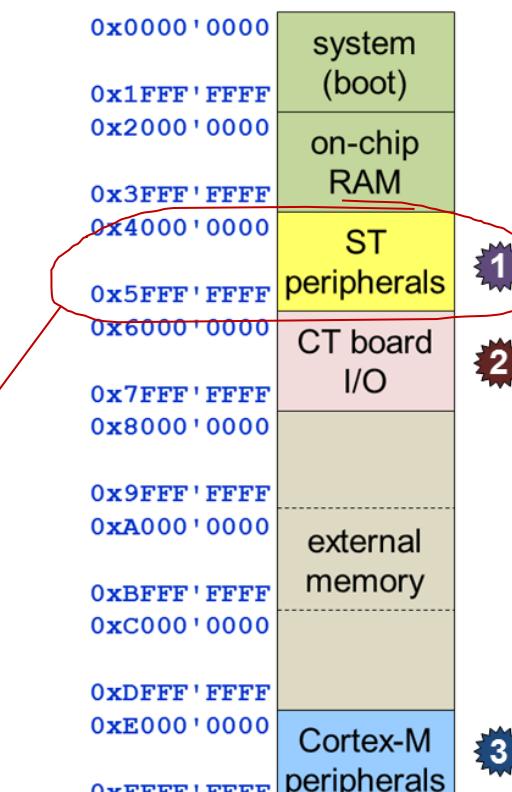
- Chip-internal and external registers



- ➊ ST peripherals
e.g. Timers, ADC, UART, SPI, ...
- ➋ CT board I/O
LEDs, dip switches, LCD, ...
- ➌ ARM Cortex-M
NVIC, ...

Control and Status Registers

■ ST Registers



The diagram illustrates the memory map for STM32F4xx devices. It shows four main vertical columns: system (boot) RAM, on-chip RAM, ST peripherals, and Cortex-M peripherals. The ST peripherals column is highlighted with a yellow background and circled with a red line. Three numbered callouts point to specific address ranges: Callout 1 points to the ST peripherals range (0x4000'0000 to 0x5FFF'FFFF), Callout 2 points to the CT board I/O range (0x6000'0000 to 0x7FFF'FFFF), and Callout 3 points to the Cortex-M peripherals range (0xE000'0000 to 0xFFFF'FFFF). The memory map is divided into several regions with boundary addresses:

Region	Boundary Address	Description
system (boot)	0x0000'0000	
on-chip RAM	0x1FFF'FFFF	
ST peripherals	0x2000'0000	Includes: FMC, RNG, HASH, CRYP, DCMI, USB OTG FS
CT board I/O	0x3FFF'FFFF	
external memory	0x4000'0000	Includes: FSMC, FMC
Cortex-M peripherals	0x5FFF'FFFF	
	0x6000'0000	
	0x7FFF'FFFF	
	0x8000'0000	
	0x9FFF'FFFF	
	0xA000'0000	
	0xBFFF'FFFF	
	0xC000'0000	
	0xDFFF'FFFF	
	0xE000'0000	
	0xFFFF'FFFF	

RM0090 Reference manual
STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx advanced ARM-based 32-bit MCUs

2.3 Memory map

See the datasheet corresponding to your device for a comprehensive diagram of the memory map. *Table 2* gives the boundary addresses of the peripherals available in all STM32F4xx devices.

Table 2. STM32F4xx register boundary addresses

Boundary address	Peripheral	Bus	Register map
0xA000 0000 - 0xA000 0FFF	FSMC control register (STM32F405xx/07xx and STM32F415xx/17xx)/ FMC control register (STM32F42xxx and STM32F43xxx)	AHB3	Section 36.6.9: FSMC register map on page 1573 Section 37.8: FMC register map on page 1653
0x5006 0800 - 0x5006 0BFF	RNG	AHB2	Section 24.4.4: RNG register map on page 752
0x5006 0400 - 0x5006 07FF	HASH		Section 25.4.9: HASH register map on page 776
0x5006 0000 - 0x5006 03FF	CRYP		Section 23.6.13: CRYP register map on page 745
0x5005 0000 - 0x5005 03FF	DCMI		Section 15.8.12: DCMI register map on page 473
0x5000 0000 - 0x5003 FFFF	USB OTG FS		Section 34.16.6: OTG_FS register map on page 1303

see chapter 2.3 of reference manual, page 64ff

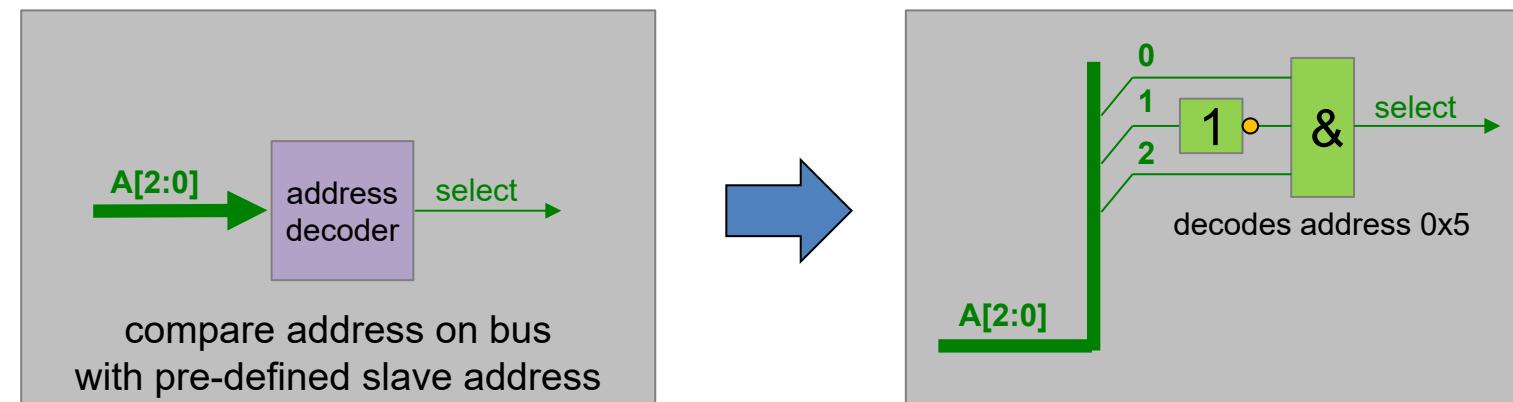
only part of table shown

■ How Does a Slave Know, That It Is the Target of an Access?

- Answer: Address decoding

■ Address Decoding

- Interpretation of address line values
 - See whether bus access targets a particular address or address range
- Example with 3 address lines
 - Select equal '1' indicates that the CPU wants to access address 0x5



Full Address Decoding

- All address lines are decoded (checked)
- A control register can be accessed at exactly one location
- 1 : 1 mapping
 - A unique address maps to a single hardware register (physical memory location)

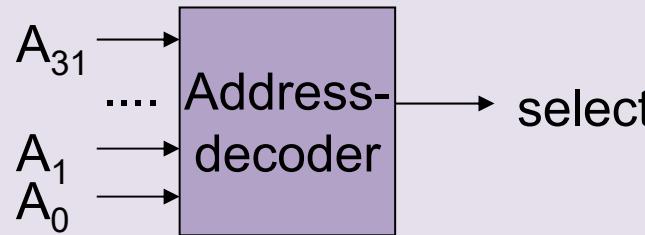
Partial Address Decoding

- Only a sub-set of the address lines is decoded
- Detects an address range or a set of addresses
- n : 1 mapping
 - n unique addresses map to the same hardware register (physical memory location)
- Motivation
 - Simpler decoding
 - Aliasing: Map a hardware register to several addresses

■ Example for 32-bit address space of Cortex-M

Full Address Decoding

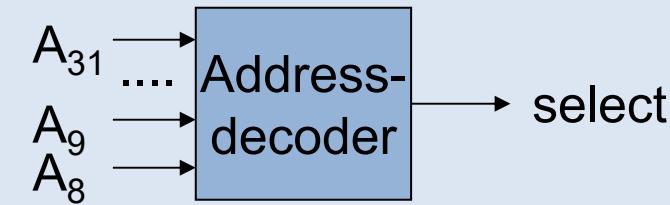
- all addresses from A31 to A0



- select is active for exactly one address
- E.g. at **0x4000'8234**

Partial Address Decoding

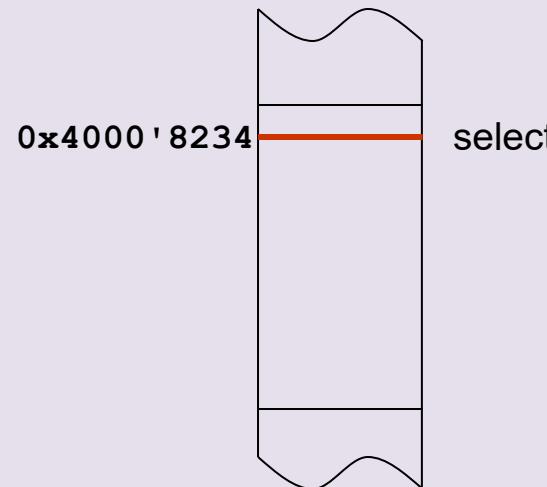
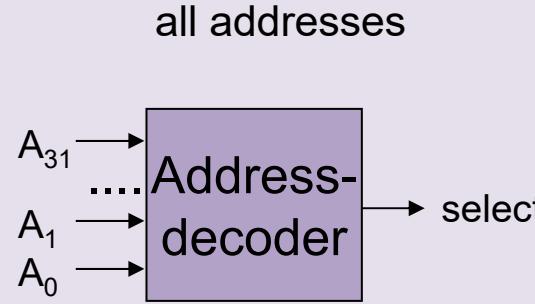
- only addresses from A31 to A8



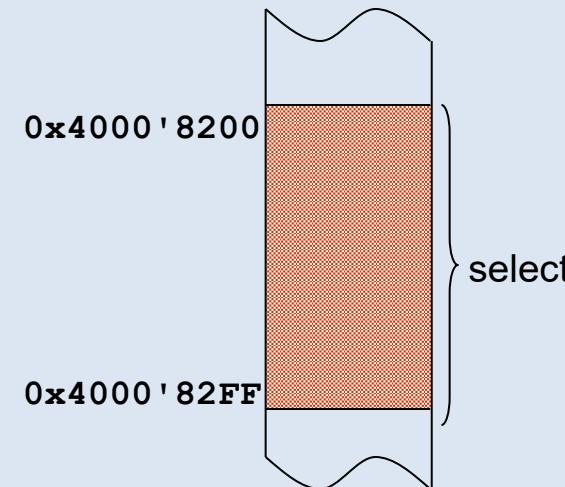
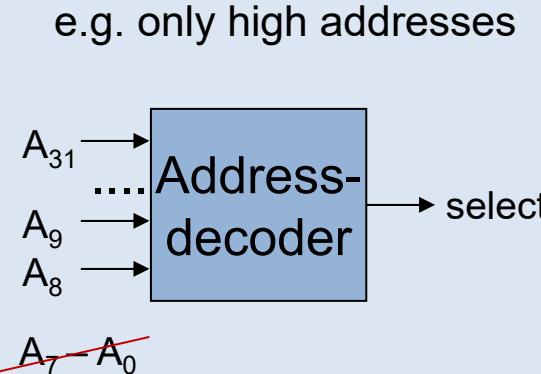
- select is active for any address within a given range (e.g. ignoring some lower address lines)
- E.g. from **0x4000'8200** to **0x4000'82FF**
→ **0x4000'82xx**

Address Decoding

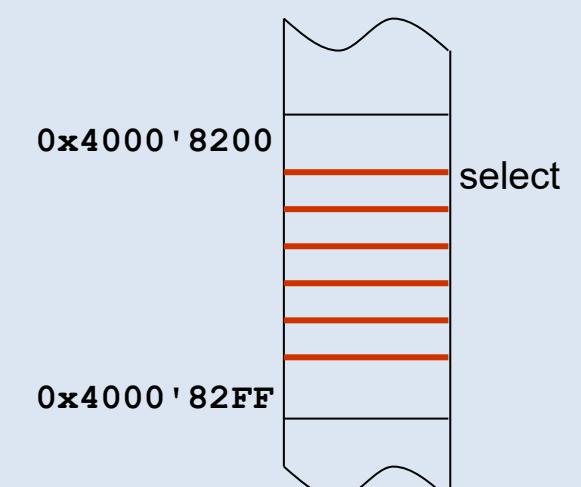
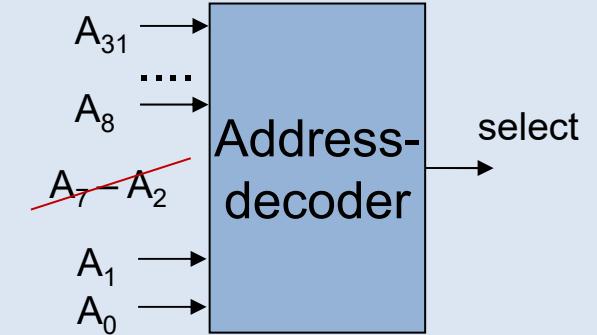
Full Address Decoding



Partial Address Decoding



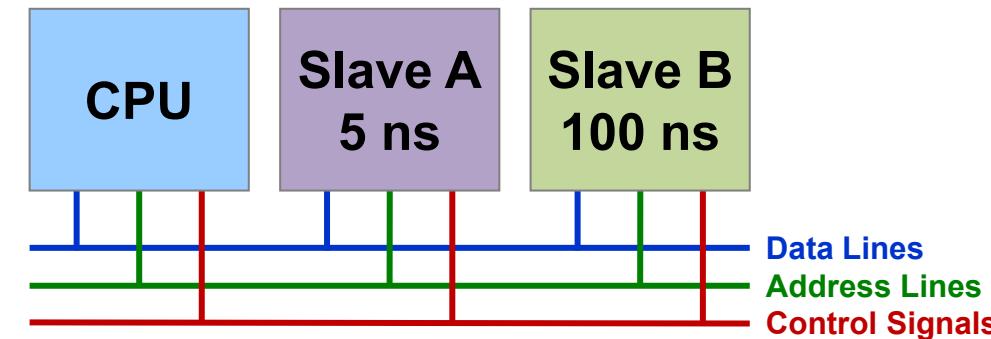
e.g. high and low addresses



address values are examples

■ Problem: Individual Slave Access Times

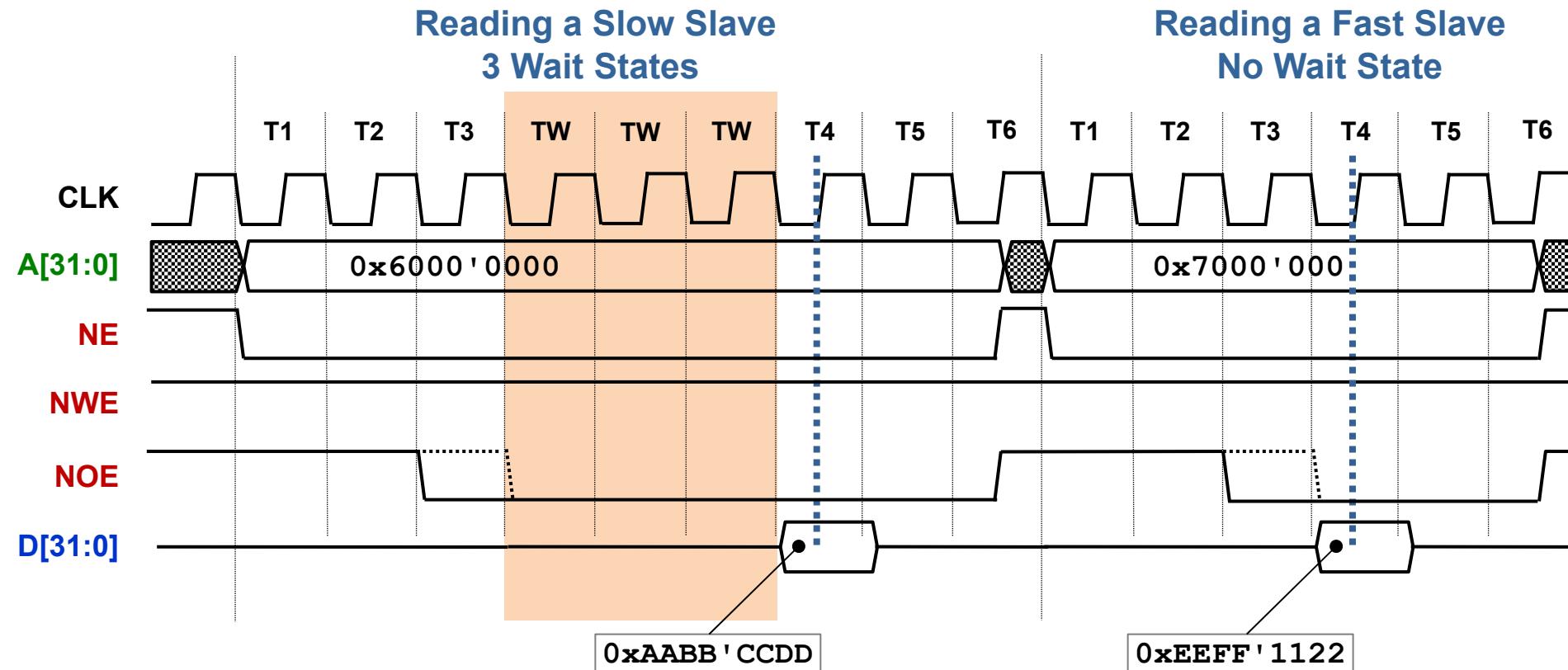
- If slowest slave defines bus cycle time
 - Reduced bus performance
- How can we get an individual bus cycle time for each slave?



Slow Slaves (Peripherals)

■ Introduce Individual Wait States for Slow Slaves

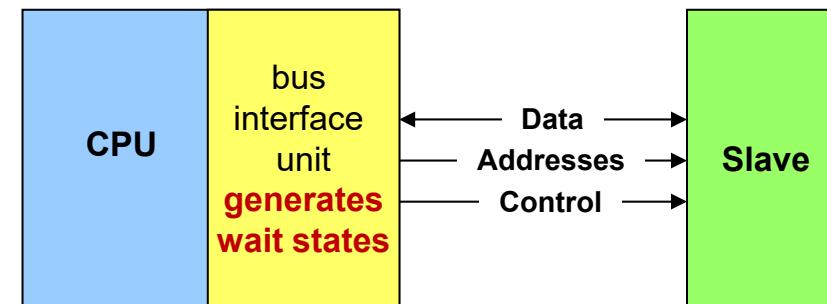
- Wait states are inserted depending on the address of an access



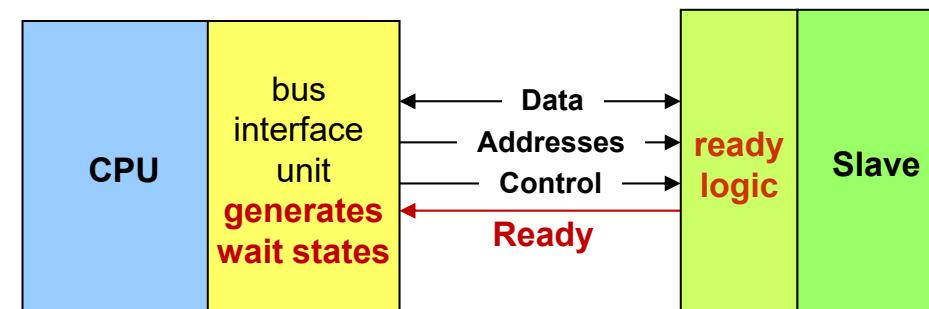
Slow Slaves (Peripherals)

■ Two Possibilities

1. Individual wait states can be programmed at a bus interface unit
 - Depending on the address of the bus cycle



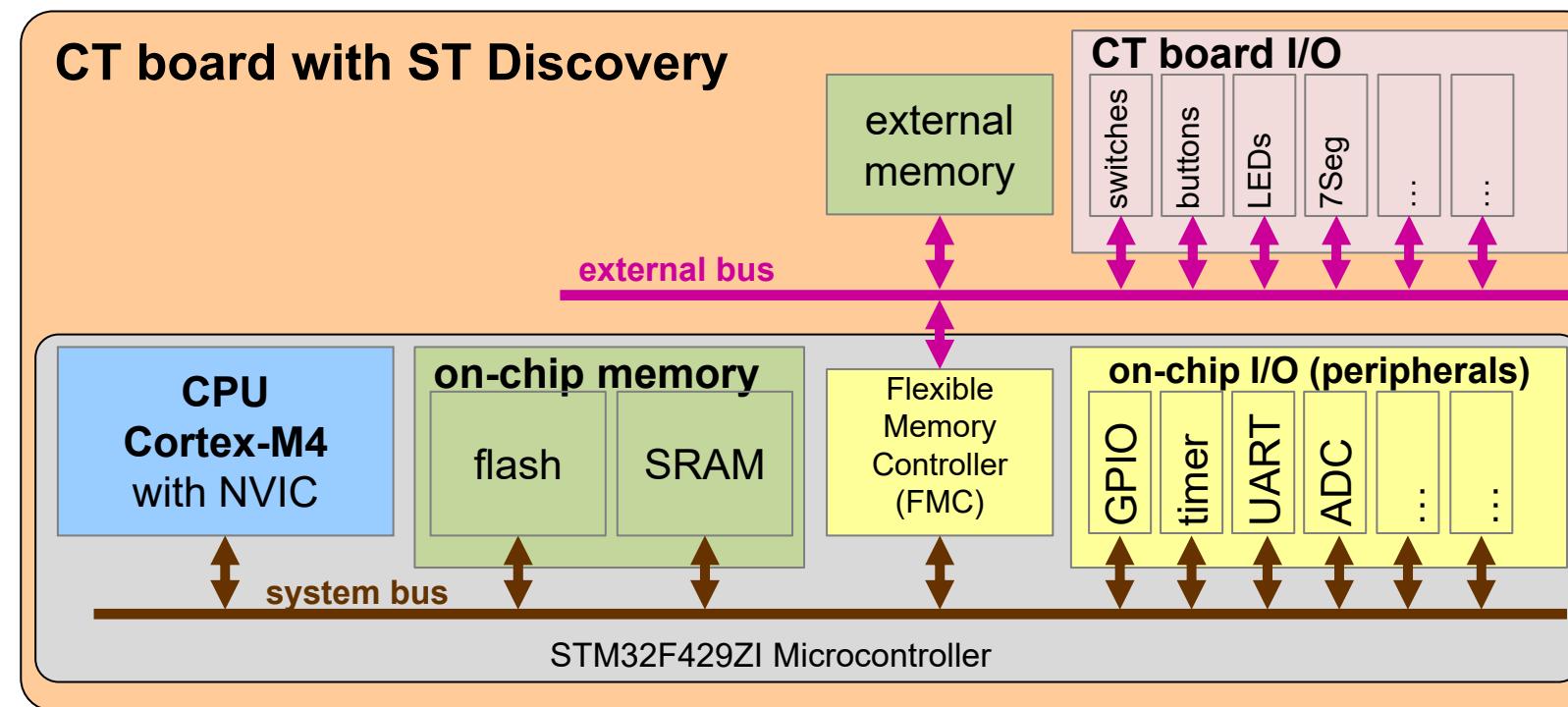
2. Slave tells bus interface unit when it is ready
 - Well suited for slaves with long or variable access times



Ready signal indicates when the slave is ready to provide the data

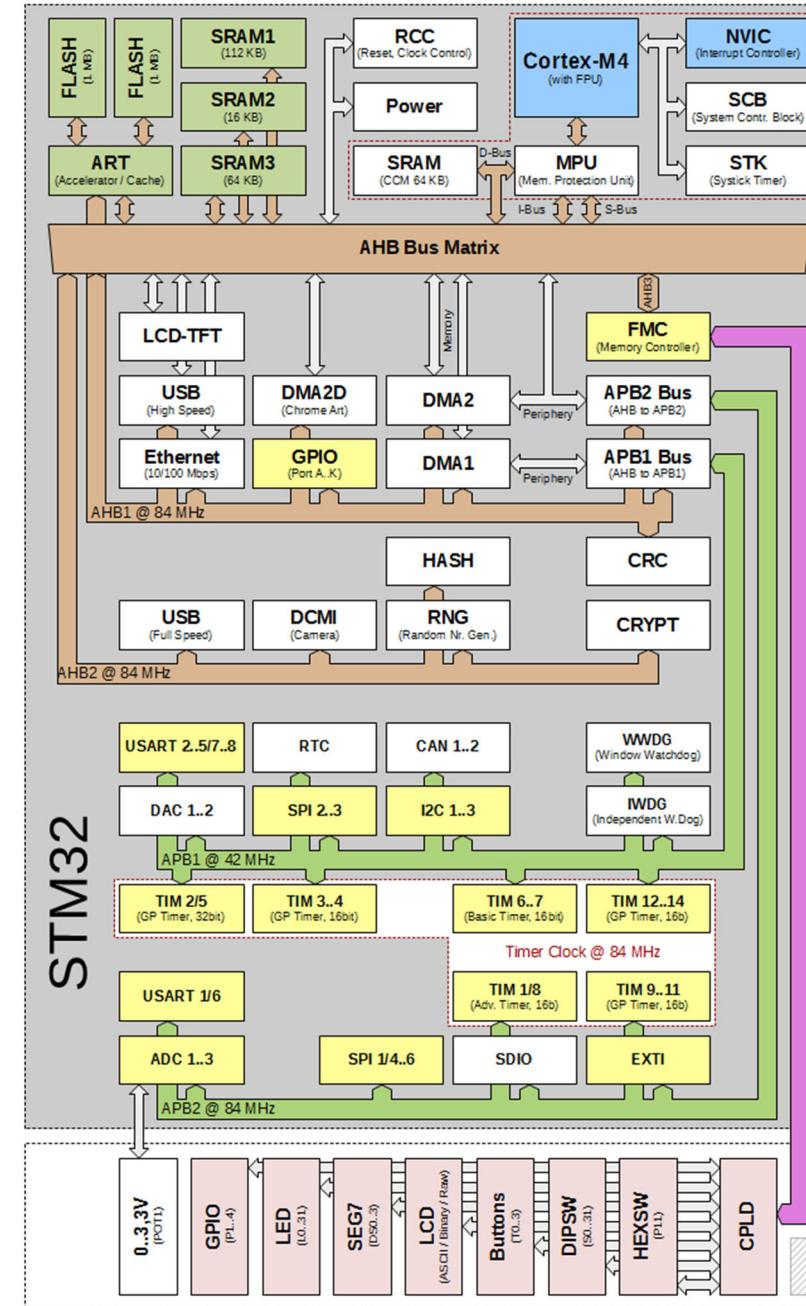
■ Simplified Model STM32F429ZI

- On-chip system bus 32 data lines, 32 address lines and control signals
- Off-chip external bus 16 data lines, 26 address lines and control signals



Bus Hierarchies

- Real-world Systems Are Partitioned into Multiple Busses



■ Hardware View

- Signals
- Timing
- Address decoding
- Wait states
- Control and status registers

■ Software View

- Accessing control and status registers in C

Accessing Control Registers in C

■ Situation

- Compiler may remove statements that have no effect from the compiler's point of view

```
uint32_t ui;

void ex_func(void)
{
    ui = 0xAAAAAAA;
    ui = 0xBBBBBBBB;
    while (ui == 0) {
        ...
    }
}
```

Optimizing compiler will
remove these statements as
they seem to have no effect

- Accesses to control registers (read/write) are memory accesses
- If writing has a side effect on the hardware and/or if reading may result in a different value than was set before in the code, the program will not behave as intended.

■ Solution

- Use qualifier volatile in variable declaration

```
volatile uint32_t ui;

void ex_func(void)
{
    ui = 0xAAAAAAA;
    ui = 0xBBBBBBBB;
    while (ui == 0) {
        ...
    }
}
```

statements will not be removed by compiler

- Tell compiler that variable may change outside the control of the compiler
 - E.g. by hardware or by an interrupt handler
- The compiler cannot make any assumption on the value
 - Needs to execute all read/write accesses as programmed
 - Prevents compiler optimizations

Accessing Control Registers in C

■ Access through Pointers

- E.g. writing to and reading from CT Board I/O

```
// a pointer called p_reg pointing to
// a volatile uint32_t
volatile uint32_t *p_reg;

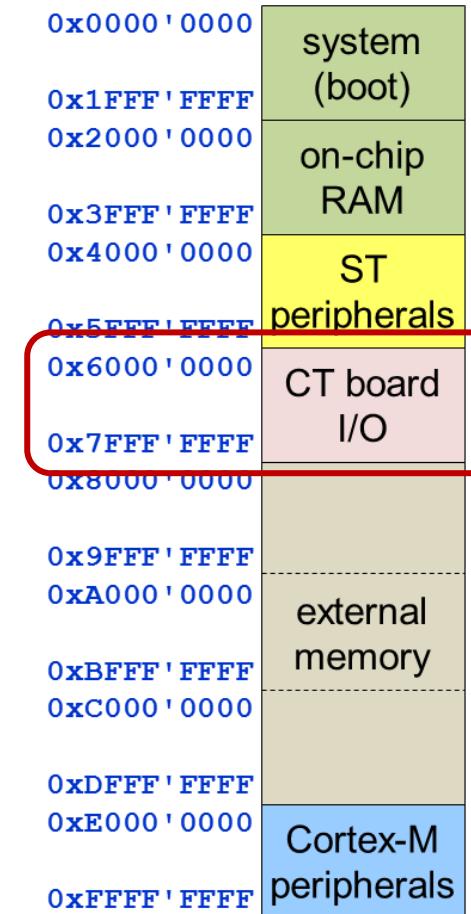
// set LEDs
p_reg = (volatile uint32_t *) (0x60000100);
*p_reg = 0xAA55AA55;
write 0xAA55 'AA55 to LEDs

// wait for dip_switches to be non-zero
p_reg = (volatile uint32_t *) (0x60000200);
while ( *p_reg == 0 ) {
}
read dip-switches
```

cast 'unsigned integer' to
'pointer to volatile uint32_t'

write 0xAA55 'AA55 to LEDs

read dip-switches



Accessing Control Registers in C

■ Using Preprocessor Macros → #define

```
dereference pointer → cast
#define LED31_0_REG      (*((volatile uint32_t *)(0x60000100)))
#define BUTTON_REG        (*((volatile uint32_t *)(0x60000210)))

// Write LED register to 0xBBCC'DDEE
LED31_0_REG = 0xBBCCDDEE;

// Read button register to aux_var
aux_var = BUTTON_REG;
```

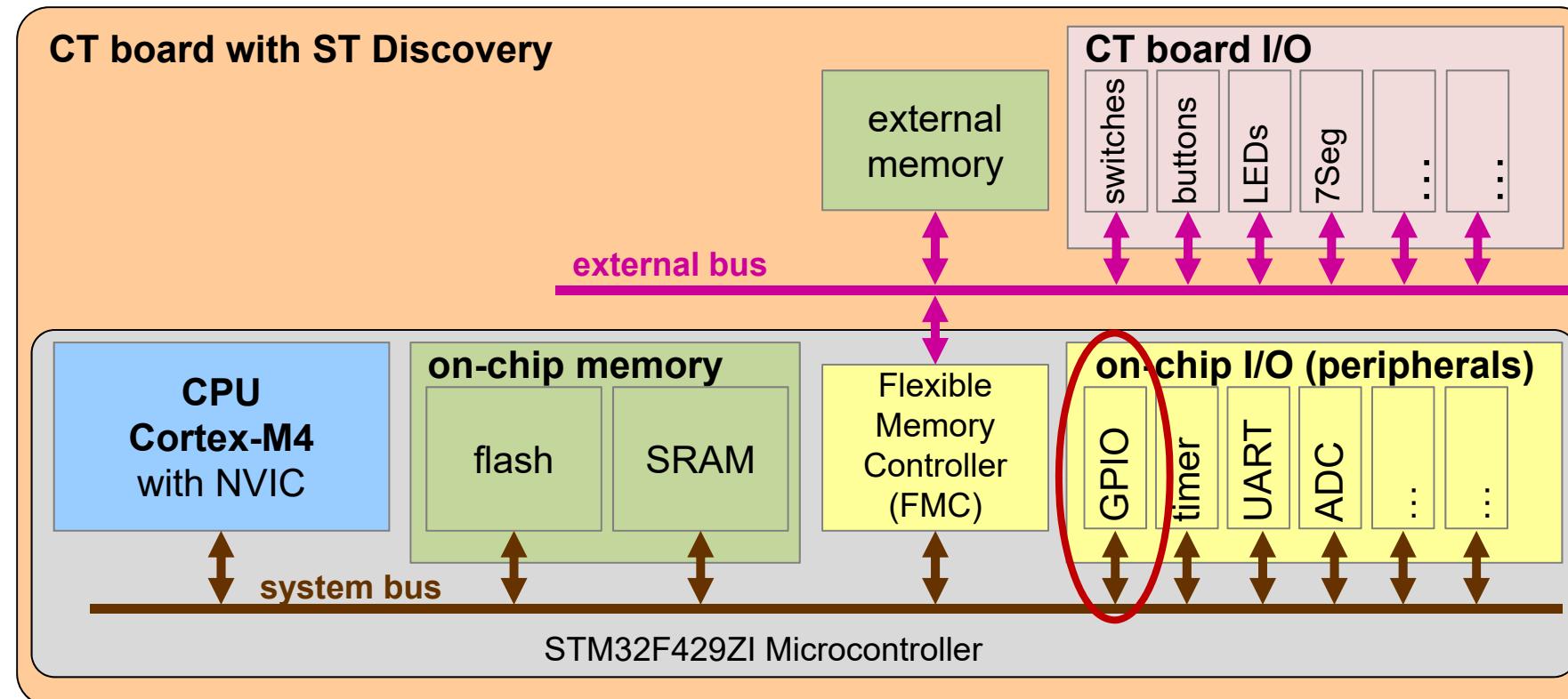
- **Microcontrollers → Embedded Systems**
 - Low cost, real time, low power, extreme environments
- **System Bus**
 - Address, data and control lines
 - Synchronous or asynchronous
 - CPU (master) reads from or writes to slave
 - Timing sequences
 - Wait states
- **Address Decoding**
 - Who is the CPU talking to?
 - Full vs. partial address decoding
- **Accessing Control Registers in C**
 - Qualifier volatile
 - Use of pointers for memory accesses

General Purpose I/O (GPIO)

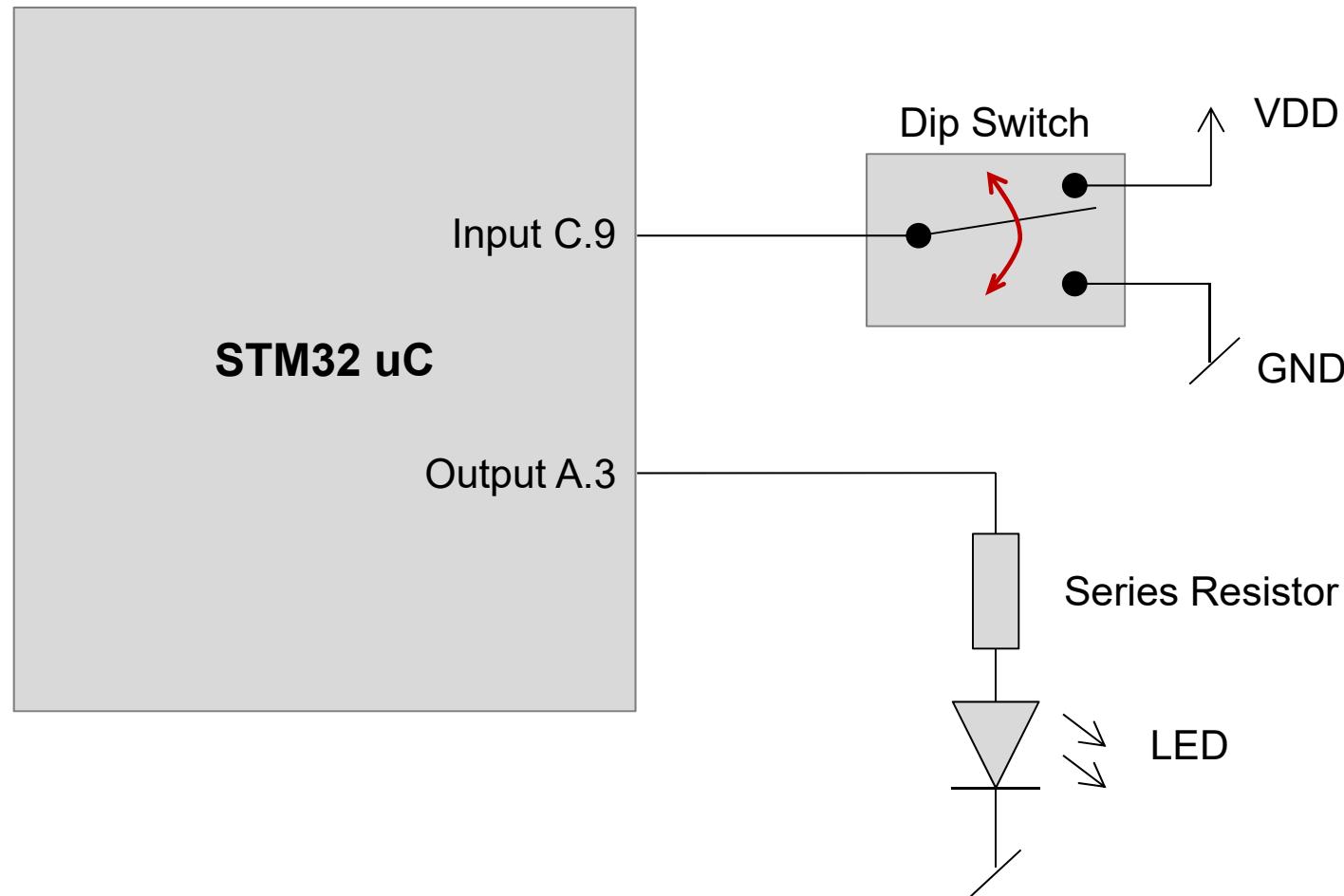
Computer Engineering 2

■ General Purpose Input / Output GPIO

- Reference Manual Pages 278 - 285



Why GPIOs?



Learning Objectives

At the end of this lesson you will be able

- to work with register descriptions in reference manuals
- to explain the concept and implementation of GPIOs
- to explain the differences between open-drain and push-pull
- to use GPIOs in your own programs
- to explain the idea of a HAL

- **Working with documents**
- **General Purpose Input / Output (GPIO)**
- **GPIO Structure**
- **Configuring**
 - Direction
 - Output Type
 - Pull-up / Pull-down
 - Speed
- **Data Registers**
 - Reading Input Data
 - Writing Output Data, Setting and Clearing Bits
- **GPIO Cookbook**
- **Hardware Abstraction Layer (HAL)**

■ F4 Datasheet

- Pin out
- Block diagram
- Etc.

■ F4 Reference Manual

- Chapter “General-purpose I/Os (GPIO)”

RM0090
Reference manual

STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and
STM32F43xxx advanced ARM-based 32-bit MCUs

Introduction

This reference manual targets application developers. It provides complete information on how to use the STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx microcontroller memory and peripherals.

The STM32F405xx/07xx, STM32F415xx/17xx, STM32F42xxx and STM32F43xxx constitute a family of microcontrollers with different memory sizes, packages and peripherals.

For ordering information, mechanical and electrical device characteristics please refer to the datasheets.

For information on the ARM Cortex™-M4 with FPU core, please refer to the *Cortex™-M4 with FPU Technical Reference Manual*.

Related documents

Available from STMicroelectronics web site (<http://www.st.com>):

- STM32F40x and STM32F41x datasheets
- STM32F42x and STM32F43x datasheets
- For information on the ARM Cortex™-M4 core with FPU, refer to the *STM32F3xx/F4xxx Cortex™-M4 programming manual (PM0214)*.

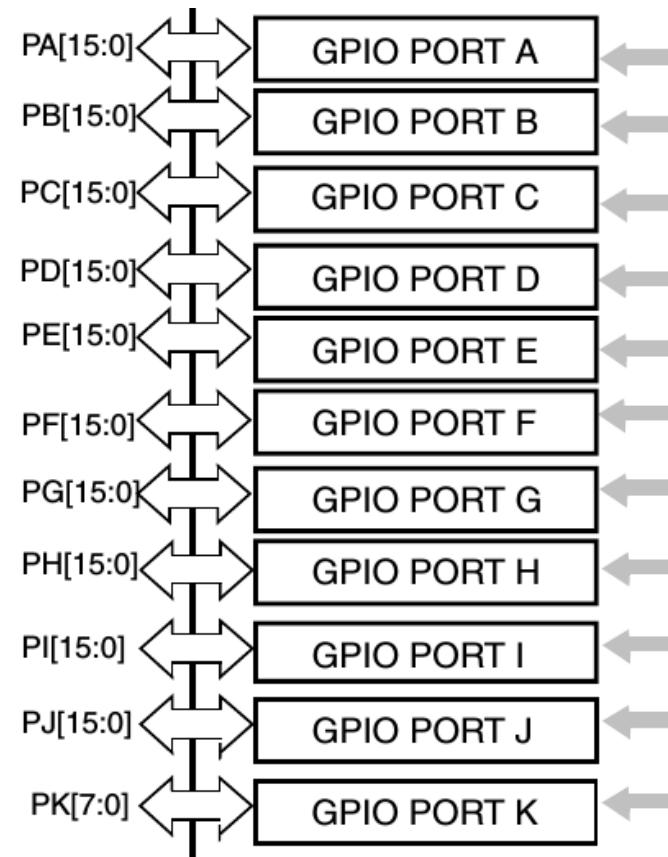
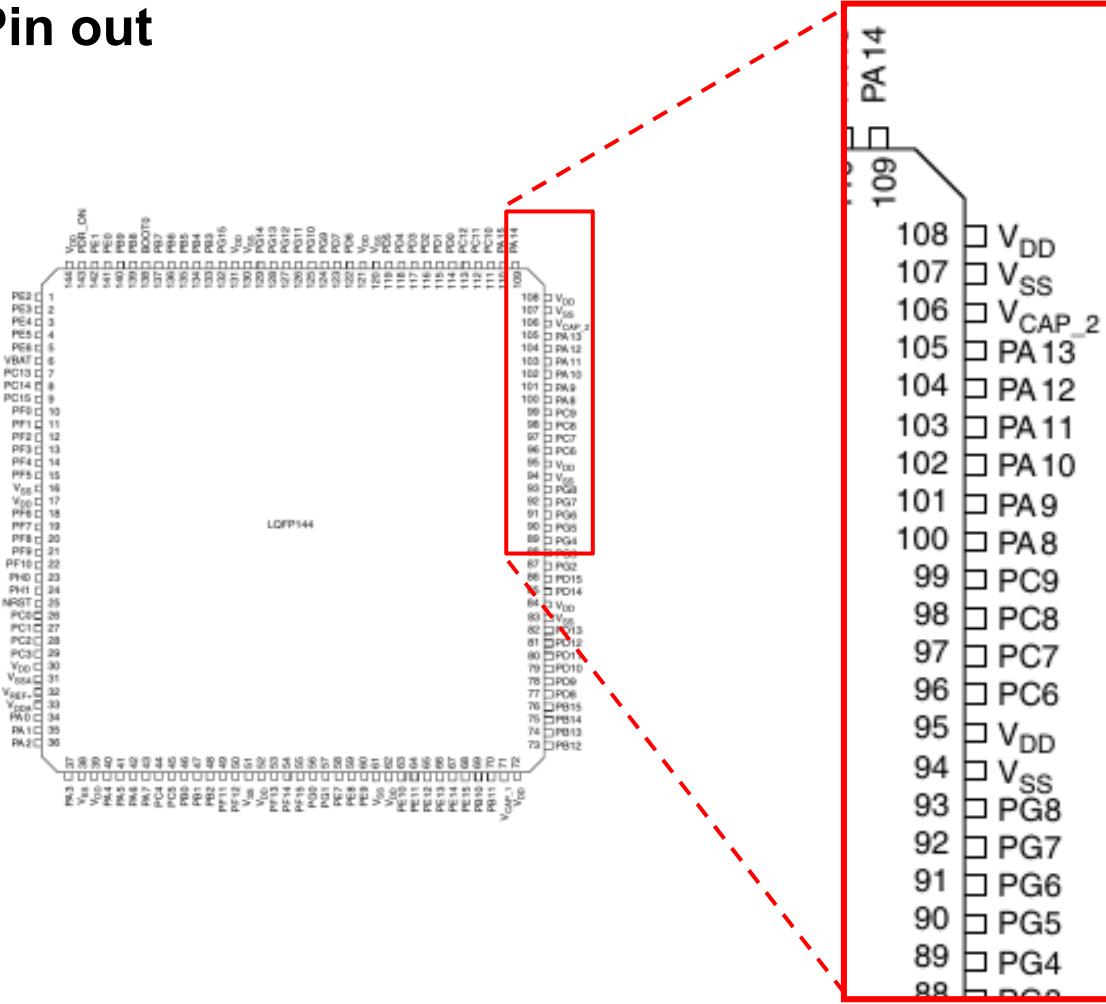
Table 1. Applicable products

Product family	Part numbers and product categories
Microcontrollers	STM32F405xx, STM32F407xx, STM32F415xx, STM32F417xx, STM32F427xx, STM32F437xx, STM32F429xx and STM32F439xx.

September 2013 Doc ID 018909 Rev 5 1/1705 www.st.com

Working with Documents

■ Pin out



Sources: *STM F4 Reference Manual*
STM F4 Data Sheet

Note: The reference manual of ST shows memory maps with the low address at the bottom.

■ Register address = Base address + Offset

- Offset is given for each register in reference manual
- Base address defined in memory map
→ Reference Manual

base address

Boundary address	Peripheral
0x4002 2800 - 0x4002 2BFF	GPIOK
0x4002 2400 - 0x4002 27FF	GPIOJ
0x4002 2000 - 0x4002 23FF	GPIOI
0x4002 1C00 - 0x4002 1FFF	GPIOH
0x4002 1800 - 0x4002 1BFF	GPIOG
0x4002 1400 - 0x4002 17FF	GPIOF
0x4002 1000 - 0x4002 13FF	GPIOE
0x4002 0C00 - 0x4002 0FFF	GPIOD
0x4002 0800 - 0x4002 0BFF	GPIOC
0x4002 0400 - 0x4002 07FF	GPIOB
0x4002 0000 - 0x4002 03FF	GPIOA

Boundary address	Peripheral	Bus	Register map
0xA000 0000 - 0xA000 0FFF	FSMC control register (STM32F405xx/07xx and STM32F415xx/17xx)/ FMC control register (STM32F42xxx and STM32F43xxx)	AHB3	Section 36.6.9: FSMC register map on page 1573 Section 37.8: FMC register map on page 1653
0x5006 0800 - 0x5006 0BFF	RNG		Section 24.4.4: RNG register map on page 752
0x5006 0400 - 0x5006 07FF	HASH		Section 25.4.9: HASH register map on page 776
0x5006 0000 - 0x5006 03FF	CRYP		Section 23.6.13: CRYP register map on page 745
0x5005 0000 - 0x5005 03FF	DCMI		Section 15.8.12: DCMI register map on page 473
0x5000 0000 - 0x5003 FFFF	USB OTG FS		Section 34.16.6: OTG_FS register map on page 1303
0x4004 0000 - 0x4007 FFFF	USB OTG HS		Section 35.12.6: OTG_HS register map on page 1445
0x4002 8000 - 0x4002 BBFF	DMA2D		Section 11.5: DMA2D registers on page 349
0x4002 9000 - 0x4002 93FF			
0x4002 8C00 - 0x4002 8FFF			
0x4002 8800 - 0x4002 8BFF			
0x4002 8400 - 0x4002 87FF			
0x4002 8000 - 0x4002 83FF			
0x4002 6400 - 0x4002 67FF	DMA2		Section 10.5.11: DMA register map on page 332
0x4002 6000 - 0x4002 63FF	DMA1		
0x4002 4000 - 0x4002 4FFF	BKPSRAM		
0x4002 3C00 - 0x4002 3FFF	Flash interface register		Section 3.5: Flash interface registers
0x4002 3800 - 0x4002 3BFF	RCC		Section 7.3.25: RCC register map on page 263
0x4002 3000 - 0x4002 33FF	CRC		Section 4.4.4: CRC register map on page 114
0x4002 2800 - 0x4002 2BFF	GPIOK		Section 8.4.11: GPIO register map on page 284
0x4002 2400 - 0x4002 27FF	GPIOJ		
0x4002 2000 - 0x4002 23FF	GPIOI		
0x4002 1C00 - 0x4002 1FFF	GPIOH		
0x4002 1800 - 0x4002 1BFF	GPIOG		
0x4002 1400 - 0x4002 17FF	GPIOF		
0x4002 1000 - 0x4002 13FF	GPIOE		
0x4002 0C00 - 0x4002 0FFF	GPIOD		
0x4002 0800 - 0x4002 0BFF	GPIOC		
0x4002 0400 - 0x4002 07FF	GPIOB		
0x4002 0000 - 0x4002 03FF	GPIOA		
0x4001 6800 - 0x4001 6BFF	LCD-TFT		Section 16.7.26: LTDC register map on page 504
0x4001 5800 - 0x4001 5BFF	SAI1		Section 25.17.9: SAI register map on page 944
0x4001 5400 - 0x4001 57FF	SPI6		
0x4001 5000 - 0x4001 53FF	SPI5		Section 28.5.10: SPI register map on page 906
0x4001 4800 - 0x4001 4BFF	TIM11		
0x4001 4400 - 0x4001 47FF	TIM10		Section 19.5.11: TIM10/11/13/14 register map on page 676

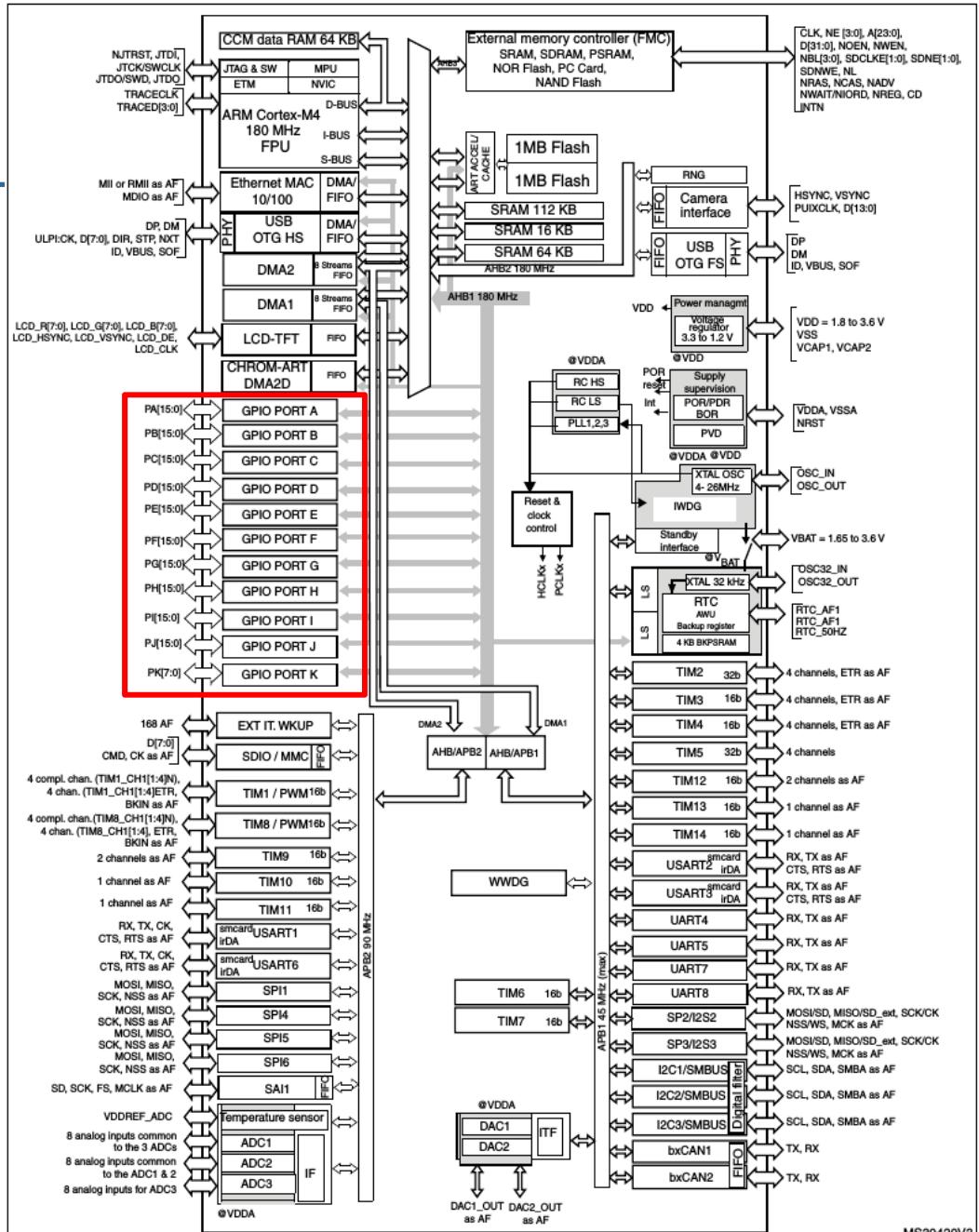
■ Exercise

Use the data sheet and reference manual to answer the following questions for an STM32F429 LQFP144

- What are the pin numbers (0-144) for GPIO ports A.3 and E.1?
- What is the address of register GPIOA_OTYPER?

GPIO

General Purpose Input / Output



General Purpose Input / Output

■ Situation

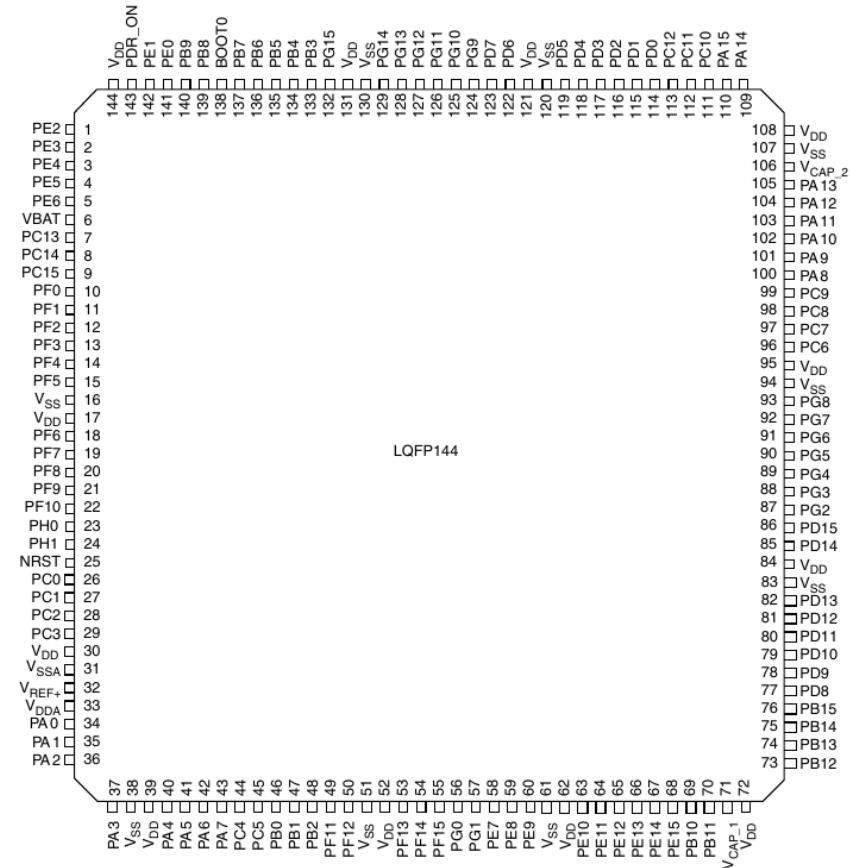
- Microcontroller as general purpose device
 - Many functional blocks included

■ Problem

- Limited number of pins
 - For a specific configuration, not all functions can be routed to I/O pins

■ Solution

- Many (all) pins configurable
 - Select the needed I/O pins / functions
 - „pin sharing“
 - Output multiplexer needs to be configured



Pinout STM32F429 LQFP144
Source: STM F4 Reference Manual

■ **Multiple functions „share“ a single pin (pin sharing)**

- Digital inputs / outputs (GPIO)
- Serial interfaces
- Timers / Counters
- ADC (A/D conversion)

■ **Consequences**

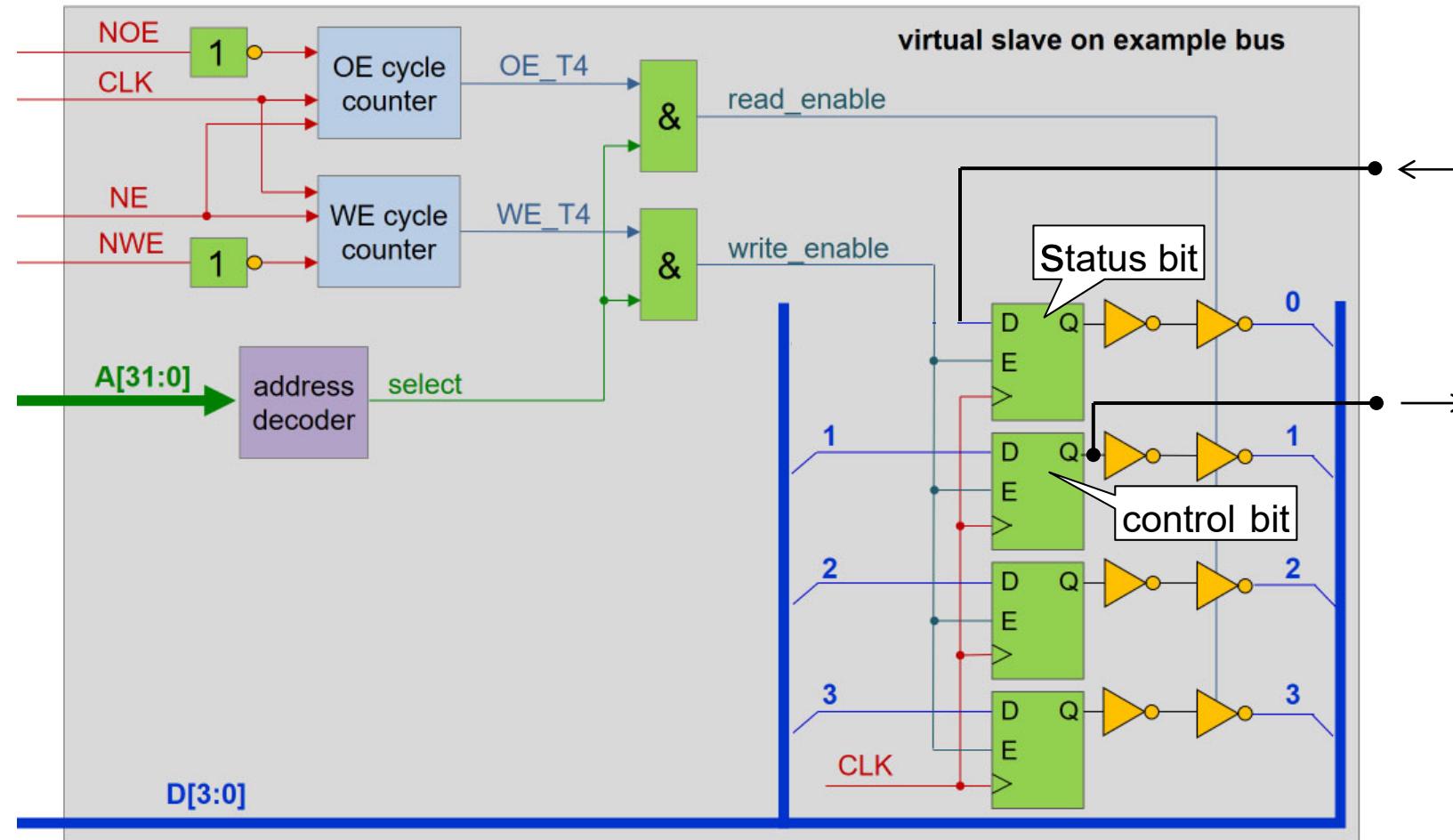
- Not all functions externally available at the same time
- Programming of internal registers defines pin use
- Pin / function configuration is usually static, i.e. set once at startup

→ **Not all combinations possible simultaneously**

■ Features

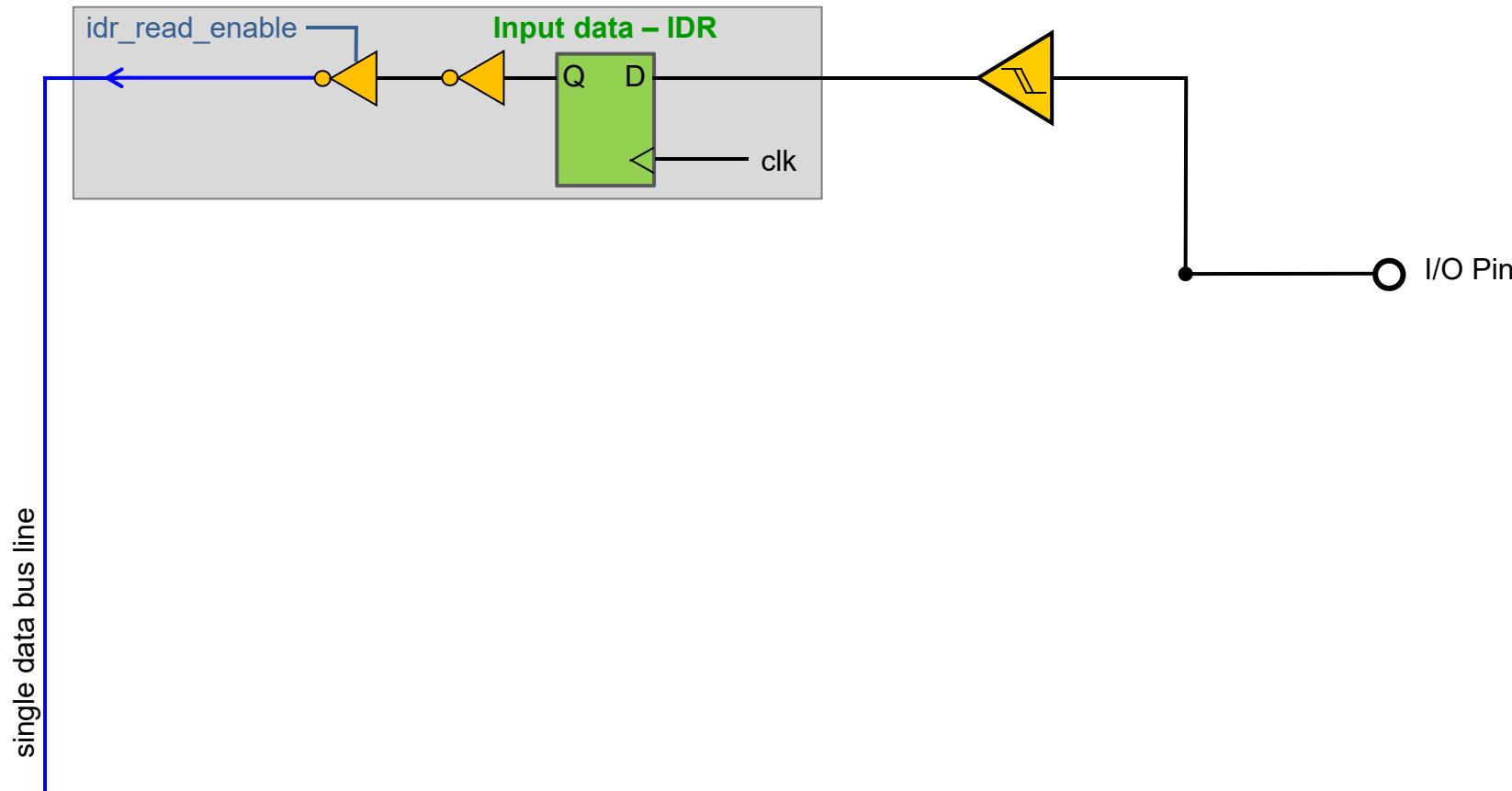
- GPIO pins configurable by software
 - output (push-pull or open-drain; with or without pull-up or pull-down)
 - input (floating, with or without pull-up or pull-down)
 - peripheral alternate function
- High-current-capable
- Speed selection
- Maximum I/O toggling up to 90 MHz
- Most of the GPIO pins are shared with alternate digital or analog functions

- Repetition: Hardware slave on synchronous bus



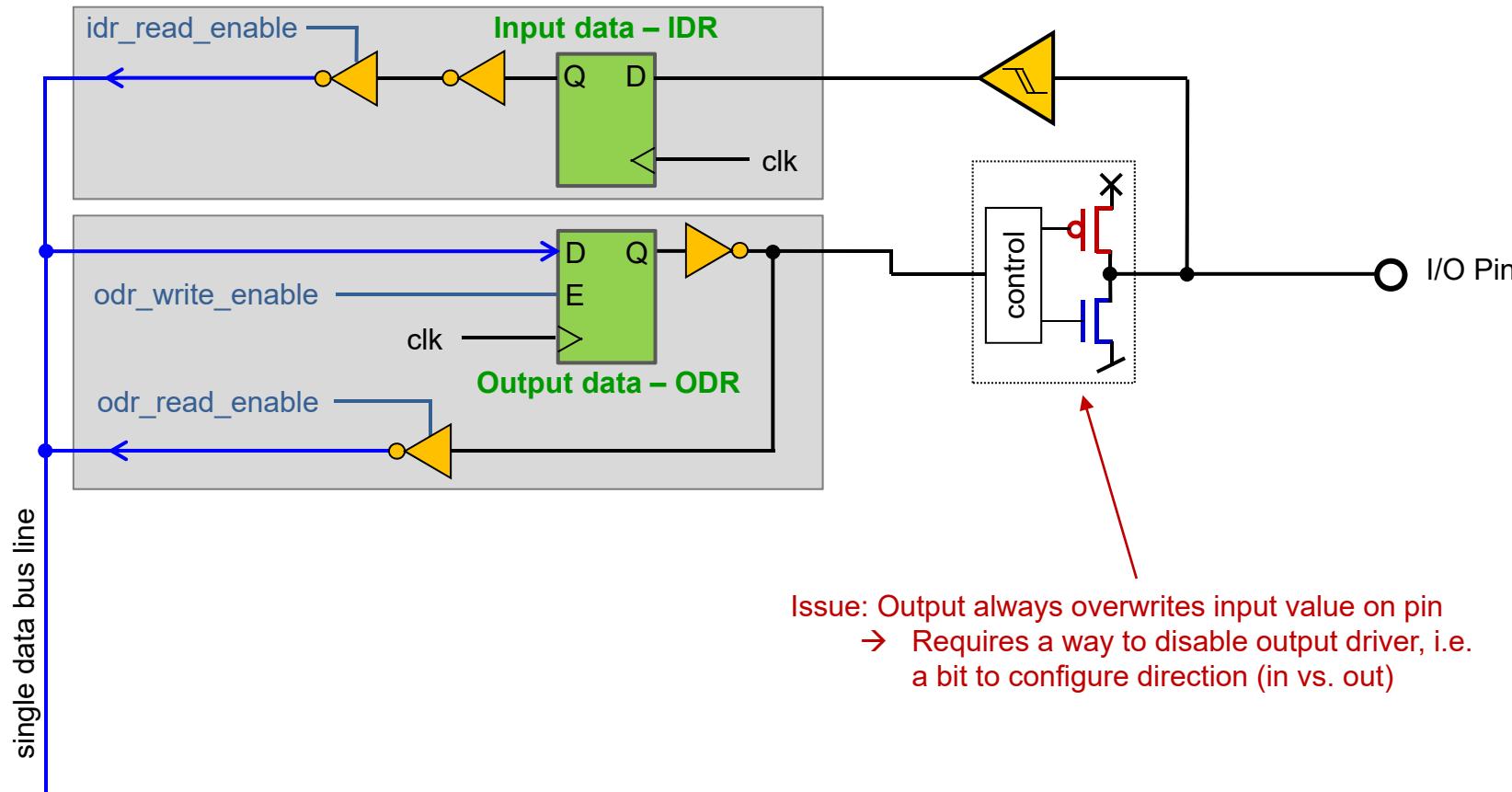
■ Input

Address decoding: Register accesses
activate the corresponding enable signals



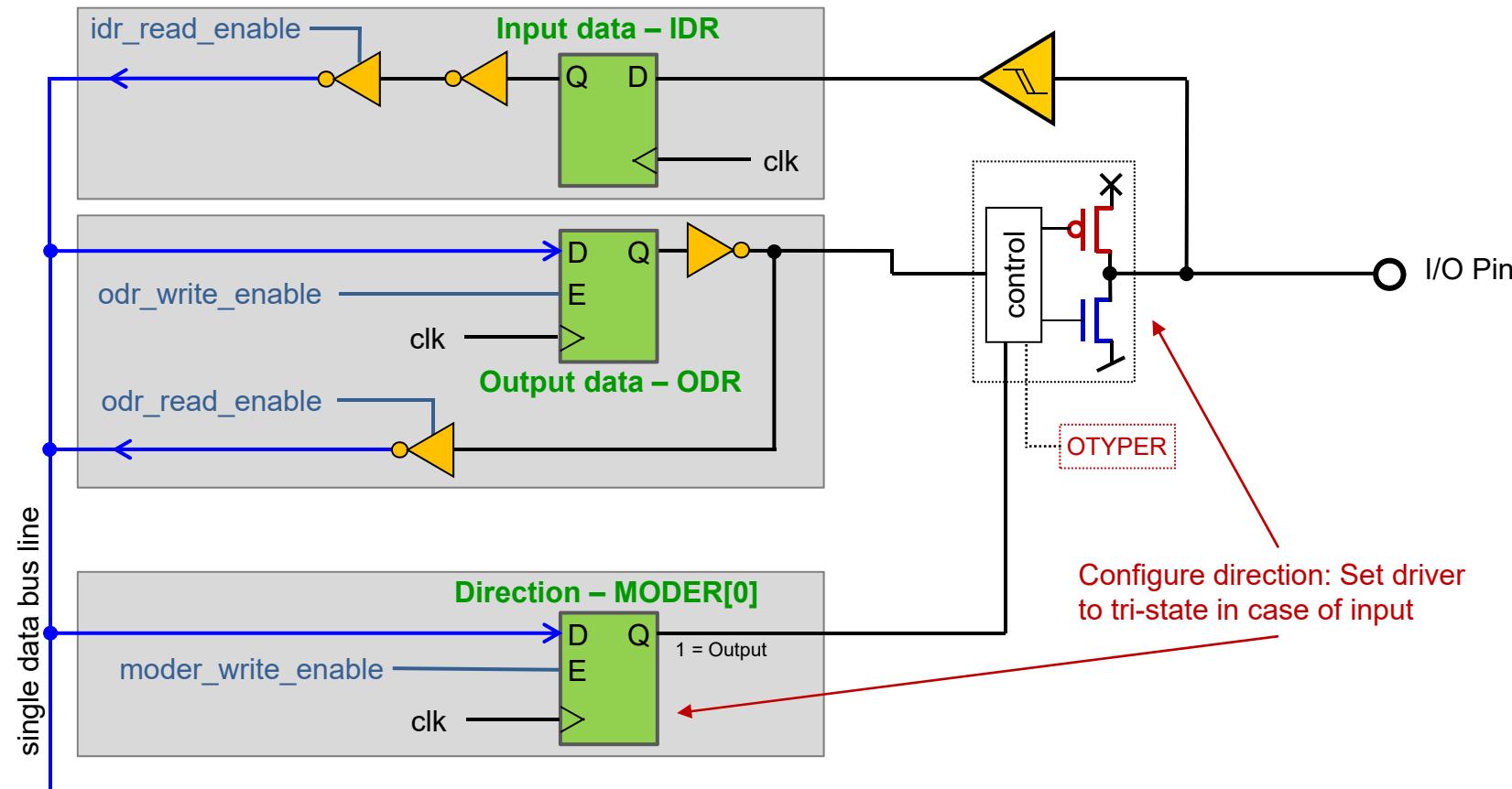
■ Adding output

Address decoding: Register accesses activate the corresponding enable signals

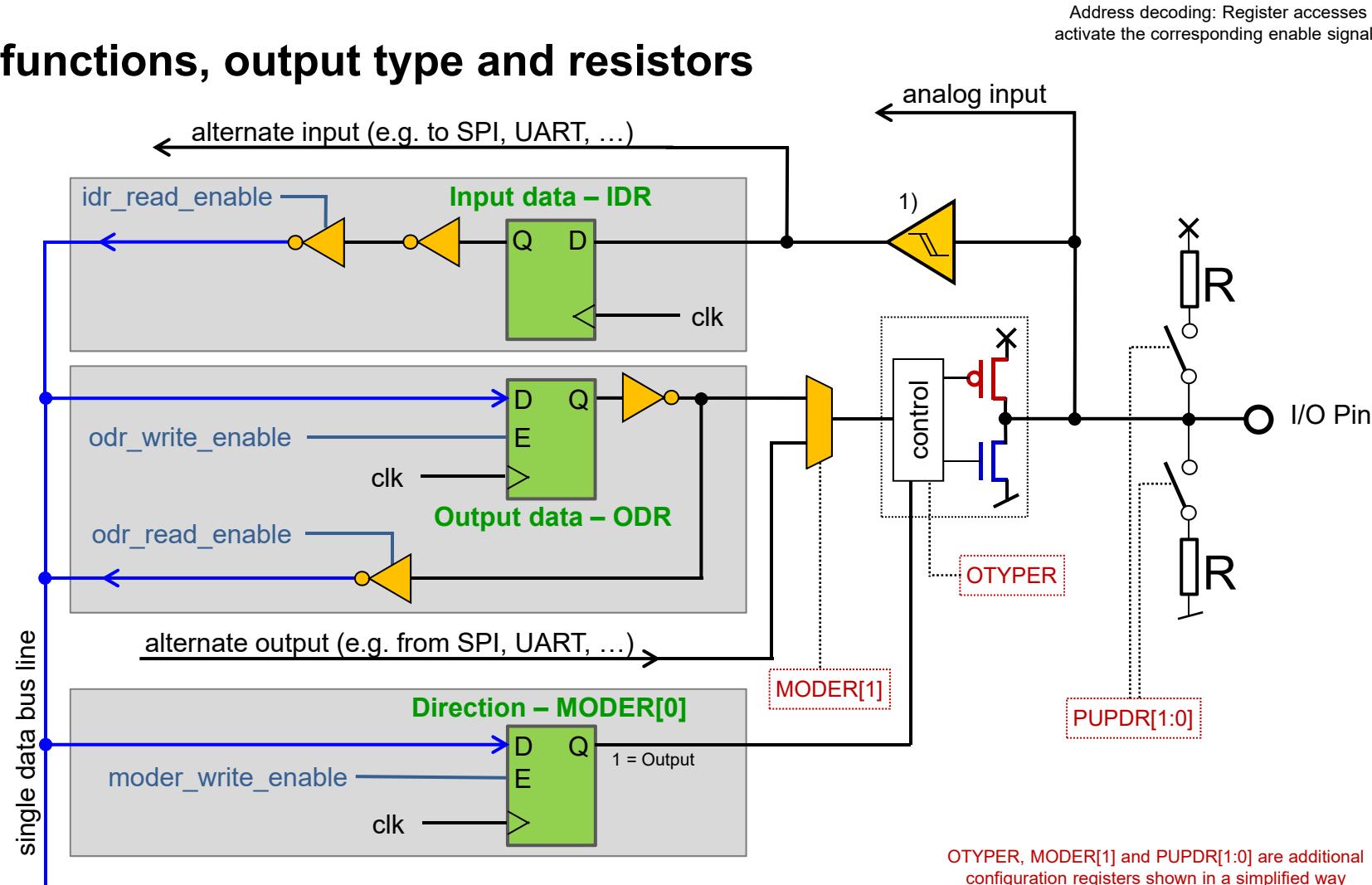


■ Choosing between input and output

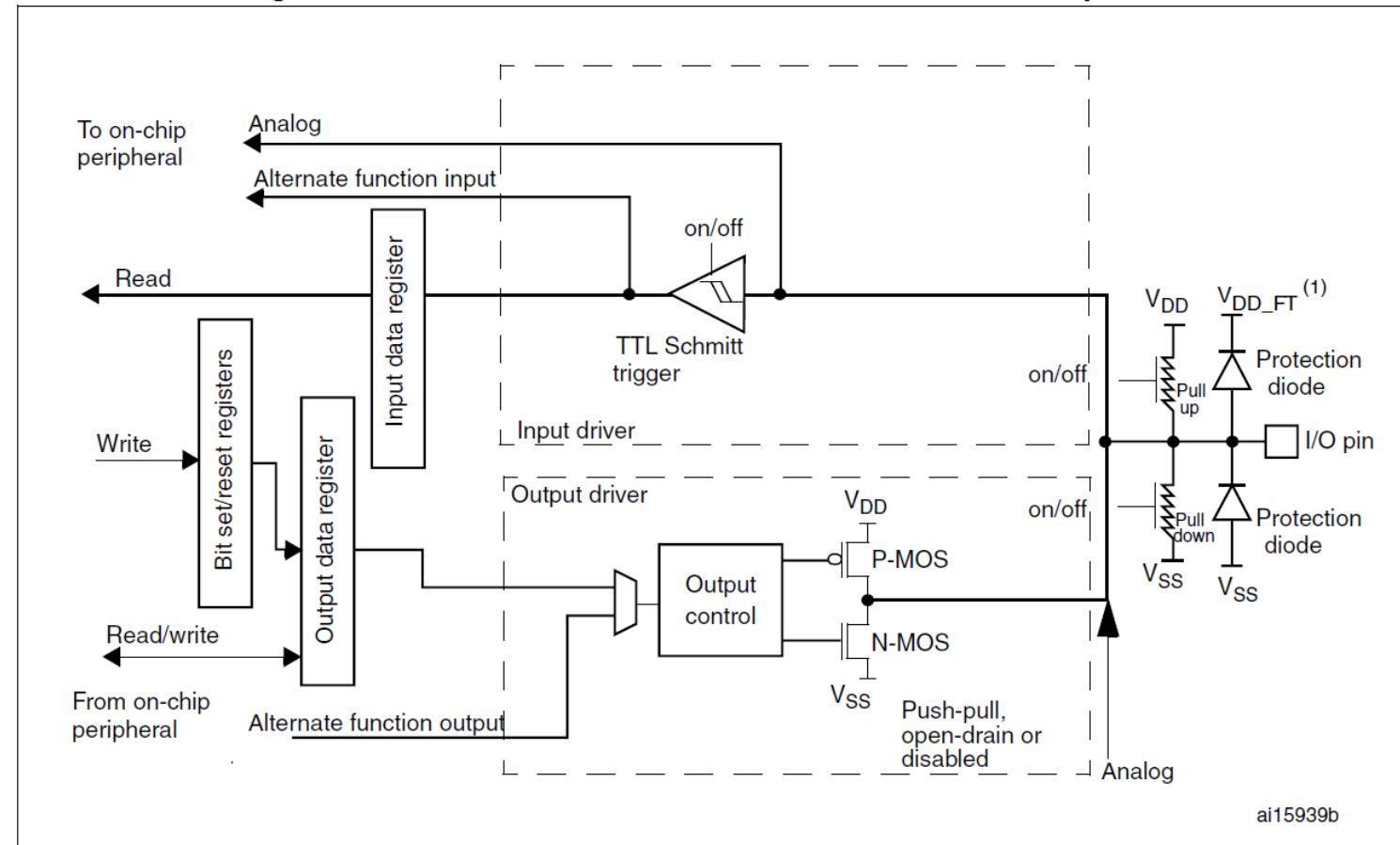
Address decoding: Register accesses activate the corresponding enable signals



■ Alternate functions, output type and resistors



■ GPIO as shown in ST Reference Manual



ai15939b

Source: ST F4 Reference Manual

Configuring Direction

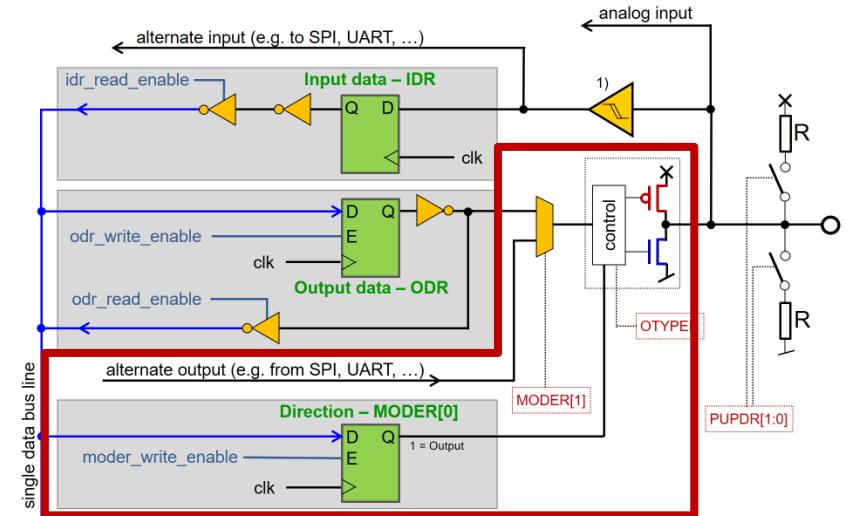
■ Mode register (**GPIOx_MODER**)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]	MODER14[1:0]	MODER13[1:0]	MODER12[1:0]	MODER11[1:0]	MODER10[1:0]	MODER9[1:0]	MODER8[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]	MODER6[1:0]	MODER5[1:0]	MODER4[1:0]	MODER3[1:0]	MODER2[1:0]	MODER1[1:0]	MODER0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- MODER[1:0]
 - 00: Input
 - 01: General purpose output mode
 - 10: Alternate function mode
 - 11: Analog mode

In case of alternate function:

The individual alternate function (SPI, UART, etc.) can be selected by programming GPIOx_AFRL/H (i.e. Alternate function register low / high)



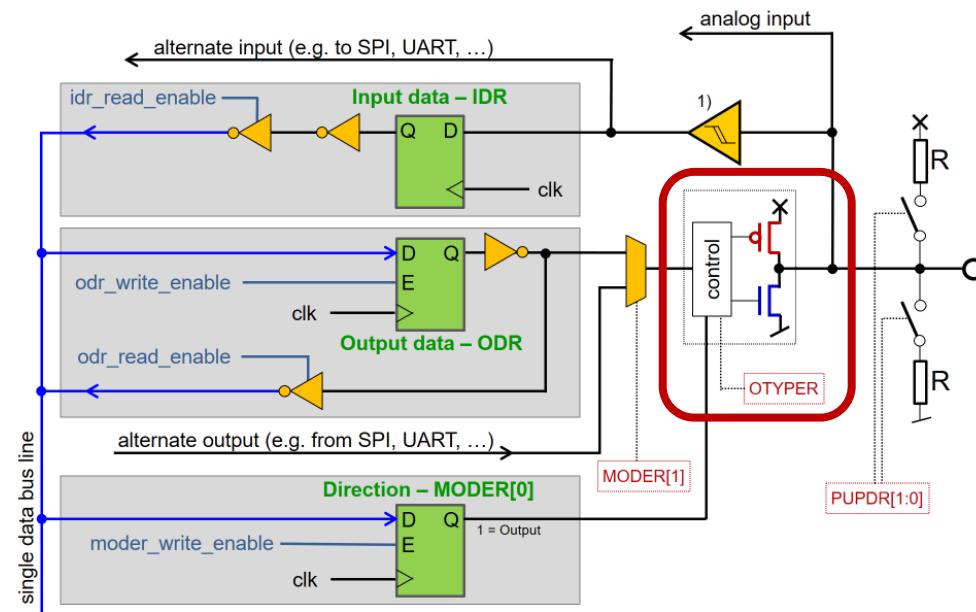
See reference manual

Configuring Output Type

■ Output type register (**GPIOx_OTYPER**)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- OT
 - 0: Output push-pull
 - 1: Output open-drain

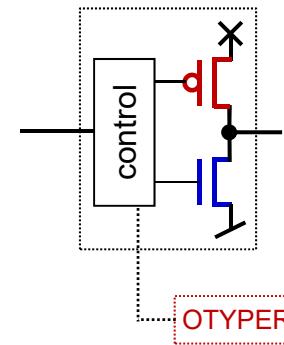


See reference manual

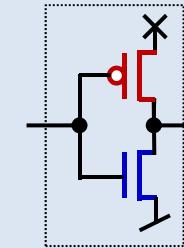
Output Type

■ Push-pull vs. Open-drain

- Case output i.e. MODER[0] = '1'

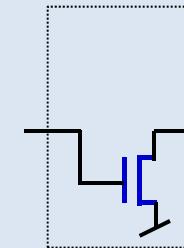


OT = '0' → push-pull



Output stage can
drive output
'high' or 'low'

OT = '1' → open-drain



Output stage can
only drive output
'low'

Output Type

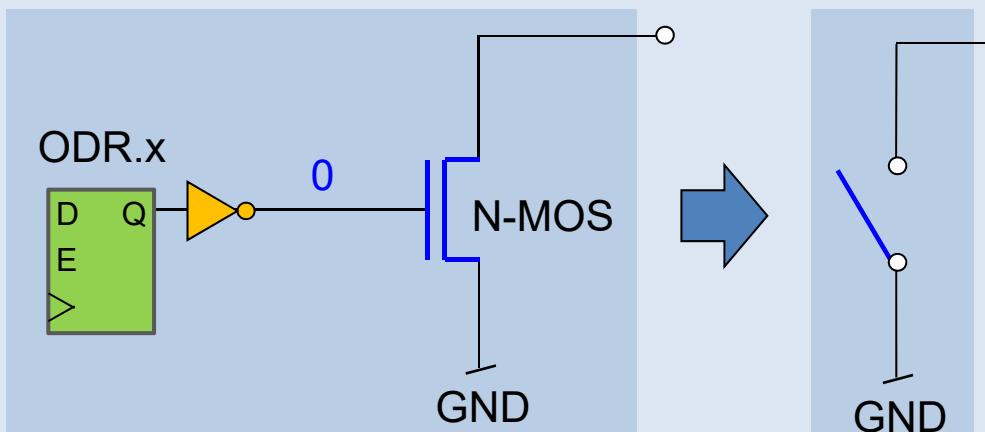
■ Open-drain

- Pull-down transistor only → no pull-up transistor

ODR.x = '1'

- R_{DS} is high
- Transistor is blocking → switch is open
- Output high impedance, floating

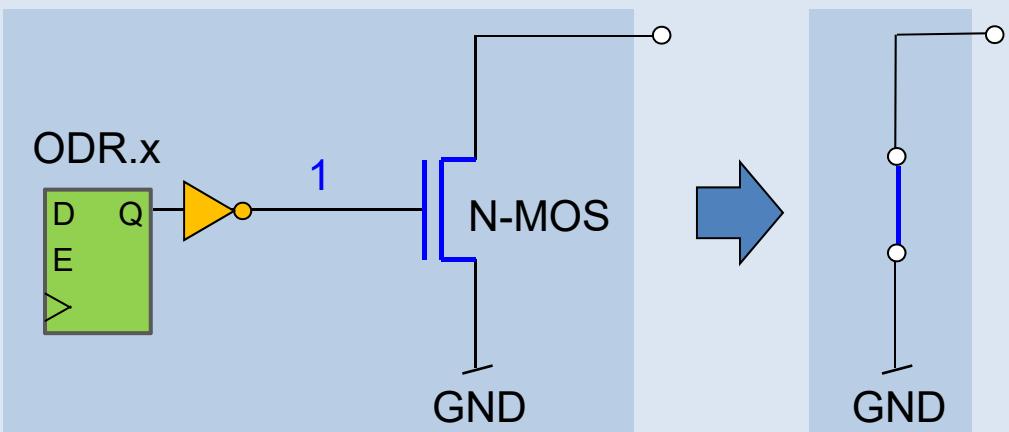
*high impedance,
floating*



ODR.x = '0'

- R_{DS} is low
- Transistor is conducting → switch closed
- Output pulled to GND

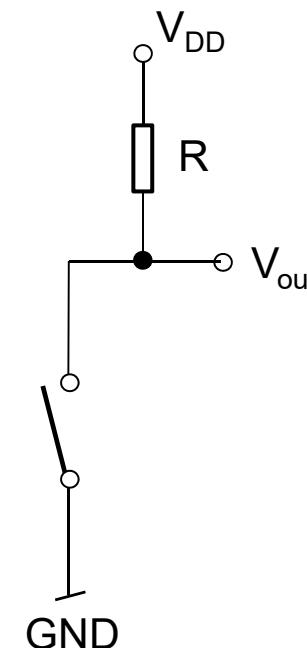
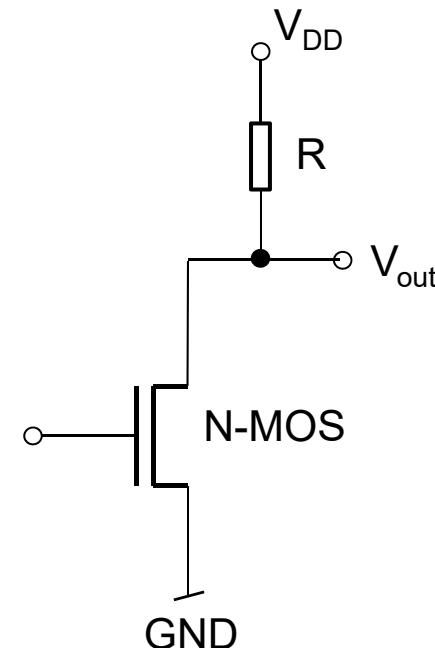
$\approx \text{GND}$



■ Open-drain with pull-up resistor

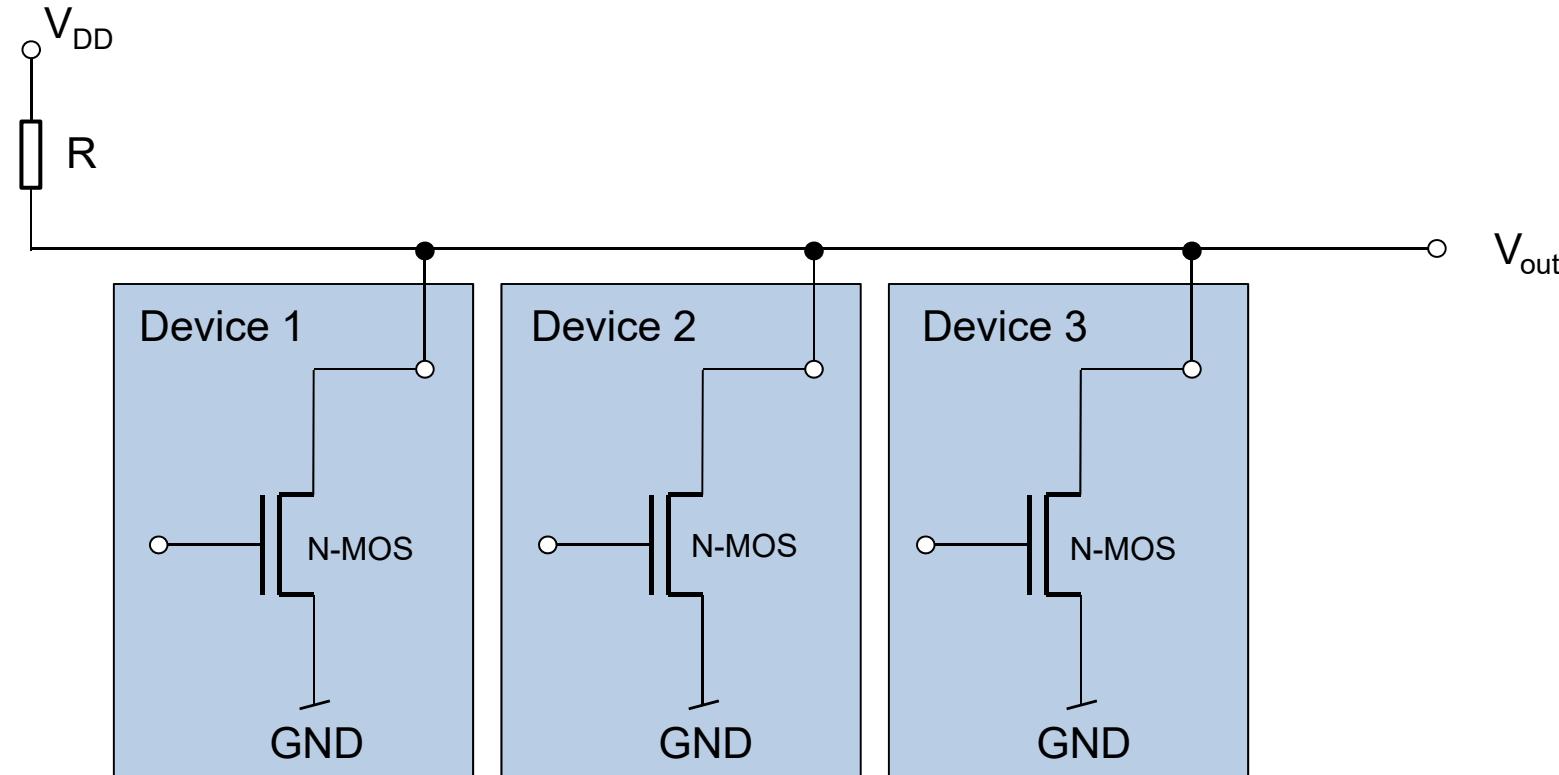
- Transistor blocking (= switch open)
- Transistor conducting (= switch closed)

→ V_{out} is pulled up to level of VDD
→ V_{out} goes to GND



■ Multiple open-drain outputs on a bus line

- No electrical signal conflicts possible
- Any device can pull signal low at any time

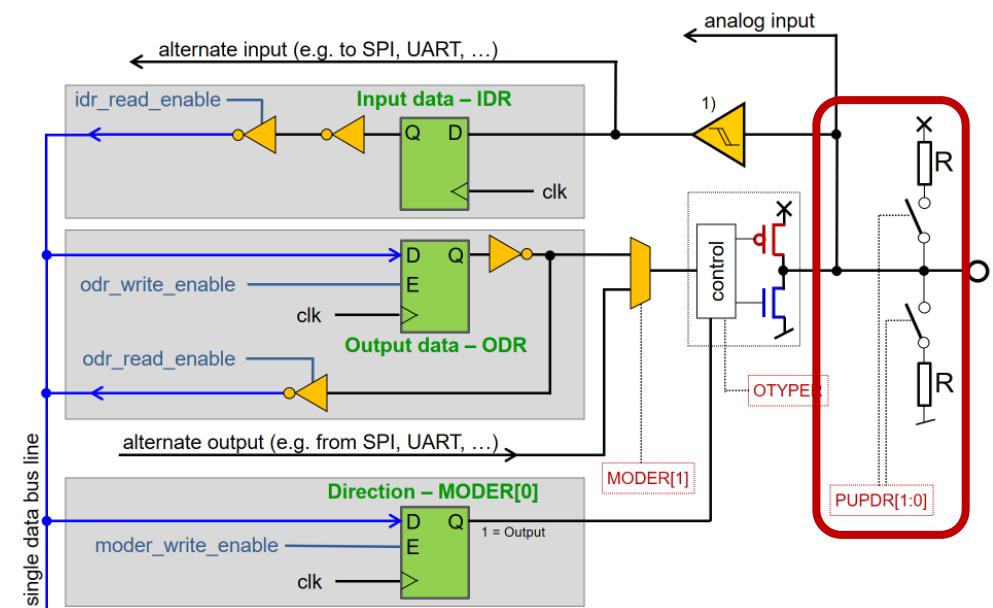


Configuring Pull-up / Pull-down

■ Pull-up/pull-down register (**GPIOx_PUPDR**)

PUPDR15[1:0]	PUPDR14[1:0]	PUPDR13[1:0]	PUPDR12[1:0]	PUPDR11[1:0]	PUPDR10[1:0]	PUPDR9[1:0]	PUPDR8[1:0]
rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8
PUPDR7[1:0]	PUPDR6[1:0]	PUPDR5[1:0]	PUPDR4[1:0]	PUPDR3[1:0]	PUPDR2[1:0]	PUPDR1[1:0]	PUPDR0[1:0]
rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8

- PUPDR[1:0]
 - 00: No pull-up, no pull-down
 - 01: Pull-up
 - 10: Pull-down
 - 11: Reserved



See reference manual

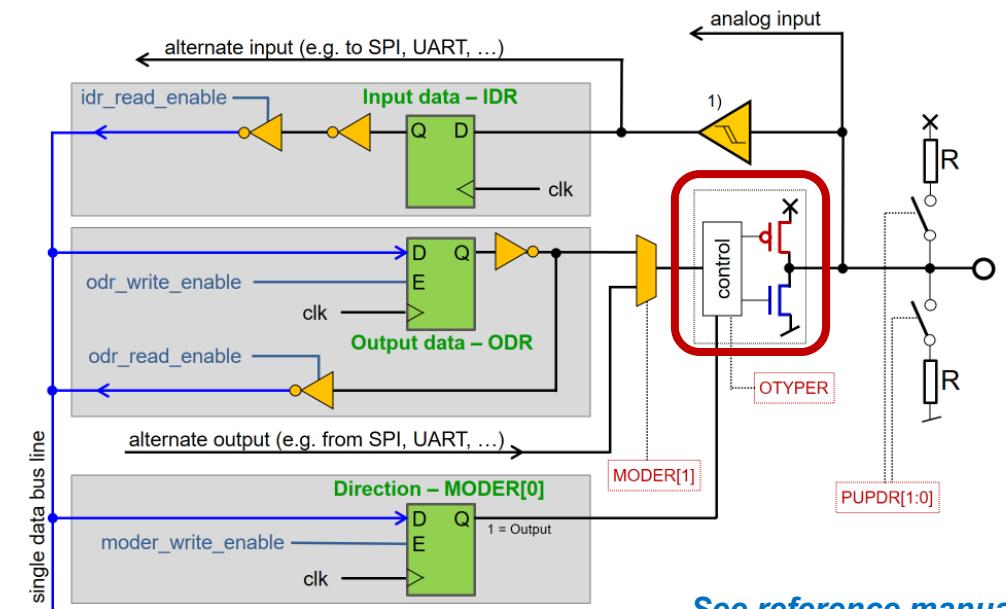
Configuring Speed

■ Output speed register (**GPIOx_OSPEEDR**)

OSPEEDR15 [1:0]	OSPEEDR14 [1:0]	OSPEEDR13 [1:0]	OSPEEDR12 [1:0]	OSPEEDR11 [1:0]	OSPEEDR10 [1:0]	OSPEEDR9 [1:0]	OSPEEDR8 [1:0]
rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8
OSPEEDR7[1:0]	OSPEEDR6[1:0]	OSPEEDR5[1:0]	OSPEEDR4[1:0]	OSPEEDR3[1:0]	OSPEEDR2[1:0]	OSPEEDR1 [1:0]	OSPEEDR0 1:0]
rw	rw	rw	rw	rw	rw	rw	rw
1	0						

- OSPEEDR[1:0]
 - 00: Low speed
 - 01: Medium speed
 - 10: Fast speed
 - 11: High speed

Motivation: Match output stage to impedance of transmission line.
Control steepness of edge → e.g for EMC reasons



See reference manual

I/O Port Configuration

■ Overview

GP = general-purpose

PP = push-pull

PU = pull-up

PD = pull-down

OD = open-drain

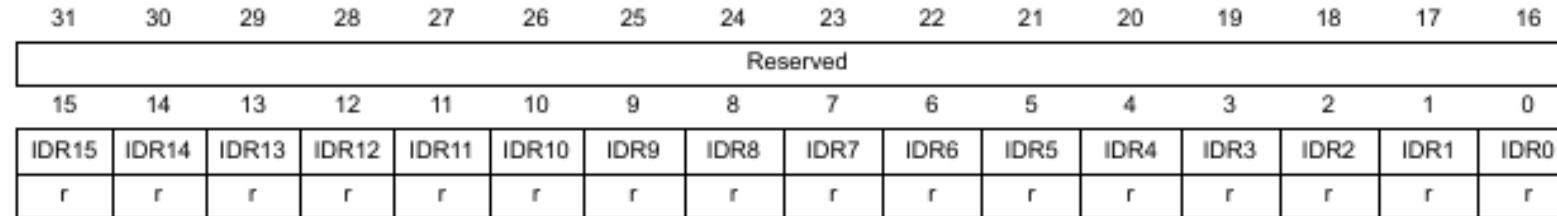
AF = alternate function

MODER(i) [1:0]	OTYPER(i)	OSPEEDR(i) [B:A]	PUPDR(i) [1:0]	I/O configuration	
01	0	SPEED [B:A]	0	0	GP output PP
	0		0	1	GP output PP + PU
	0		1	0	GP output PP + PD
	0		1	1	Reserved
	1		0	0	GP output OD
	1		0	1	GP output OD + PU
	1		1	0	GP output OD + PD
	1		1	1	Reserved (GP output OD)
	0		0	0	AF PP
10	0	SPEED [B:A]	0	1	AF PP + PU
	0		1	0	AF PP + PD
	0		1	1	Reserved
	1		0	0	AF OD
	1		0	1	AF OD + PU
	1		1	0	AF OD + PD
	1		1	1	Reserved
	x	00	x	x	Input Floating
	x		x	x	Input PU
	x		x	x	Input PD
	x		x	x	Reserved (input floating)
00	x	11	x	x	Input/output Analog
	x		x	x	0
	x		x	x	1
	x		x	x	0
11	x		x	x	1
	x		x	x	1

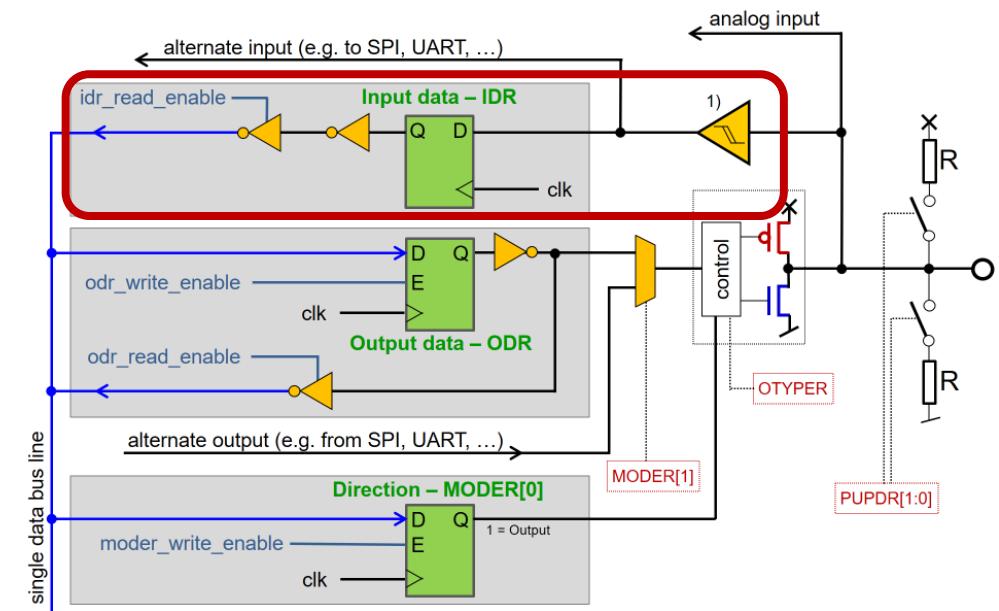
Source: ST F4 Reference Manual

Reading Input Data

■ GPIO port input data register (**GPIOx_IDR**)



- IDR: Port input data
 - contain input value of corresponding I/O port
 - read-only



See reference manual

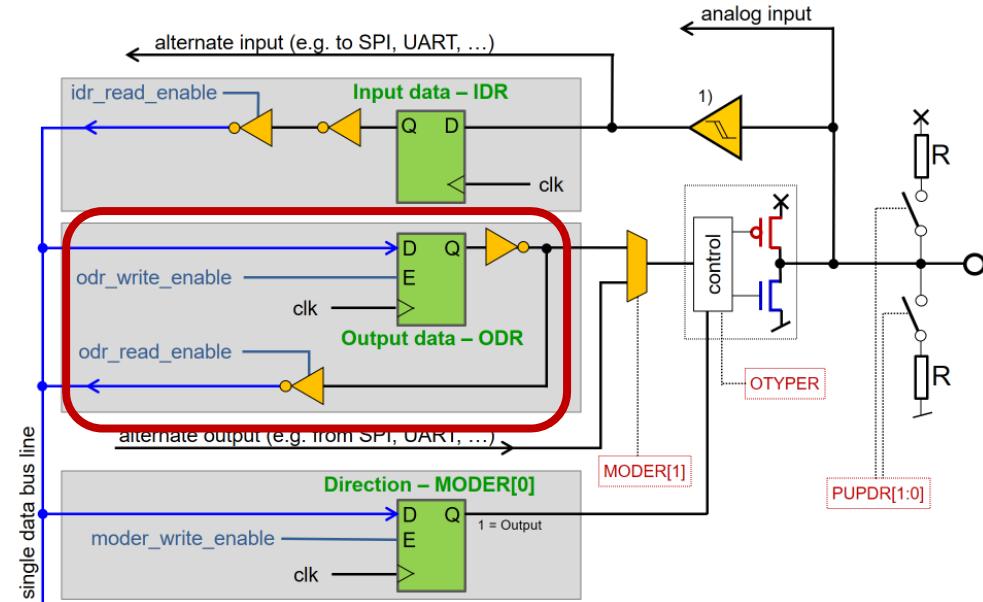
Writing Output Data

■ GPIO port output data register (**GPIOx_ODR**)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0

rw rw

- ODR: Port output data
 - read and write by software



See reference manual

■ GPIO port bit set/reset register (**GPIOx_BSRR**)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Set bits →	BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

- Clear port bit by writing a ‘1’ to BSRR[bit+16]
- Set port bit by writing a ‘1’ to BSRR[bit]
- Ensures atomic access in software (no interruption possible)
 - Setting a bit through ODR requires ‘read ODR’, ‘OR operation with bit mask’ and ‘write ODR’

[See reference manual](#)

GPIO Register Map

Offset	Register	Reset value
0x00	GPIOA_MODER	
0x00	GPIOB_MODER	
0x00	GPIOx_MODER (where x = C..IJK)	
0x04	GPIOx_OTYPER (where x = A..IJK) Reset value	
0x08	GPIOx_OSPEEDER (where x = A..IJK except B) Reset value	
0x08	GPIOB_OSPEEDER	
0x0C	GPIOA_PUPDR	
0x0C	GPIOB_PUPDR	
0x0C	GPIOx_PUPDR (where x = C..IJK) Reset value	
0x10	GPIOx_IDR (where x = A..IJK) Reset value	
0x14	GPIOx_ODR (where x = A..IJK) Reset value	
0x18	GPIOx_BSRR (where x = A..IJK) Reset value	
0x1C	GPIOx_LCKR (where x = A..IJK) Reset value	
0x20	GPIOx_AFRL (where x = A..IJK) Reset value	AFRL7[3:0] AFRL6[3:0] AFRL5[3:0] AFRL4[3:0]
0x24	GPIOx_AFRH (where x = A..IJK) Reset value	AFRH15[3:0] AFRH14[3:0] AFRH13[3:0] AFRH12[3:0] AFRH11[3:0] AFRH10[3:0] AFRH9[3:0] AFRH8[3:0]

[See reference manual](#)

■ Basic input / output configuration

- Find pin numbers related to GPIOx or vice versa
- Configure through configuration registers
 - GPIOx_MODER
 - GPIOx_OTYPER
 - GPIOx_OSPEEDR
 - GPIOx_PUPDR

■ Data operations

- Input → Read register GPIOx_IDR
- Output → Write register GPIOx_ODR or GPIOx_BSRR

Exercise: GPIO Configuration

- **On a STM32F429 LQFP144 Pin 37 should be configured as low speed output with open-drain and pull-up**
 - What registers (names and addresses) must be configured?
 - Indicate the bits to be configured and their values!
 - No code required.

■ Accessing a register

```
#define GPIOA_MODER      (*((volatile uint32_t *)(0x40020000)))  
  
GPIOA_MODER = 0x55555555;      // all output
```

■ However

- Each GPIO port has the same 10 registers
 - There are 11 GPIO ports → GPIOA – GPIOK
- } → Results in 110 macros with repetitive code

■ We want an abstraction similar to base address and offset

Hardware Abstraction Layer (HAL)

reg_stm32f4xx.h

Base addresses

Pointers to struct of type reg_gpio_t

```
#define GPIOA          ( (reg_gpio_t *) 0x40020000 )
#define GPIOB          ( (reg_gpio_t *) 0x40020400 )
#define GPIOC          ( (reg_gpio_t *) 0x40020800 )
#define GPIOD          ( (reg_gpio_t *) 0x40020c00 )
#define GPIOE          ( (reg_gpio_t *) 0x40021000 )
#define GPIOF          ( (reg_gpio_t *) 0x40021400 )
#define GPIOG          ( (reg_gpio_t *) 0x40021800 )
#define GPIOH          ( (reg_gpio_t *) 0x40021c00 )
#define GPIOI          ( (reg_gpio_t *) 0x40022000 )
#define GPIOJ          ( (reg_gpio_t *) 0x40022400 )
#define GPIOK          ( (reg_gpio_t *) 0x40022800 )
```

base addresses

Offset

Typedef for reg_gpio_t

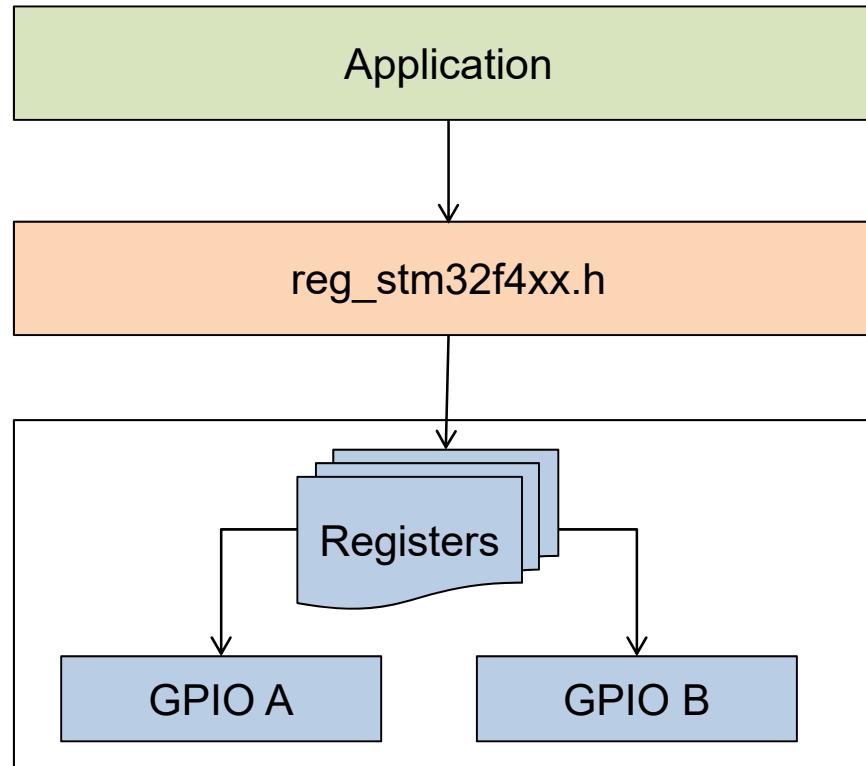
```
/**
 * \struct reg_gpio_t
 * \brief Representation of GPIO register.
 *
 * Described in reference manual p.265ff.
 */
typedef struct {
    volatile uint32_t MODER;      /**< Port mode register. */
    volatile uint32_t OTYPER;     /**< Output type register. */
    volatile uint32_t OSPEEDR;    /**< Output speed register. */
    volatile uint32_t PUPDR;      /**< Port pull-up/pull-down register. */
    volatile uint32_t IDR;        /**< Input data register. */
    volatile uint32_t ODR;        /**< output data register. */
    volatile uint32_t BSRR;       /**< Bit set/reset register */
    volatile uint32_t LCKR;       /**< Port lock register. */
    volatile uint32_t AFRL;       /**< AF low register pin 0..7. */
    volatile uint32_t AFRH;       /**< AF high register pin 8..15. */
} reg_gpio_t;
```

register names as
in reference manual

size of registers

```
GPIOA->MODER = 0x55555555; // all output
```

Hardware Abstraction Layer (HAL)



Lowest level of hardware abstraction layer, contains

- Base addresses
- Structs → members correspond to hardware registers
- Helper macros

Exercise: GPIO Configuration

■ Write the code to configure the bits from the last exercise

- GPIOA_MODER[7:6] → MODER3 = 01 → GP output
- GPIOA_OTYPER[3] → OT3 = 1 → open-drain
- GPIOA_OSPEEDR[7:6] → OSPEEDR3 = 00 → low speed
- GPIOA_PUPDR[7:6] → PUPDR3 = 01 → pull up

- Use the base addresses and structs in `reg_stm32f4xx.h`
- Do not change the other bits

```
#include "reg_stm32f4xx.h"

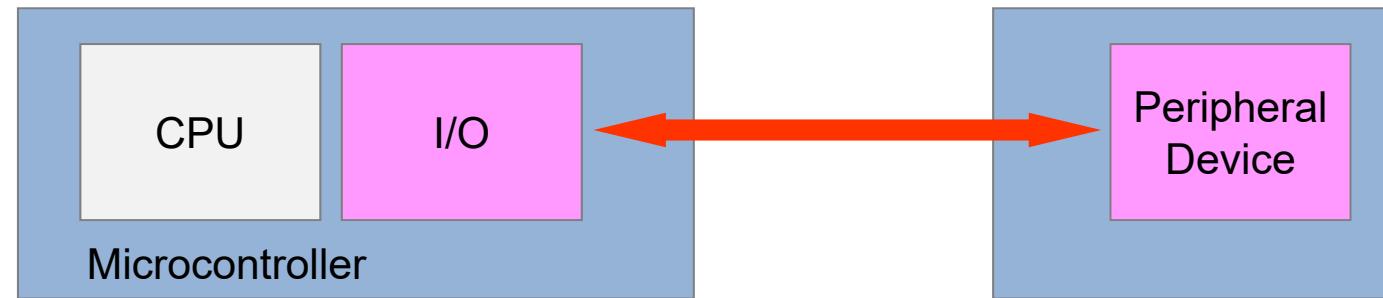
void config_gpioa_pin3(void)
{
}

}
```

Serial Data Transfer – SPI

Computer Engineering 2

■ Communication CPU – Peripheral Devices



■ Parallel Bus

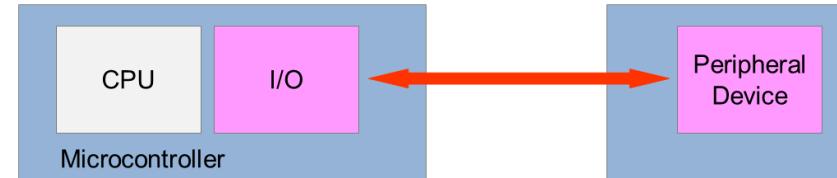
- n address lines
- m data lines
- Control lines (NWE, NOE)
- Decode logic

■ Serial Bus

- 2 to 4 lines
 - (CLK)
 - Data: Din, Dout
 - (Select)

■ Serial Connection → UART, SPI, I2C, etc.

- Provide simple, low level physical connection
 - Simpler → save PCB area
 - Reduce number of switching lines → reduce power, improve EMC¹⁾



- Requires "higher level" protocol
 - Usually in software
 - Error detection, reliability, quality-of-service (QoS)
 - Interpretation of commands

1) Electromagnetic compatibility:
Simultaneous switching of lines creates unwanted emission

- **Serial Communication**
- **SPI – Serial Peripheral Interface**
 - SPI Basics
 - SPI Modes
 - SPI – STM32F4xxx
 - SPI – Flash Devices

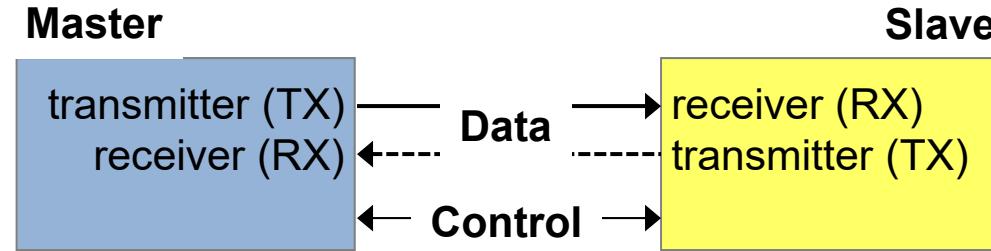
Learning Objectives

At the end of this lesson you will be able

- to explain what SPI is and how it works
- to outline the differences between the four SPI modes of operation
- to draw and interpret SPI timing diagrams
- to interpret the SPI block diagram of the STM32F4xxx
- to outline how SPI data transmission and reception has to be handled by software on the STM32F4xxx

■ Set-up

- Serial data line(s)
- Optional control lines



■ Communication Modes

- | | |
|---------------|---|
| • Simplex | Unidirectional, one way only |
| • Half-duplex | Bidirectional, only one direction at a time |
| • Full-duplex | Bidirectional, both directions simultaneously |

■ Timing

- | | |
|----------------|---|
| • Asynchronous | Each node uses an individual clock |
| • Synchronous | Both nodes use same clock
Clock often provided by master |

■ SPI – Serial Peripheral Interface

- Serial bus for on-board connections
 - Used for short distance communication
- Connects microcontroller and external devices
 - Sensors, A/D converters, displays, flash memories, codecs
 - IOs, Real Time Clocks (RTC), wireless transceivers...
- Synchronous
 - Master distributes the clock to slaves
- Compared to a parallel bus
 - Saves board area
 - Lowers pin count on both chips (TX and RX) → smaller, low-cost
 - Simplifies EMC (Electromagnetic compatibility)

■ SPI → De facto Standard

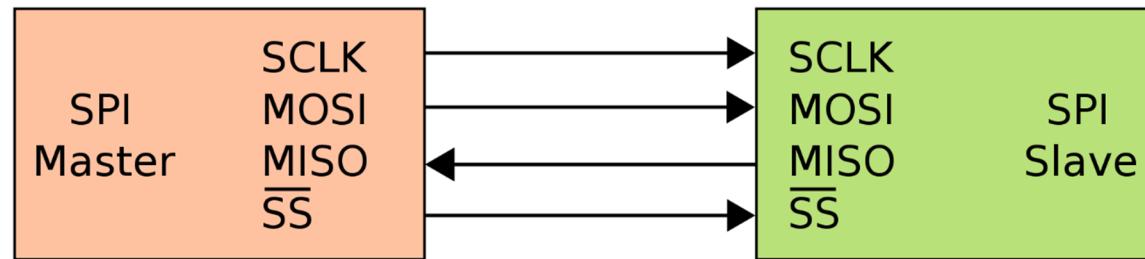
- No official standards organization
- No legally binding specification
 - Only chip datasheets and application notes
 - Many different variants exist
- Introduced by Motorola (today NXP) around 1979
- Also called 4-wire Bus

A de facto standard is a custom, convention, product, or system that has achieved a dominant position by public acceptance or market forces.

source: Wikipedia

■ Synchronous Serial Data Connection

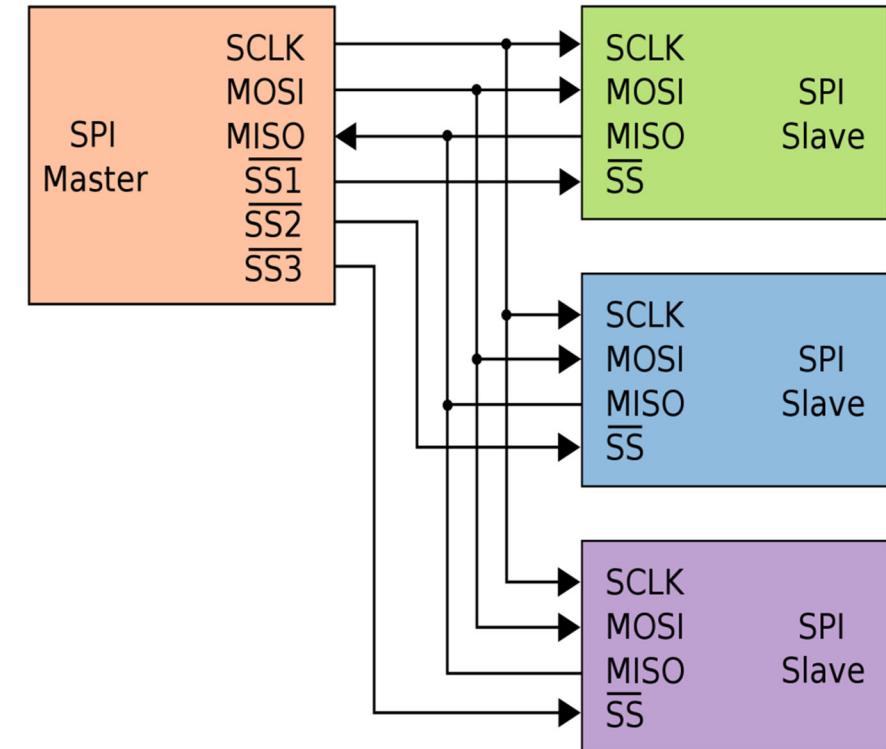
- Full duplex



- Master (single master)
 - Generates clock (SCLK)
 - Initiates data transfer by setting $\overline{SS} = 0$ (Slave Select)
- MOSI – Master Out Slave In
 - Data from master to slave
- MISO – Master In Slave Out
 - Data from slave to master

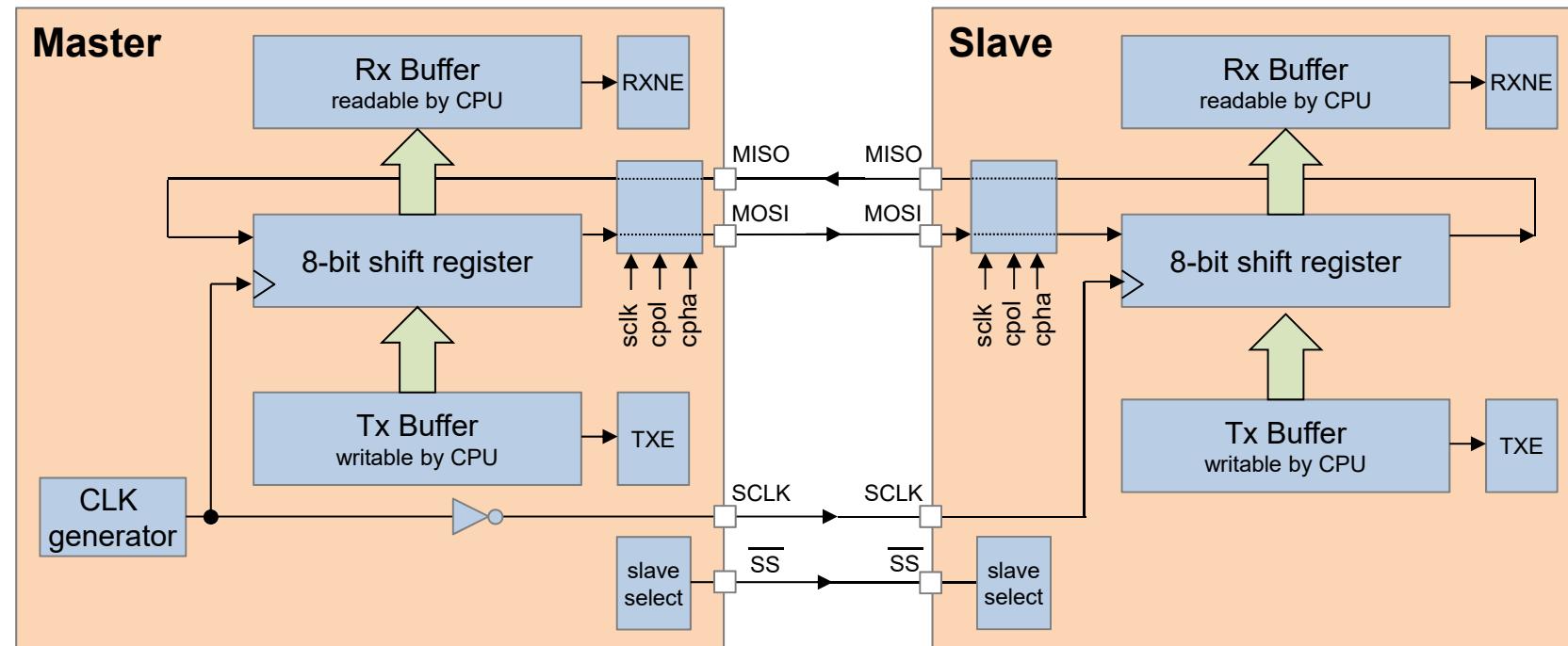
■ Single Master – Multiple Slaves

- Master generates a common clock signal for all slaves
- MOSI
 - From master output to all slave inputs
- MISO
 - All slave outputs connected to single master input
- Slaves
 - Individual select $\overline{SS1}$, $\overline{SS2}$, $\overline{SS3}$
 - $\overline{SSx} = '1'$ \rightarrow Slave output MISO x is tri-state



Source: Wikipedia

■ Implementation Using Shift Registers

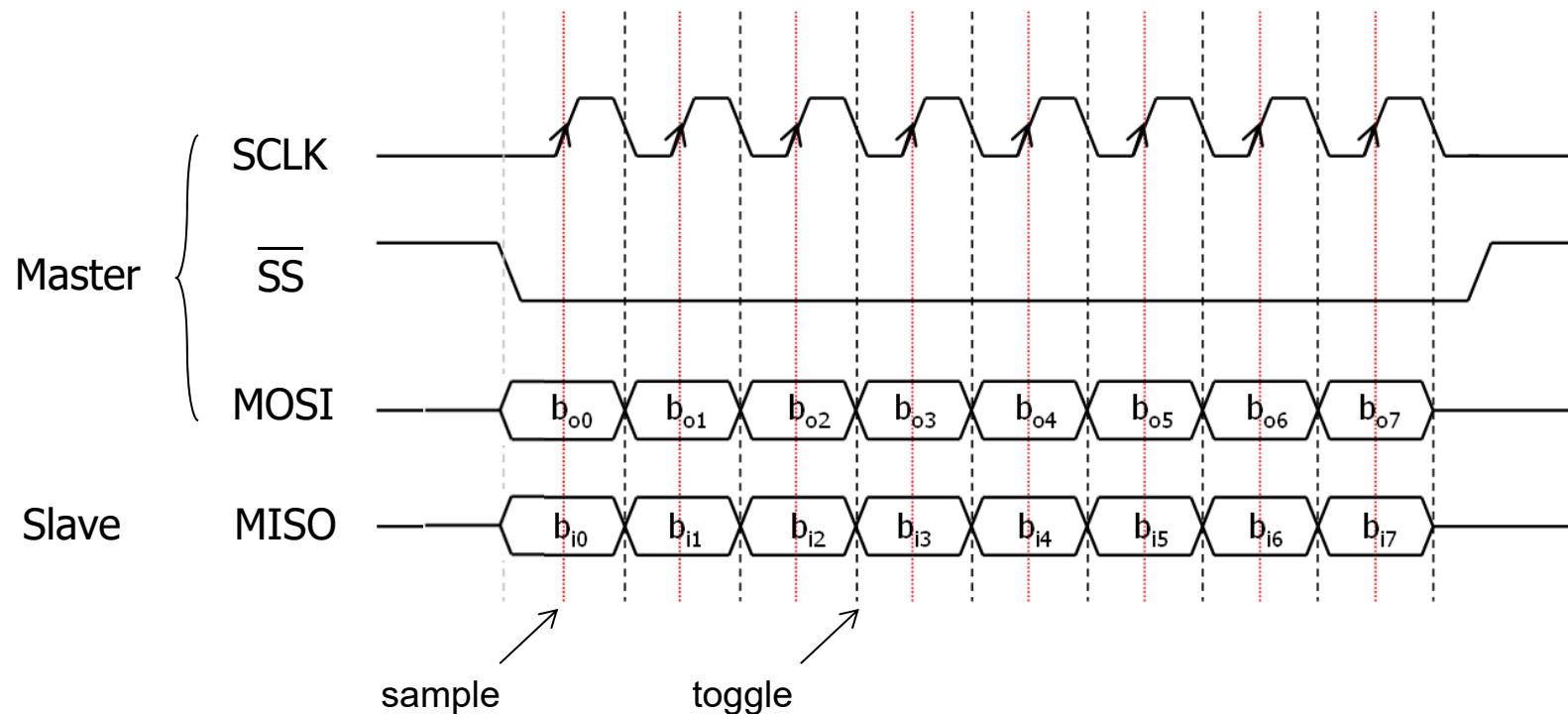
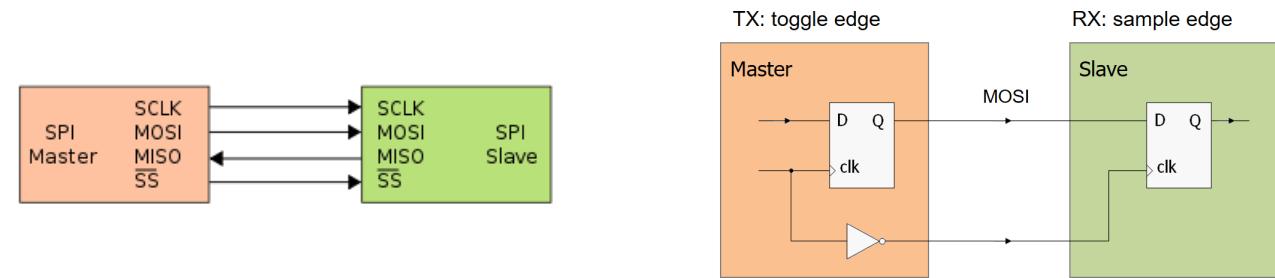


'LSB first' vs. 'MSB first' is configurable in most microcontrollers.
Slaves are often hard-wired.

Status bits with Interrupt
TXE Tx Buffer Empty
RXNE Rx Buffer Not Empty

■ Timing

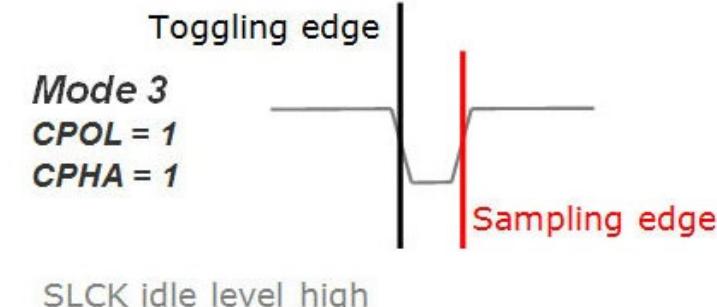
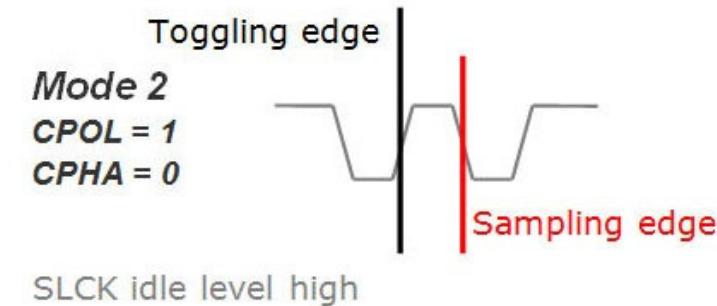
- Toggle output on one clock edge
- Sample on other clock edge



Clock Polarity and Clock Phase

- TX provides data on 'Toggling Edge'
- RX takes over data with 'Sampling Edge'

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

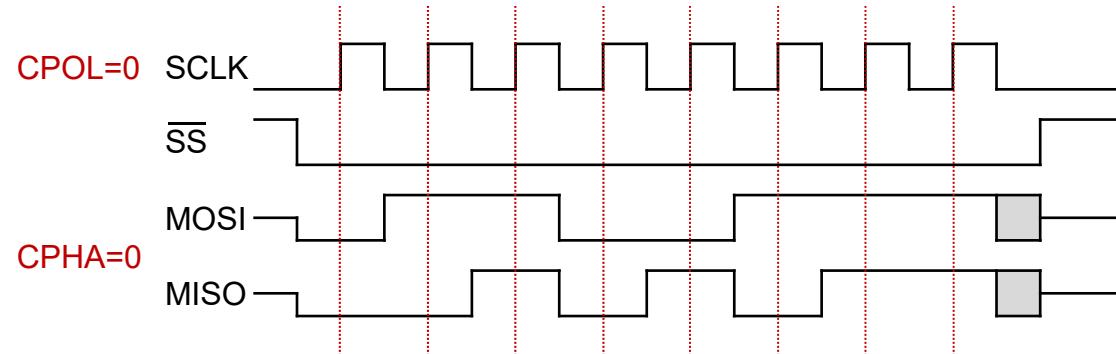


Source: <http://www.byteparadigm.com>

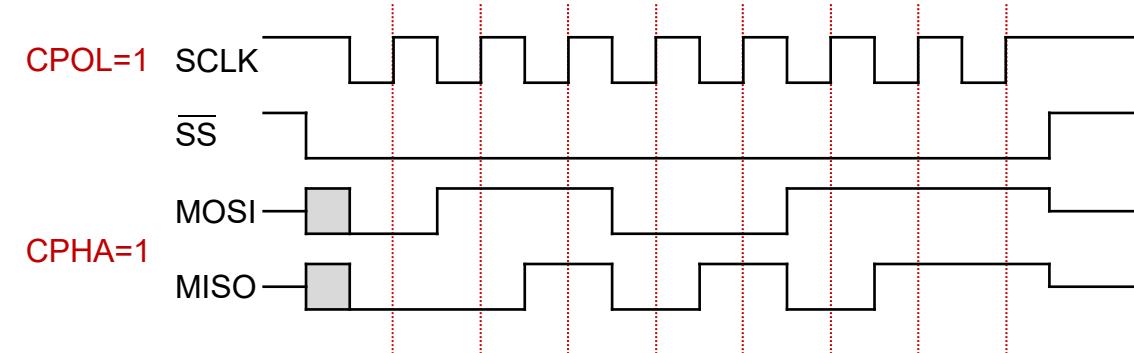
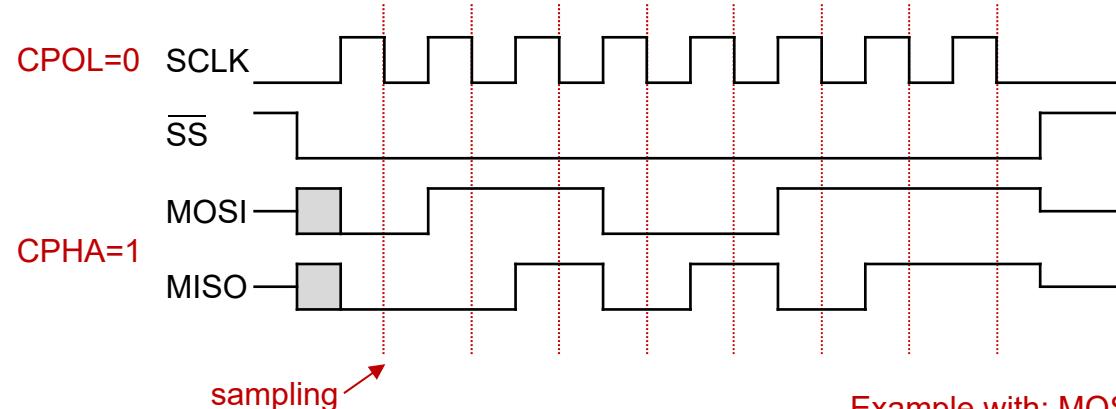
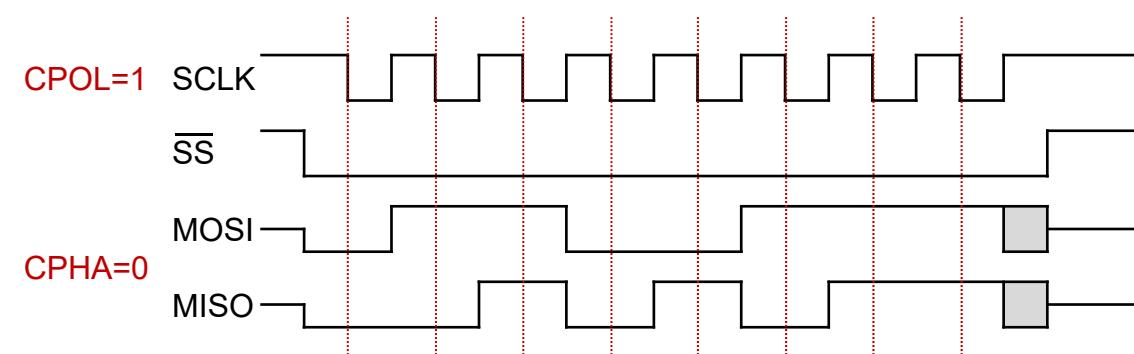
Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Clock Polarity and Clock Phase

- TX provides data on 'Toggling Edge'



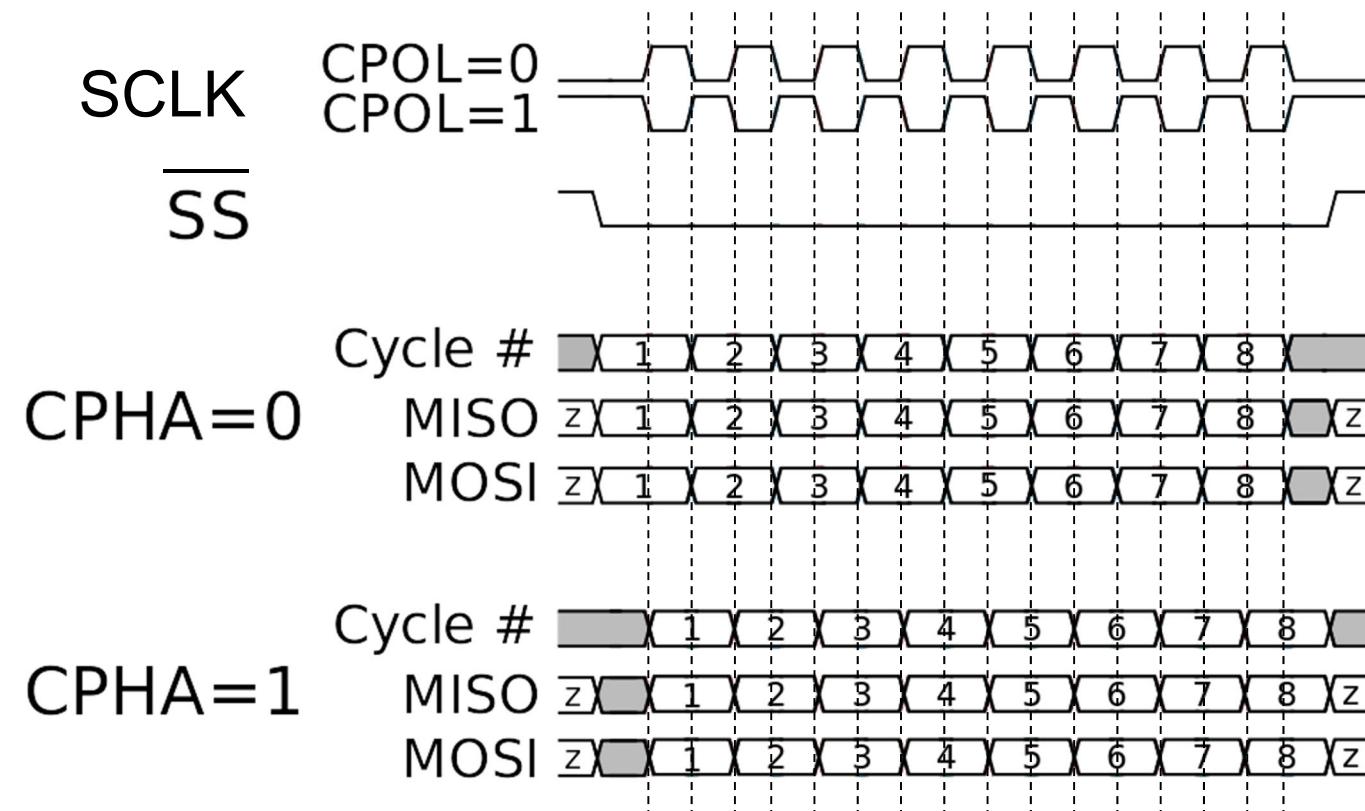
RX takes over data with 'Sampling Edge'



Example with: MOSI = 0x67 MISO = 0x2B, MSB first

■ Data and Clock

- Summary of all possible combinations

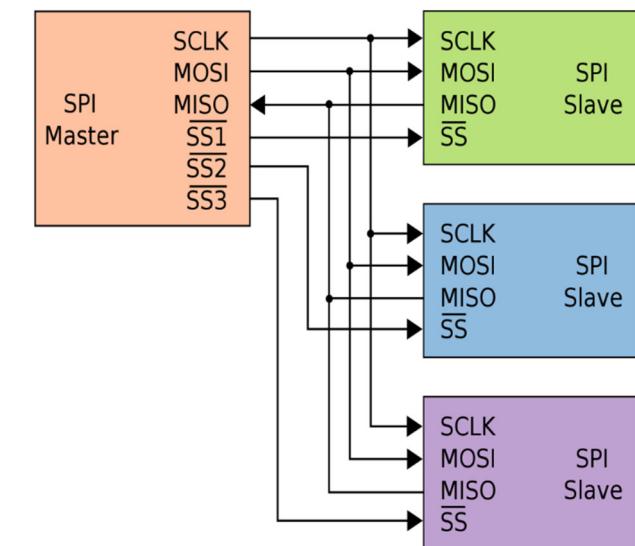


Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Source: Wikipedia

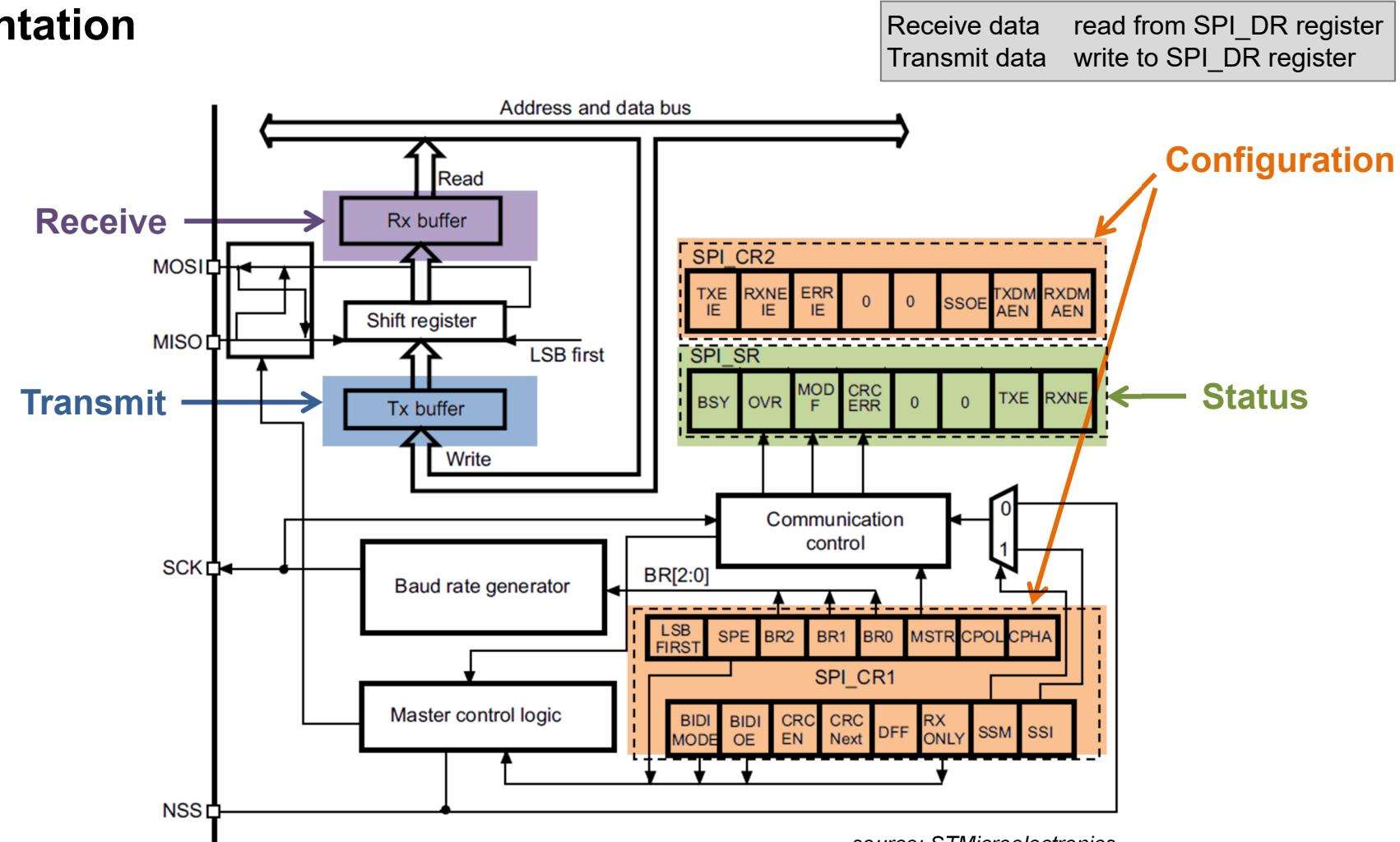
■ Properties

- No defined addressing scheme
 - Use of \overline{SS} instead → KISS
- Transmission without receive acknowledge and error detection
 - Has to be implemented in higher level protocols
- Originally used only for transmission of single bytes
 - \overline{SS} deactivated after each byte
 - Today also used for streams (endless transfers)
- Data rate
 - Highly flexible as clock signal is transmitted
- No flow-control available
 - Master can delay the next clock edge
 - Slave can not influence the data rate
- Susceptible to spikes on clock line



Source: Wikipedia

■ Implementation

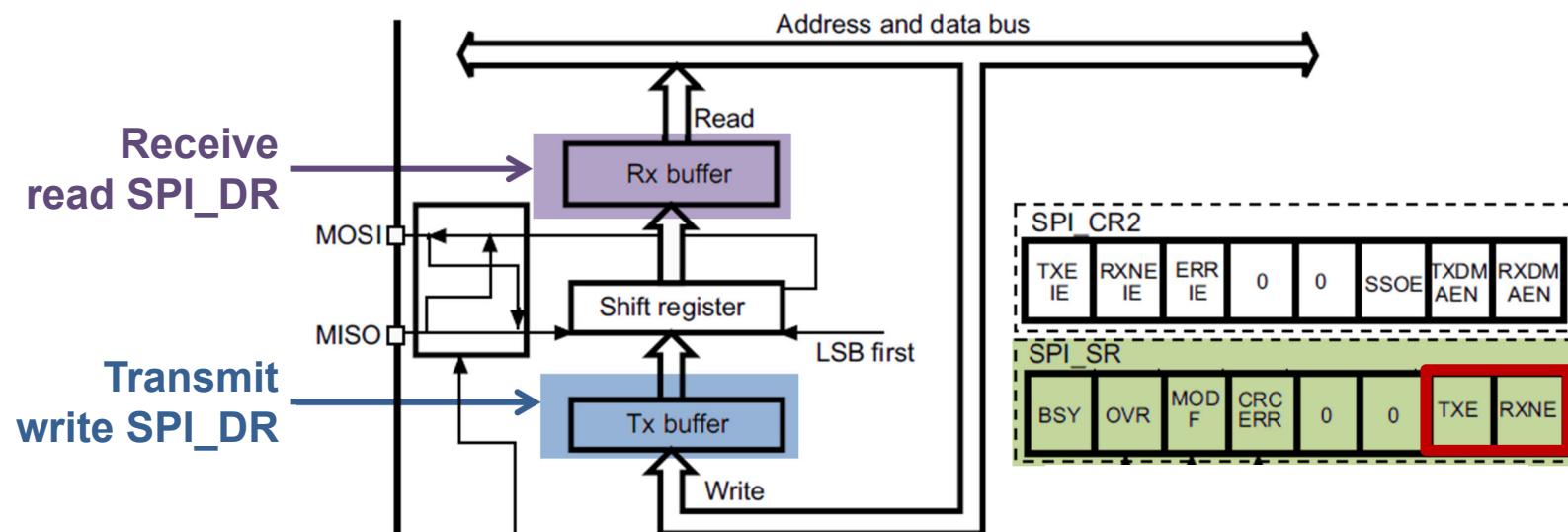


source: STMicroelectronics

■ Synchronizing Hardware and Software

- When shall software access the shift register?

- | | | |
|--------|---------------------|--|
| - TXE | TX Buffer Empty | Software can write next TX Byte to register SPI_DR |
| - RXNE | RX Buffer Not Empty | A byte has been received. Software can read it from SPI_DR |

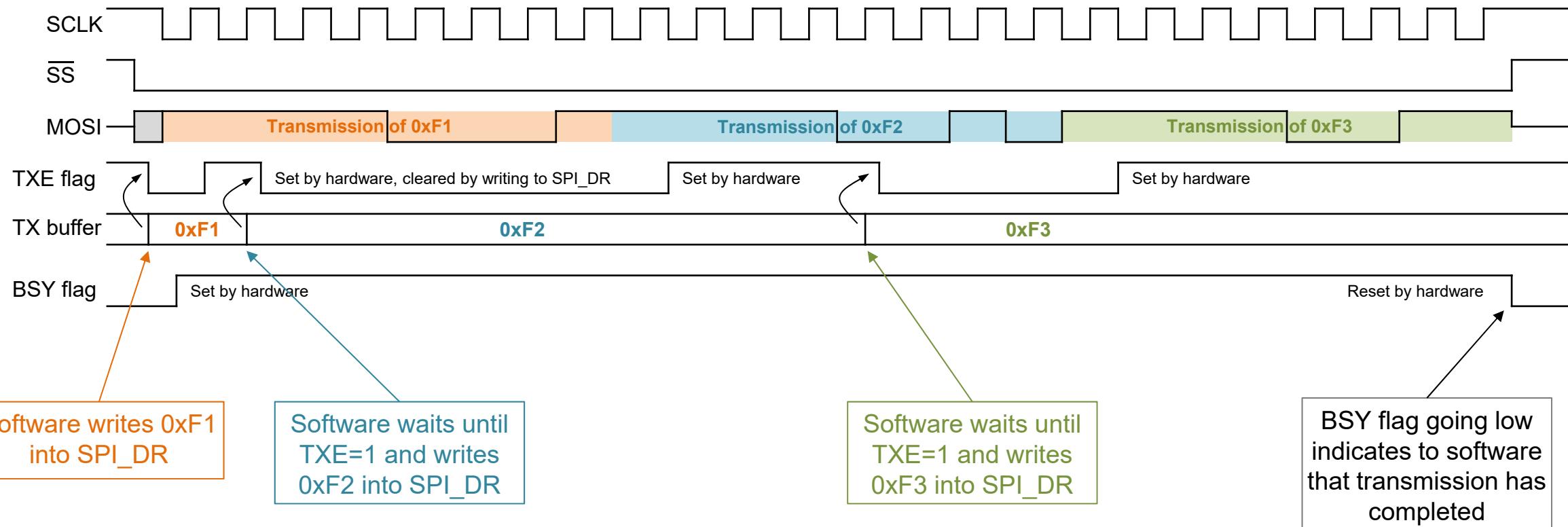


source: STMicroelectronics

■ Transmitting in Software

- Example: SW wants to transmit bytes 0xF1, 0xF2, 0xF3

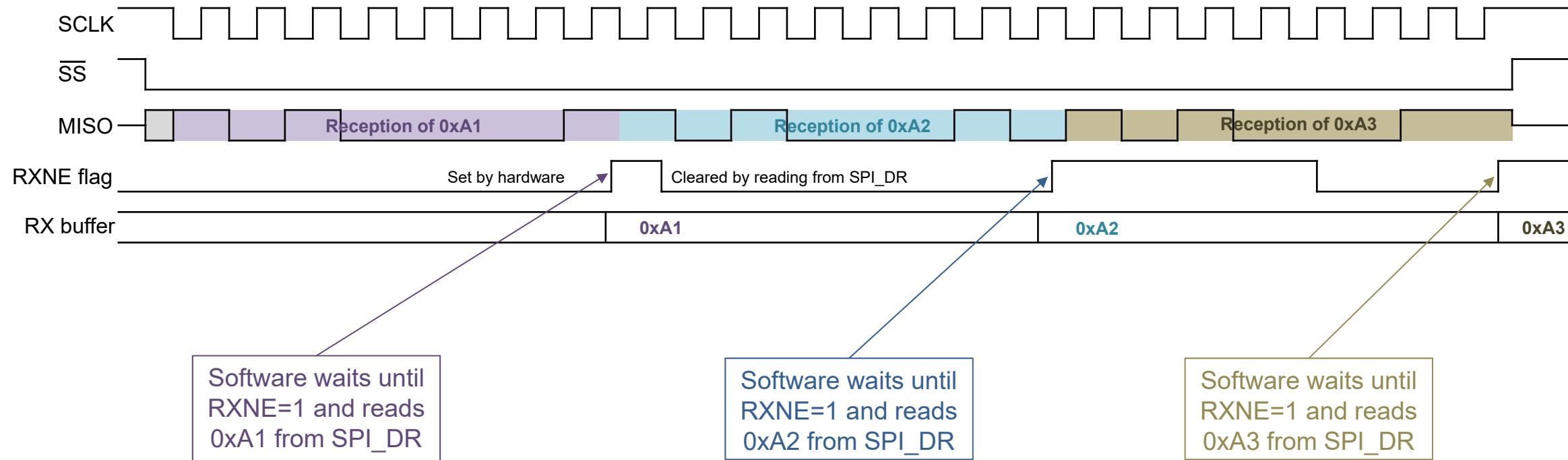
Example with CPOL=1, CPHA=1, MSB first



■ Receiving in Software

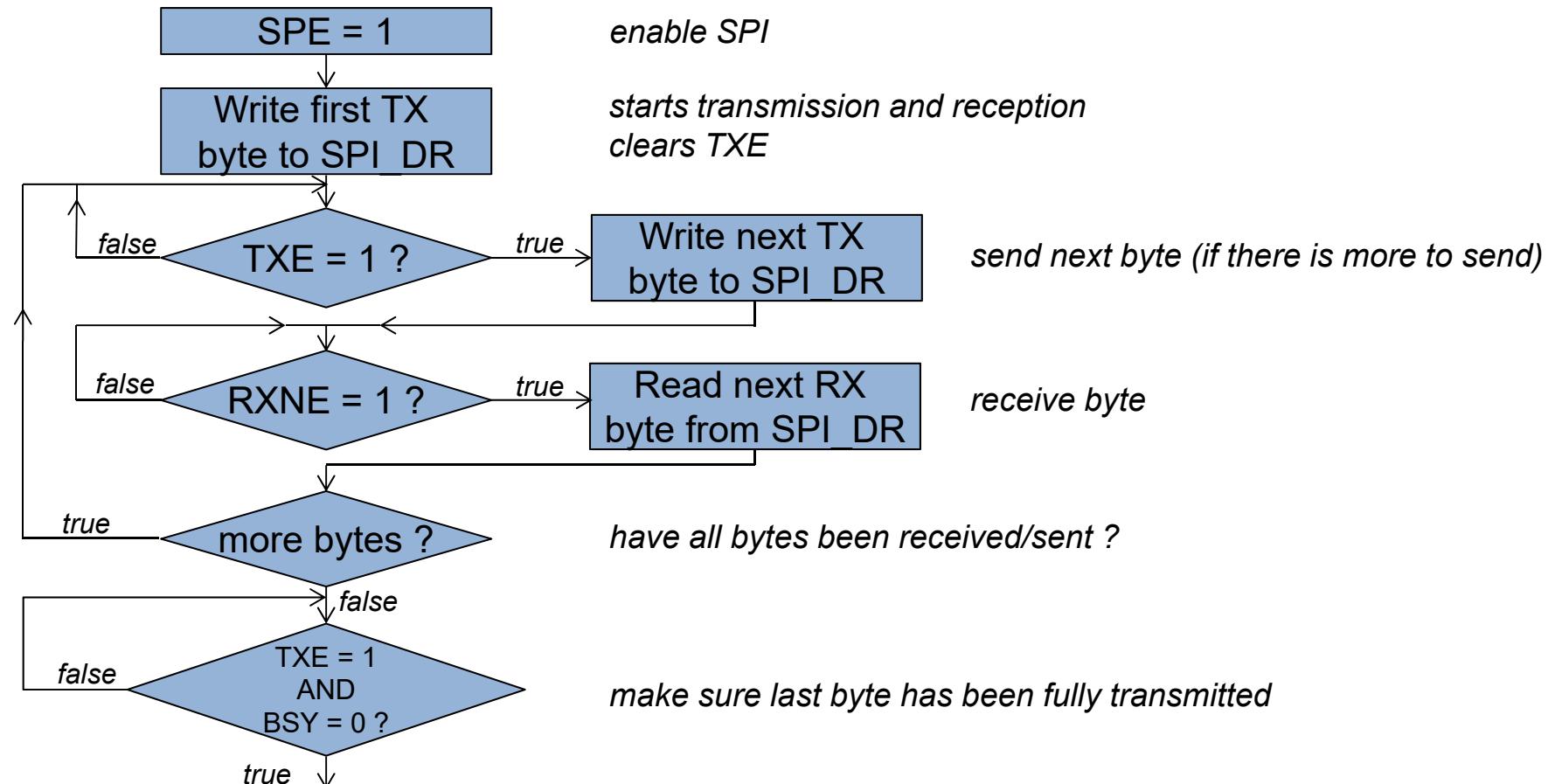
- Example: SW receives bytes 0xA1, 0xA2, 0xA3

Example with CPOL=1, CPHA=1, MSB first



■ Software: Simultaneously Handling Data Transmission and Reception

- Full duplex: Check TXE and RXNE bits



■ Interrupts

- Interrupts can be used instead of polling for TXE and RXNE bits

Position	Priority	Type of priority	Acronym	Description	Address
31	38	settable	I2C1_EV	I ² C1 event interrupt	0x0000 00BC
32	39	settable	I2C1_ER	I ² C1 error interrupt	0x0000 00C0
33	40	settable	I2C2_EV	I ² C2 event interrupt	0x0000 00C4
34	41	settable	I2C2_ER	I ² C2 error interrupt	0x0000 00C8
35	42	settable	SPI1	SPI1 global interrupt	0x0000 00CC
36	43	settable	SPI2	SPI2 global interrupt	0x0000 00D0
37	44	settable	USART1	USART1 global interrupt	0x0000 00D4
38	45	settable	USART2	USART2 global interrupt	0x0000 00D8
39	46	settable	USART3	USART3 global interrupt	0x0000 00DC

SPI Registers

- Total of 6 SPI blocks
 - Set of registers for each of them

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	SPI_CR1																BIDIMODE	BIDIOE	CRCEN	CRCNEXT	DFF	RXONLY	SSM	SSI									
		Reset value															0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x04	SPI_CR2																																
		Reset value															TXEIE	RXNEIE	SPE	ERRIE	FRF	Reserved	SSOE	MSTR	0	0	0	0	0	0	0	0	0
0x08	SPI_SR																FRE	BSY	OVR	MODF	CRCERR	UDR	CHSIDE	TXDMAEN	CPOL	1	0						
		Reset value															0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	SPI_DR																																
		Reset value																															

```

#define SPI1 ((reg_spi_t *) 0x40013000)
#define SPI2 ((reg_spi_t *) 0x40003800)
#define SPI3 ((reg_spi_t *) 0x40003c00)
#define SPI4 ((reg_spi_t *) 0x40013400)
#define SPI5 ((reg_spi_t *) 0x40015000)
#define SPI6 ((reg_spi_t *) 0x40015400)

```

■ Register Bits

SPI CR1 SPI control register 1

BIDIMODE Bidirectional data mode enable
 BIDIOE Output enable in bidir mode
 CRCEN Hardware CRC calculation enable
 CRCNEXT CRC transfer next
 DFF Data frame format (8-bit vs. 16-bit)
 RXONLY Receive only
 SSM Software slave management
 SSI Internal slave select
 LSBFIRST Frame format (bit order)
 SPE SPI enable
 BR[2:0] Baud rate control
 MSTR Master selection (master vs. slave)
 CPOL Clock polarity
 CPHA Clock phase

SPI DR SPI data register

DR[15:0] Data register

SPI CR2 SPI control register 2

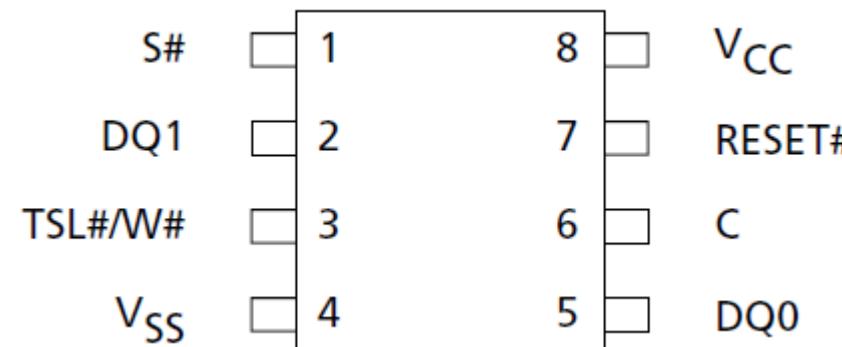
TXEIE Tx buffer empty interrupt enable
 RXNEIE RX buffer not empty interrupt enable
 ERRIE Error interrupt enable
 FRF Frame format (Motorola vs. TI mode)
 SSOE SS output enable
 TXDMAEN Txbuffer DMA enable

SPI SR SPI status register

FRE Frame format error
 BSY Busy flag (Txbuffer not empty)
 OVR Overrun flag
 MODF Mode fault
 CRCERR CRC error flag
 UDR Underrun flag
 CHSIDE Channel side (not used for SPI)
 TXE Transmit buffer empty
 RXNE Receive buffer not empty

■ Save Board Area

- E.g. Micron M25PE40 Serial Flash Memory
 - 4 Mbit NOR flash
 - 6 x 5 mm package size
 - SCLK: up to 75 MHz



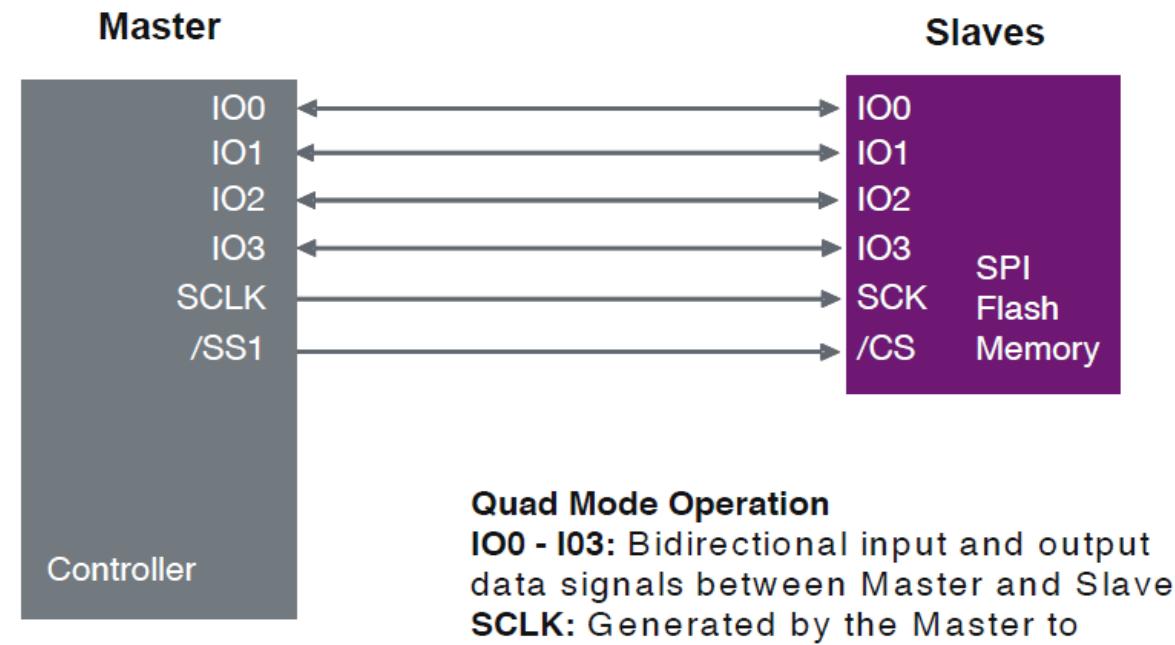
Signal Name	Function	Direction
C	Serial clock	Input
DQ0	Serial data	Input
DQ1	Serial data	Output
S#	Chip select	Input
W#	Write Protect	Input
RESET#	Reset	Input
V _{CC}	Supply voltage	–
V _{SS}	Ground	–

Source: Micron

■ Some Vendors Use More Than One Data Line

- Example: Flash using Quad I/O for Higher Bandwidth

FIGURE 1: QUAD I/O SERIAL INTERCONNECTION

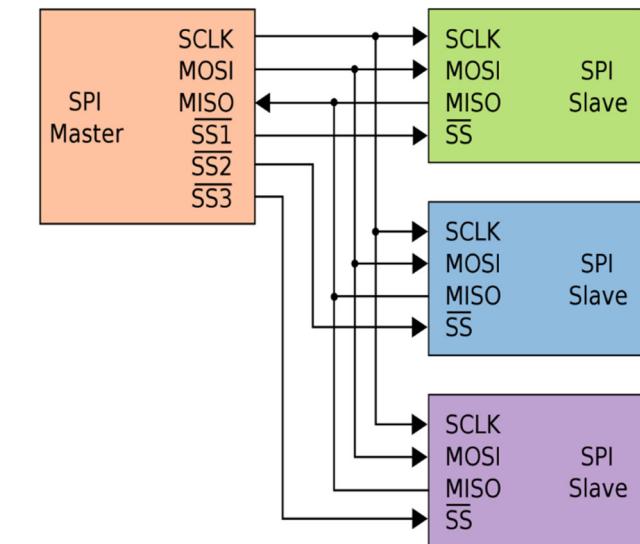
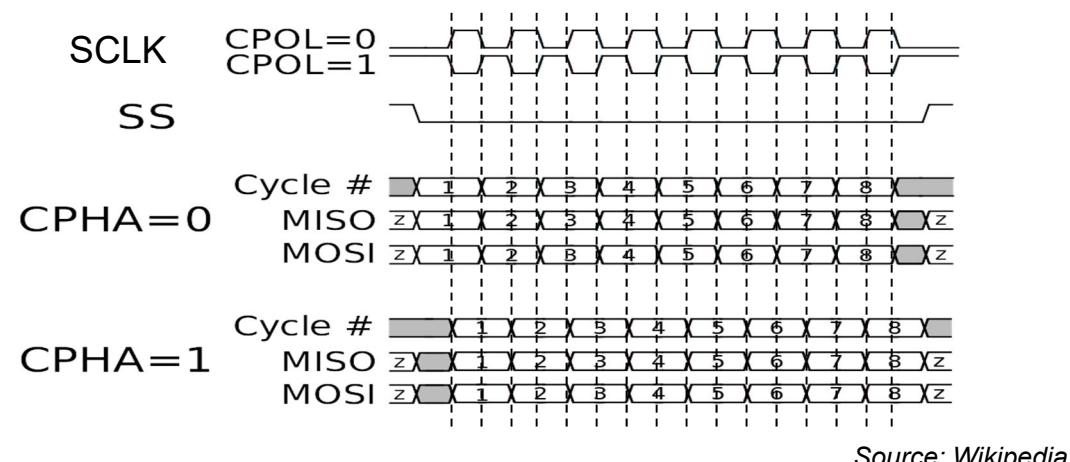


Source: Micron

Conclusions

■ SPI

- Master/Slave
- Synchronous full-duplex transmission (MOSI, MISO)
- Selection of device through Slave Select (\overline{SS})
- No acknowledge, no error detection
- Four modes → clock polarity and clock phase



Source: Wikipedia

Serial Data Transfer – UART / I2C

Computer Engineering 2

UART	SPI	I2C
serial ports (RS-232)	4-wire bus	2-wire bus
TX, RX , opt. control signals	$MOSI, MISO, SCLK, SS$	SCL, SDA
point-to-point	point-to-multipoint	(multi-) point-to-multi-point
full-duplex	full-duplex	half-duplex
asynchronous	synchronous	synchronous
only higher layer addressing	slave selection through \overline{SS} signal	7/10-bit slave address
parity bit possible	no error detection	no error detection
chip-to-chip, PC terminal program	chip-to-chip, on-board connections	chip-to-chip, board-to-board connections

The three interfaces provide the lowest layer of communication and require higher level protocols to provide and interpret the transferred data.

■ **Asynchronous Serial Interface**

- Universal Asynchronous Receiver Transmitter – UART
- Longer Distances: RS-232 / RS-485 – Electrical Characteristics
- U(S)ART – STM32F4xxx

■ **I2C Bus**

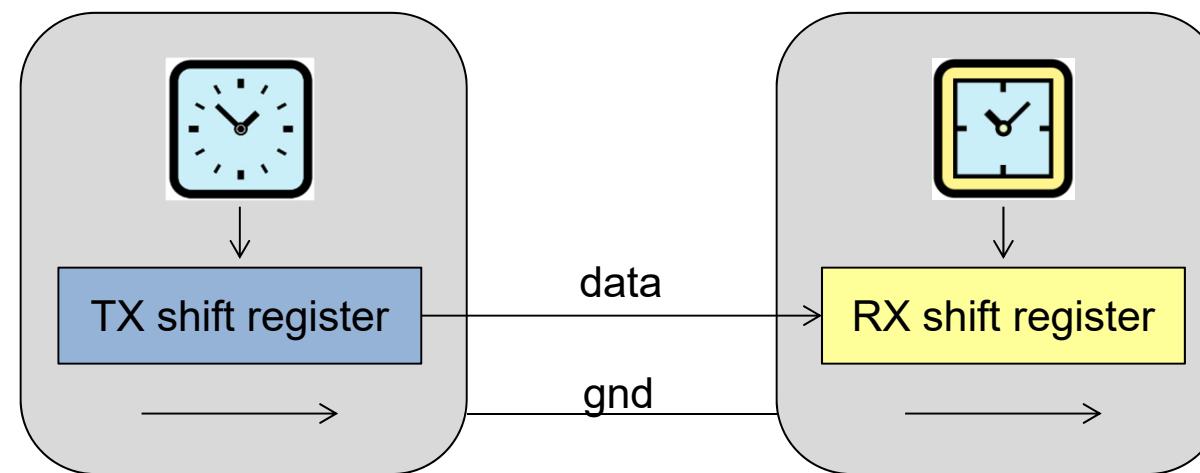
- Protocol / Operation
- I2C – STM32F4xxx

At the end of this lesson you will be able

- to explain how a UART works and which synchronization mechanism is used between transmitter and receiver
- to draw and interpret UART timing diagrams including data and overhead bits
- to program the UART interfaces on the STM32F4
- to explain what I2C is and how it works
- to interpret an I2C timing diagram
- to program the I2C interfaces on the STM32F4

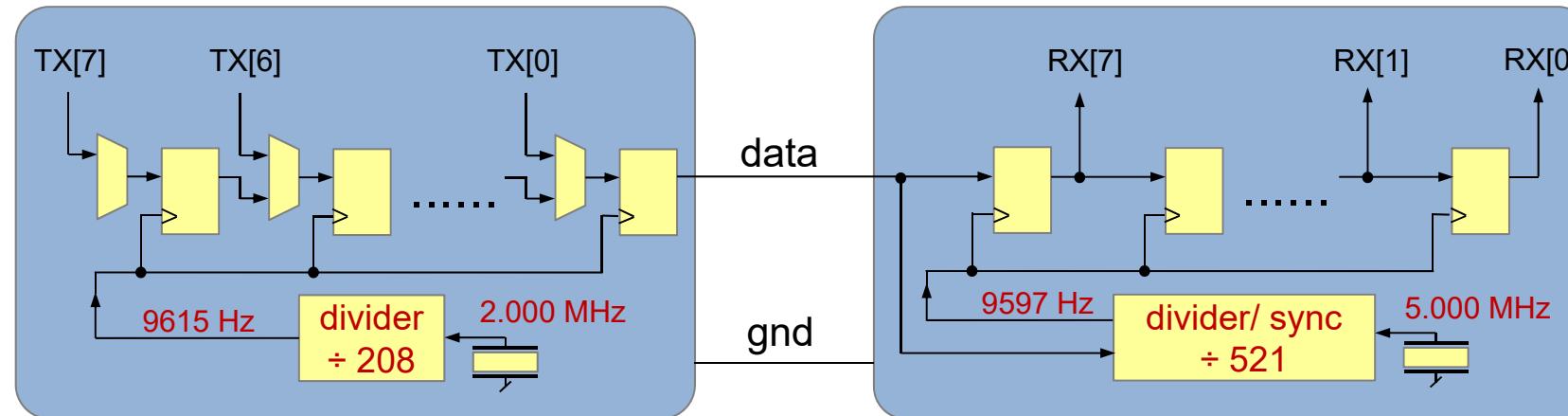
■ Universal Asynchronous Receiver Transmitter – UART

- Connecting shift registers with diverging clock sources
 - Same target frequency
 - Different tolerances and divider ratios
 - Requires synchronization at start of each data item in receiver



■ Implementation of UART → Shift register

- Example: 9'600 Baud with selected quartz frequencies



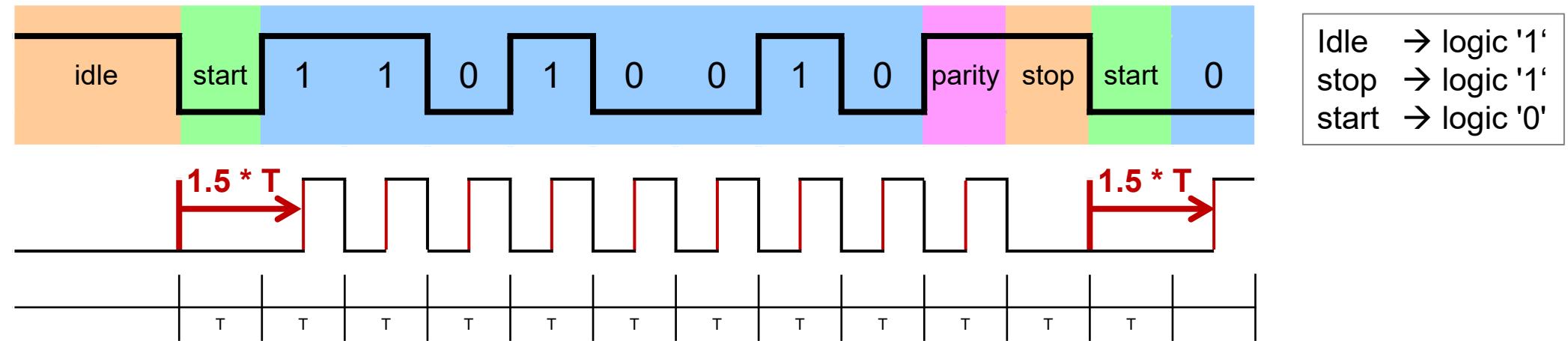
- Transmitter and receiver use closest integer to divide clock
 - $2.000 \div 208 = 9615 \text{ Hz} \rightarrow$ slightly too fast
 - $5.000 \div 521 = 9597 \text{ Hz} \rightarrow$ slightly too slow
- } settings allow successful transmission

Asynchronous Serial Interface

■ UART Timing

e.g. 9'600 Baud = symbols / s
 $\rightarrow T = (1 / 9600) \text{ s} = 0.104 \text{ ms}$

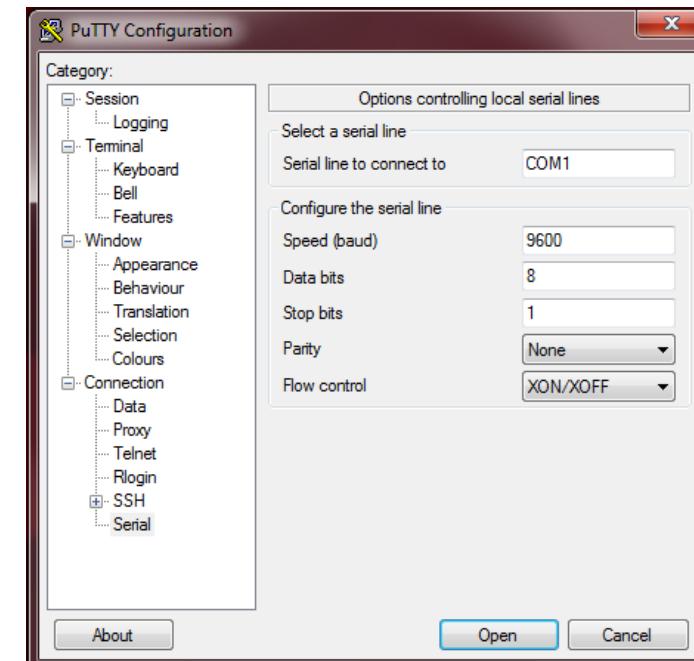
5 – 8 Bits of Data (LSB first)



- Transition stop ('1') \rightarrow start ('0')
 - Receiver detects edge at the start of each data block (5 to 8 bits)
 - Allows receiver to sample data "in middle of bits" \rightarrow red edges
 - Clocks have to be accurate enough to allow sampling up to parity bit, i.e. max. +/- 0.5 bit times aggregated deviation until last bit

■ UART – TX and RX Work with Same Settings

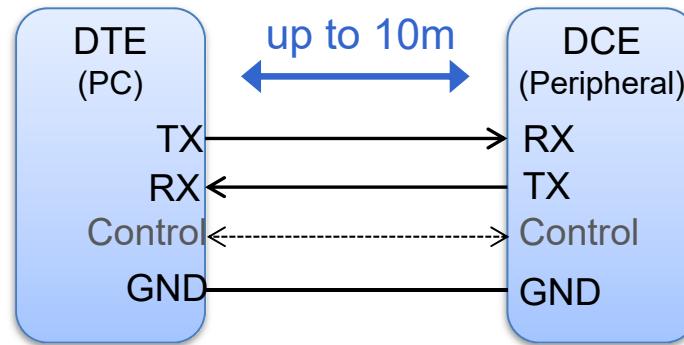
- Transmission rate
 - 2400, 4800, 9600, 19200, 38'400, 57'600, 115'200 ... bit/s
 - In case of UART: symbol rate = bit rate i.e. baud corresponds to bit/s
- Number of data bits
 - Between 5 and 8 bit
- Number of stop bits
 - 1, 1.5 or 2 bit
- Parity
 - none
 - mark (logic '1')
 - space (logic '0')
 - even
 - odd



■ UART Characteristics

- Synchronization
 - Each data item (5-8 bits) requires synchronization
- Asynchronous data transfer
 - Mismatch of clock frequencies in TX and RX
 - Requires overhead for synchronization → additional bits
 - Requires effort for synchronization → additional hardware
- Advantage
 - Clock does not have to be transmitted
 - Transmission delays are automatically compensated
- On-board connections
 - Signal levels are 3V or 5V with reference to ground
 - Off-board connections require stronger output drivers (circuits)

■ RS-232¹⁾ – Interconnecting Equipment by Cable



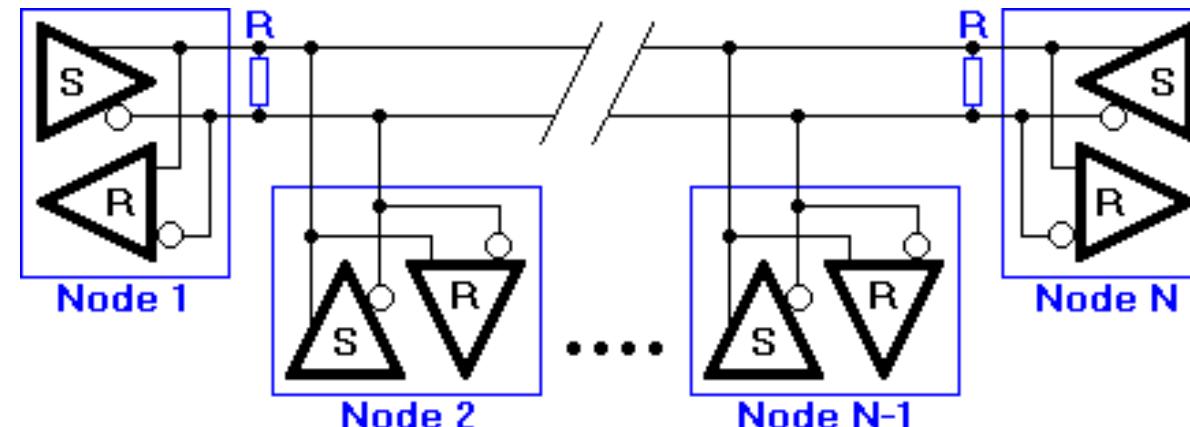
source: Olimex

- Simple, bidirectional point-to-point interface based on UART
- Optional control signals, e.g. CTS – Clear To Send
- Ground → common reference level for all signals (single ended)
- Driver circuit allows transmissions up to ~ 10 m
 - Logic '1' -3V to -15V
 - Logic '0' 3V to 15V

1) standardized by the Electronic Industries Alliance (EIA)

■ RS-485 – Differential Transmission Based on UART

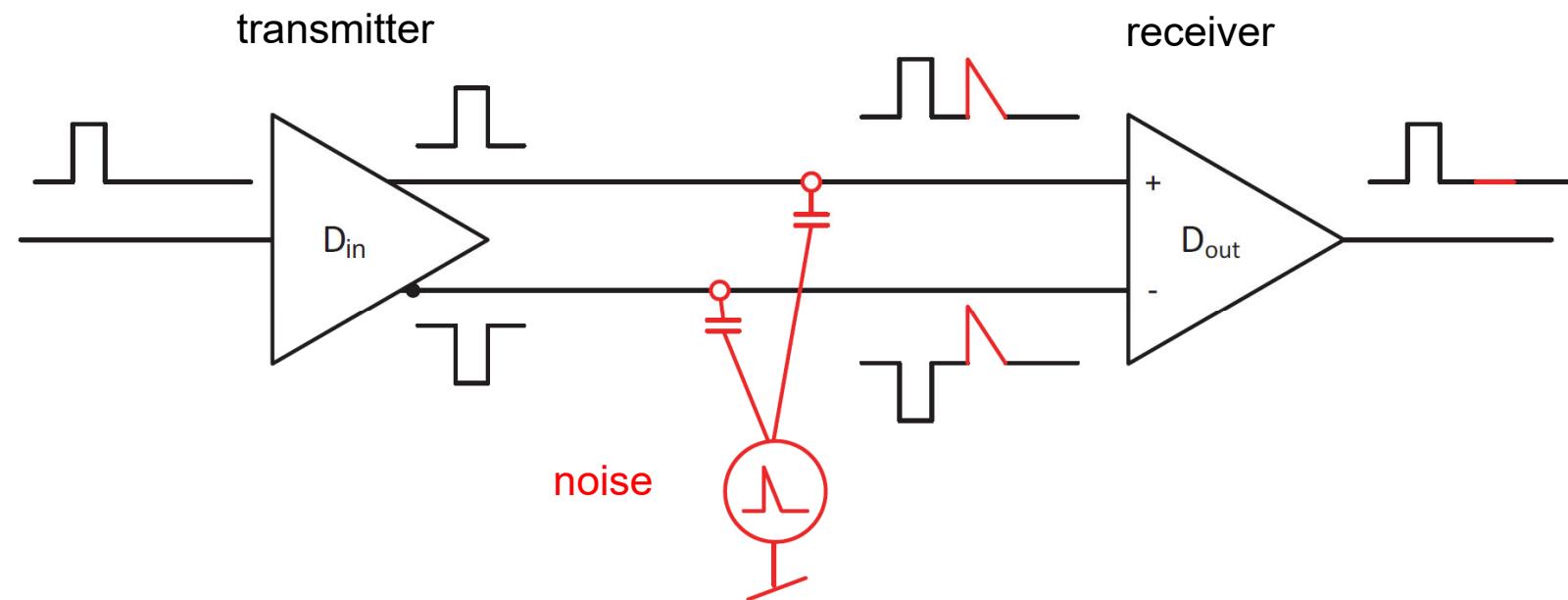
- Differential signals
 - Less susceptible to disturbances → longer distances, 100+ Meters
- Transmit and receive share the same lines
 - Multi-point communication
 - Half-duplex
- Industrial automation → E.g. lowest layer of Profibus



source: <http://www.lammertbies.nl/comm/info/RS-485.html>

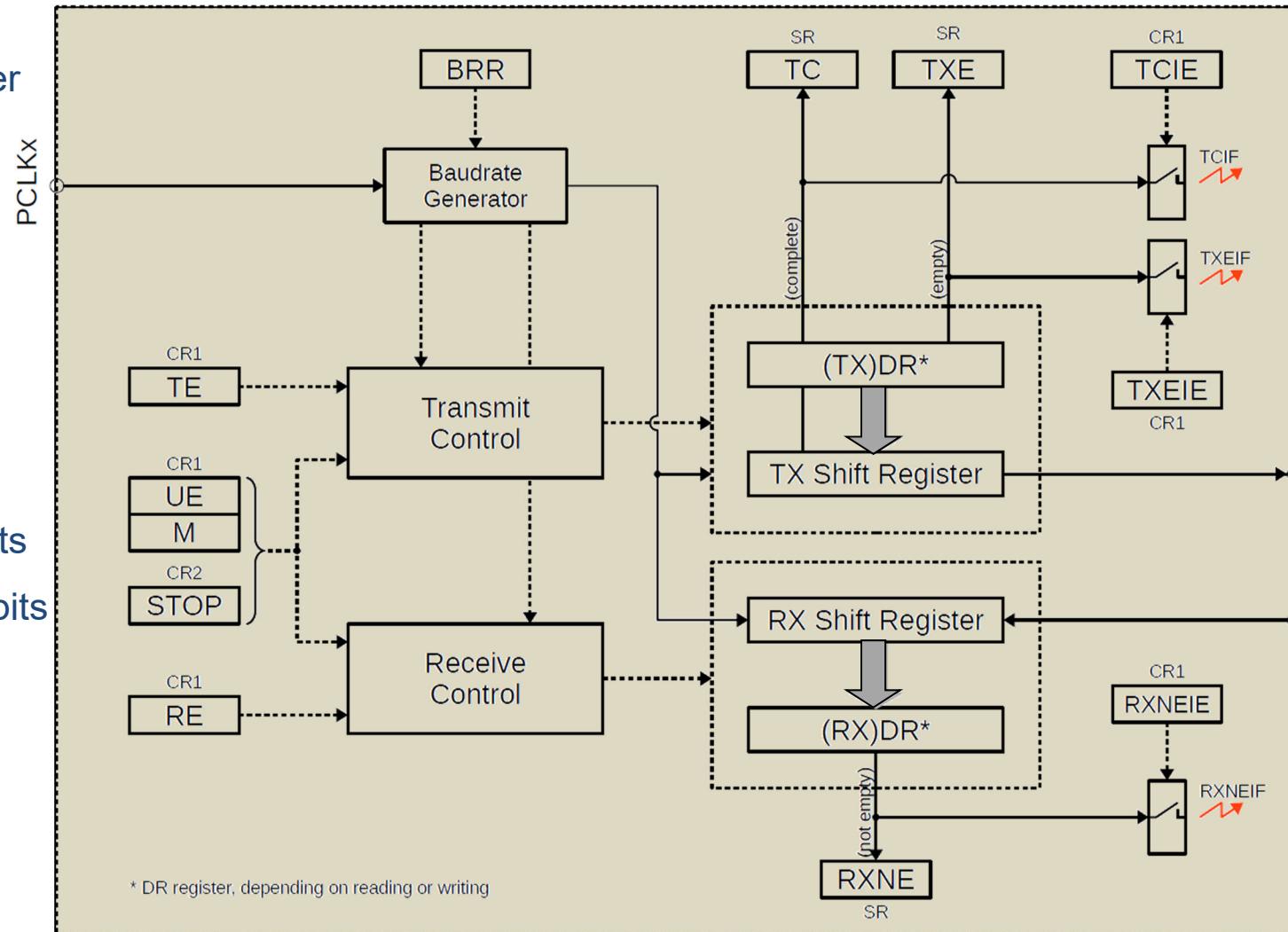
■ RS-485 – Differential Transmission

- Transmitter transforms single ended signal into differential signal
- Capacitive coupled noise affects both lines
- Receiver forms the difference of the two signals
 - Noise on the two lines cancels itself to a large extent



U(S)ART – STM32F4xxx

Baud rate register



TX enable

USART enable
M: 8 vs 9 data bits

Number of stop bits

RX enable

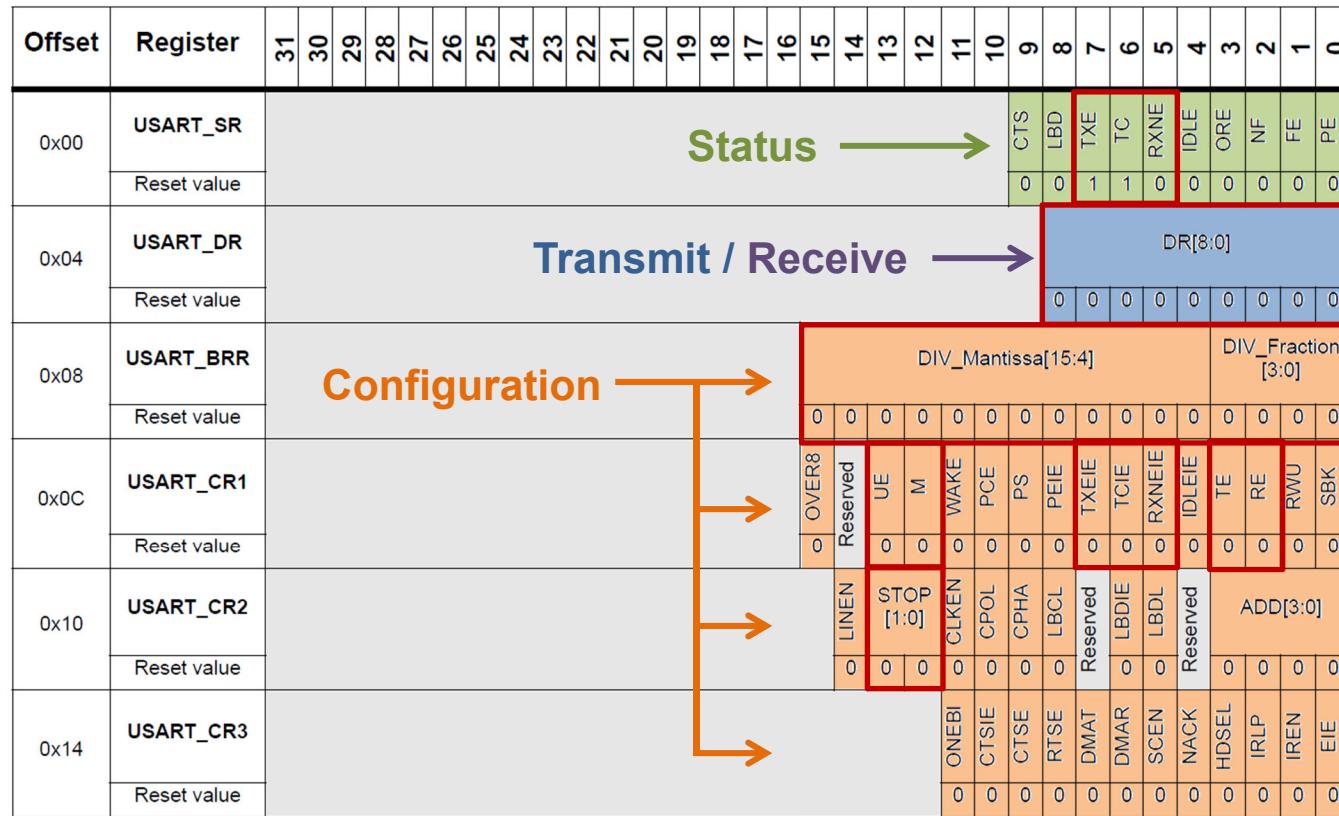
Transmission complete
interrupt (flag)

Transmit data register empty
interrupt (flag)

TX
To/from GPIO pins

RX
Received data register not
empty interrupt (flag)

■ USART Registers



Base addresses
of 8 U(S)ART blocks

```

#define USART1 ((reg_usart_t*) 0x40011000)
#define USART2 ((reg_usart_t*) 0x40004400)
#define USART3 ((reg_usart_t*) 0x40004800)
#define USART4 ((reg_usart_t*) 0x40004c00)
#define USART5 ((reg_usart_t*) 0x40005000)
#define USART6 ((reg_usart_t*) 0x40011400)
#define USART7 ((reg_usart_t*) 0x40007800)
#define USART8 ((reg_usart_t*) 0x40007c00)

```

Use of DR and TXE / RXNE is analogous to SPI

■ I²C – Inter-Integrated Circuit, pronounced I-squared-C

- Bidirectional 2-wire
- Defined by Philips Semiconductors, now NXP
- First release in 1982

Reference Documentation

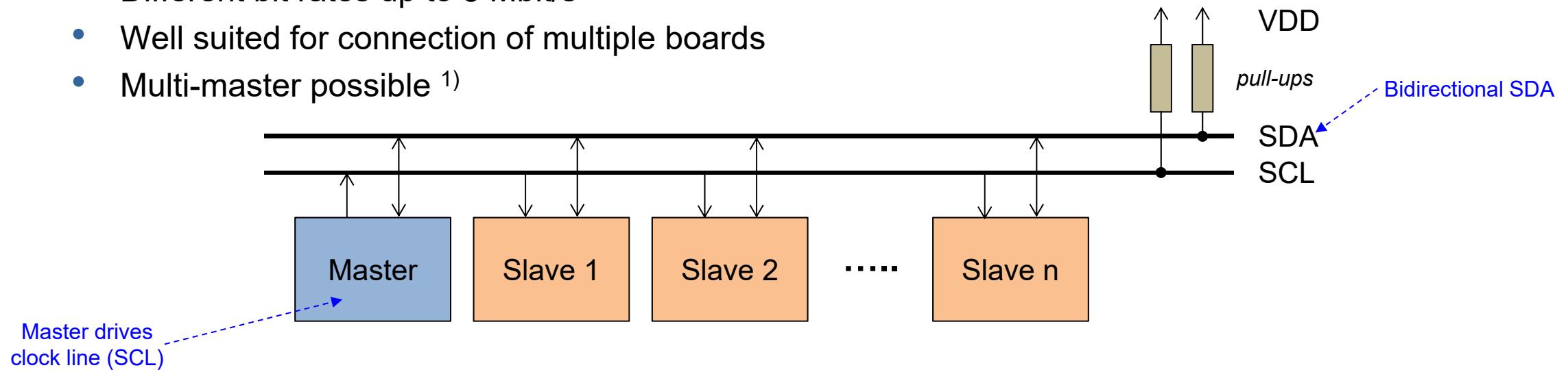
UM10204 – I2C-bus specification and user manual
Rev. 6 — 4 April 2014



source: NXP

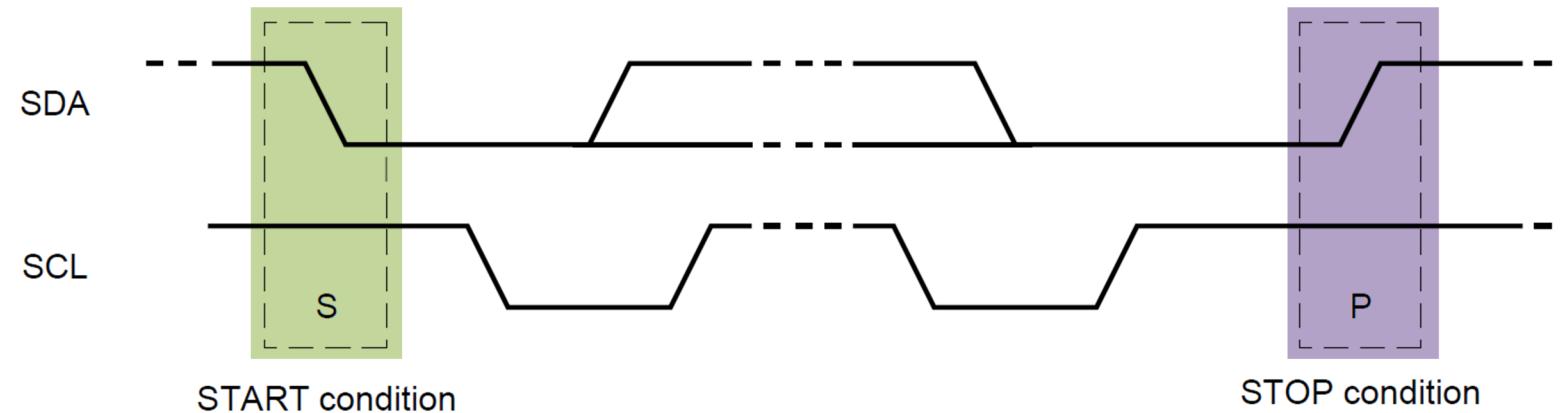
■ Overview

- 2-wire bus Clock → SCL Data → SDA
- Synchronous, half-duplex
- Each device on bus addressable through unique address
 - NXP assigns manufacturer IDs
- 8-bit oriented data transfers
- Different bit rates up to 5 Mbit/s
- Well suited for connection of multiple boards
- Multi-master possible ¹⁾



■ Operation

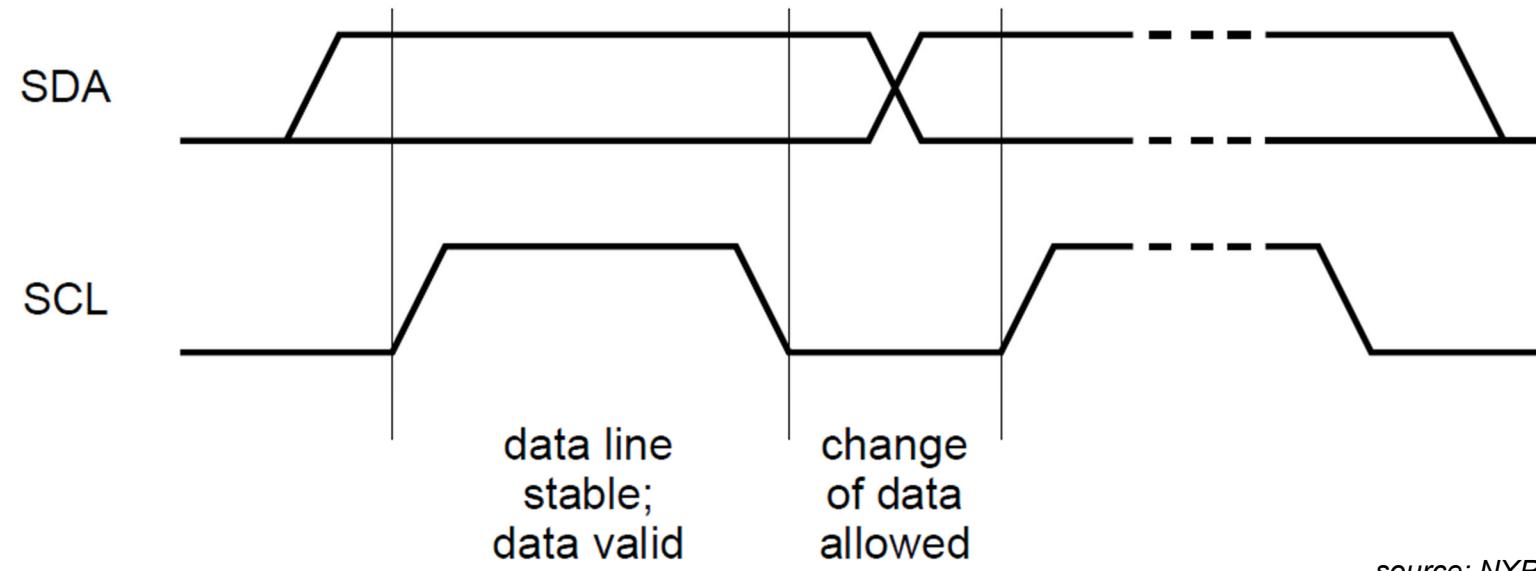
- Master drives clock line (SCL)
- Master initiates transaction through START condition
 - Falling edge on SDA when SCL high
- Master terminates transaction through STOP condition
 - Rising edge on SDA when SCL high



source: NXP

■ Driving Data on SDA

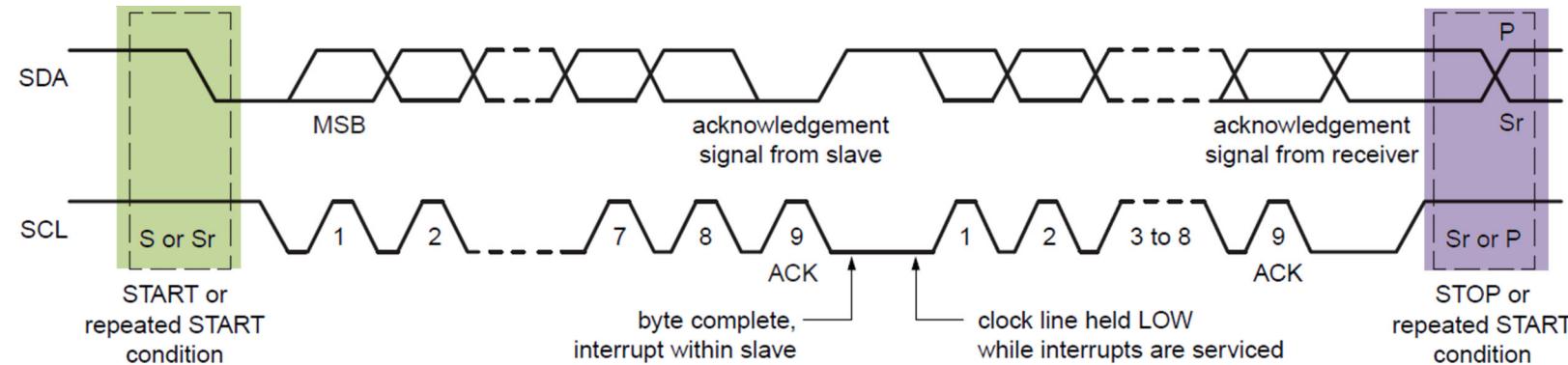
- Data driven onto SDA by master or by addressed slave
 - Depending on transaction (read/write) and point in time
 - Change of data only allowed when SCL is low
 - Allows detection of START and STOP condition



source: NXP

■ Data Transfer on I2C

- 8-bit oriented transfers
- Bit 9: Receiver acknowledges by driving SDA low
- Master defines number of 8-bit transfers (STOP)

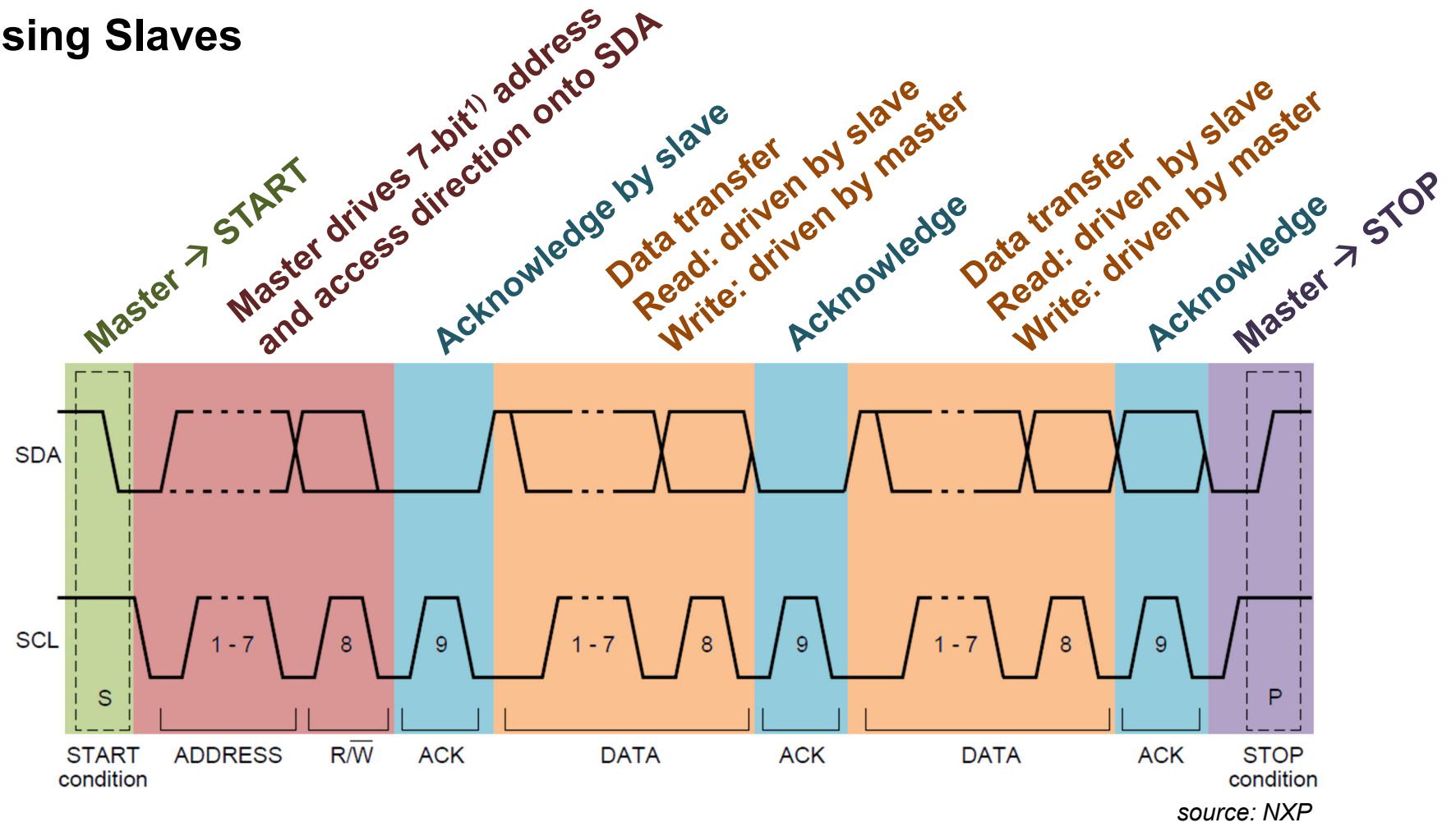


Bit numbering starts with 1; not with 0

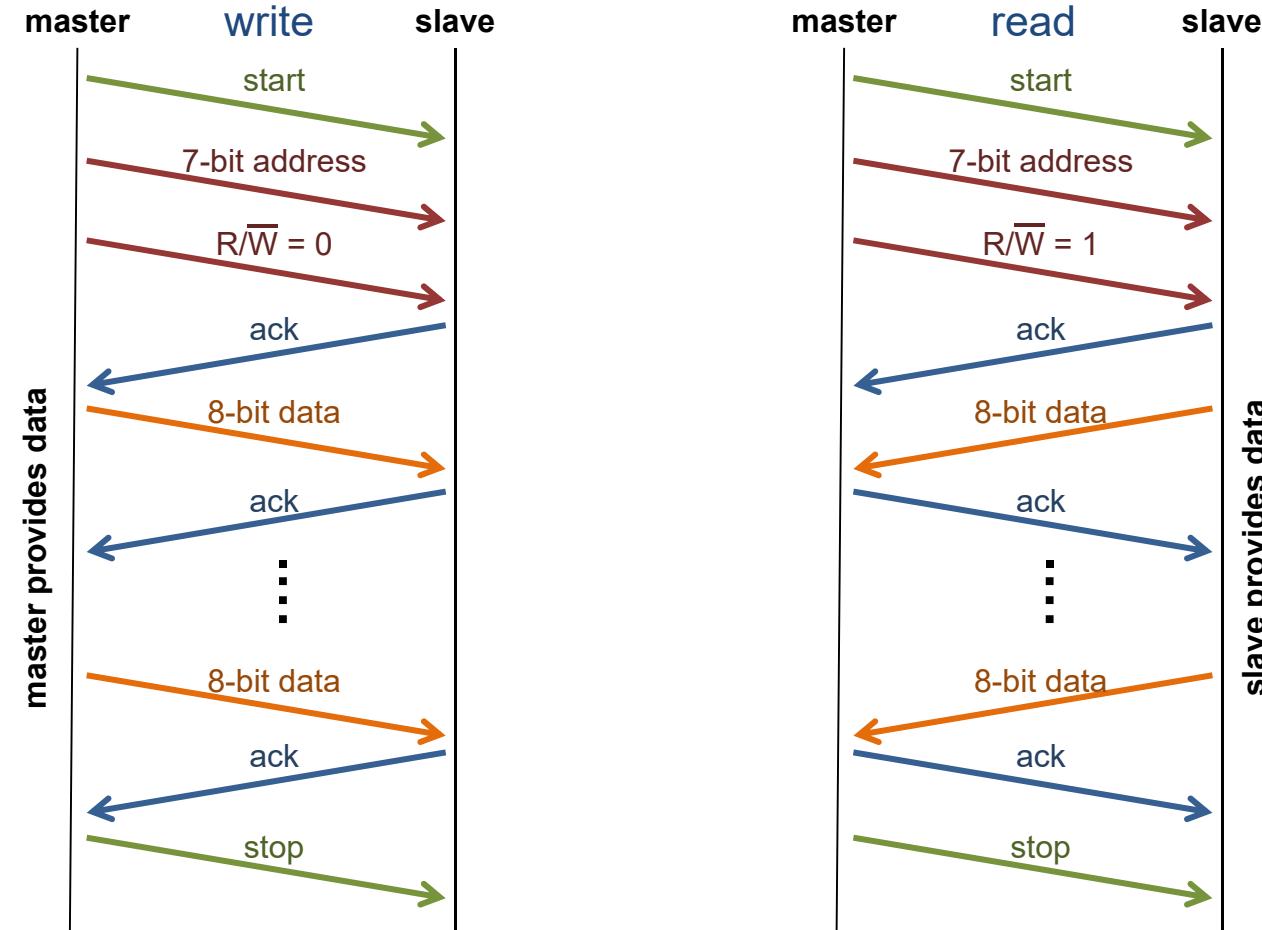
'ACK' is active low

source: NXP

■ Addressing Slaves



■ Accesses



I²C Registers

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
0x00	I ² C_CR1																																				
	Reset value																																				
0x04	I ² C_CR2																																				
	Reset value																																				
0x08	I ² C_OAR1																																				
	Reset value																																				
0x0C	I ² C_OAR2																																				
	Reset value																																				
0x10	I ² C_DR																																				
	Reset value																																				
0x14	I ² C_SR1																																				
	Reset value																																				
0x18	I ² C_SR2																																				
	Reset value																																				

Configuration

Own Addresses

Transmit / Receive

Status

Base addresses of 3 I²C blocks

```
#define I2C1 ((reg_i2c_t *) 0x40005400)
#define I2C2 ((reg_i2c_t *) 0x40005800)
#define I2C3 ((reg_i2c_t *) 0x40005c00)
```

■ Register Bits

I²C_CR1	I²C control register 1	I²C_OAR1	I²C own address register 1	I²C_SR2	I²C status register 2
SWRST	Software reset	ADDMODE	Addressing mode (7-bit vs. 10-bit)	PEC[7:0]	Packet error checking register
ALERT	SMBus alert	ADD[9:8]	Interface address	DUALF	Dual flag (slave)
PEC	Packet error checking	ADD[7:1]	Interface address	SMBHOST	SMBus host header (slave)
POS	Acknowledge / PEC Position	ADD0	Interface address	SMBDEFAULT	SMBus device default address (slave)
ACK	Acknowledge enable			GNCALL	General call address (slave)
STOP	Stop generation			TRA	Transmitter/receiver (R/W bit)
START	Start generation			BUSY	Bus busy (communication ongoing on bus)
NOSTRETCH	Clock stretching disable (slave)			MSL	Master/slave
ENGC	General call enable				
ENPEC	PEC enable				
ENARP	ARP enable				
SMBTYPE	SMBus type (device vs. host)				
SMBUS	SMBus mode (I ² C vs. SMBus)				
PE	Peripheral enable				
I²C_CR2	I²C control register 2				
LAST	DMA last transfer				
DMAEN	DMA requests enable				
ITBUFEN	Buffer interrupt enable				
ITEVEN	Event interrupt enable				
ITERREN	Error interrupt enable				
FREQ[5:0]	Peripheral clock frequency				
I²C_DR	I²C data register 2				
DR[7:0]	8-bit data register				
		I²C_OAR2	I²C own address register 2		
		ADD2[7:1]	Interface address in dual adr. mode		
		ENDUAL	Dual addressing mode enable		
				I²C_SR1	I²C status register 1
				SMBALERT	SMBus alert
				TIMEOUT	Timeout or Tlow error
				PECERR	PEC error in reception
				OVR	Overrun/Underrun
				AF	Acknowledge failure
				ARL0	Arbitration lost (master)
				BERR	Bus error
				TxE	Data register empty
				RxNE	Data register not empty
				STOPF	Stop detected (slave)
				ADD10	10-bit header sent (master)
				BTB	Byte transfer finished
				ADDR	ADDR sent (master) / matched (slave)
				SB	Start bit (master)

Comparison

UART	SPI	I2C
serial ports (RS-232)	4-wire bus	2-wire bus
TX, RX , opt. control signals	$MOSI, MISO, SCLK, SS$	SCL, SDA
point-to-point	point-to-multipoint	(multi-) point-to-multi-point
full-duplex	full-duplex	half-duplex
asynchronous	synchronous	synchronous
only higher layer addressing	slave selection through \overline{SS} signal	7/10-bit slave address
parity bit possible	no error detection	no error detection
chip-to-chip, PC terminal program	chip-to-chip, on-board connections	chip-to-chip, board-to-board connections

The three interfaces provide the lowest layer of communication and require higher level protocols to provide and interpret the transferred data.

■ **Asynchronous Serial Interface**

- Transmitter and receiver use diverging clocks
- Synchronization using start/stop bits → overhead
- Longer connections require line drivers → RS-232/RS-485

■ **SPI**

- Master/Slave
- Synchronous full-duplex transmission (MOSI, MISO)
- Selection of device through Slave Select (\overline{SS})
- No acknowledge, no error detection
- Four modes → clock polarity and clock phase

■ **I2C**

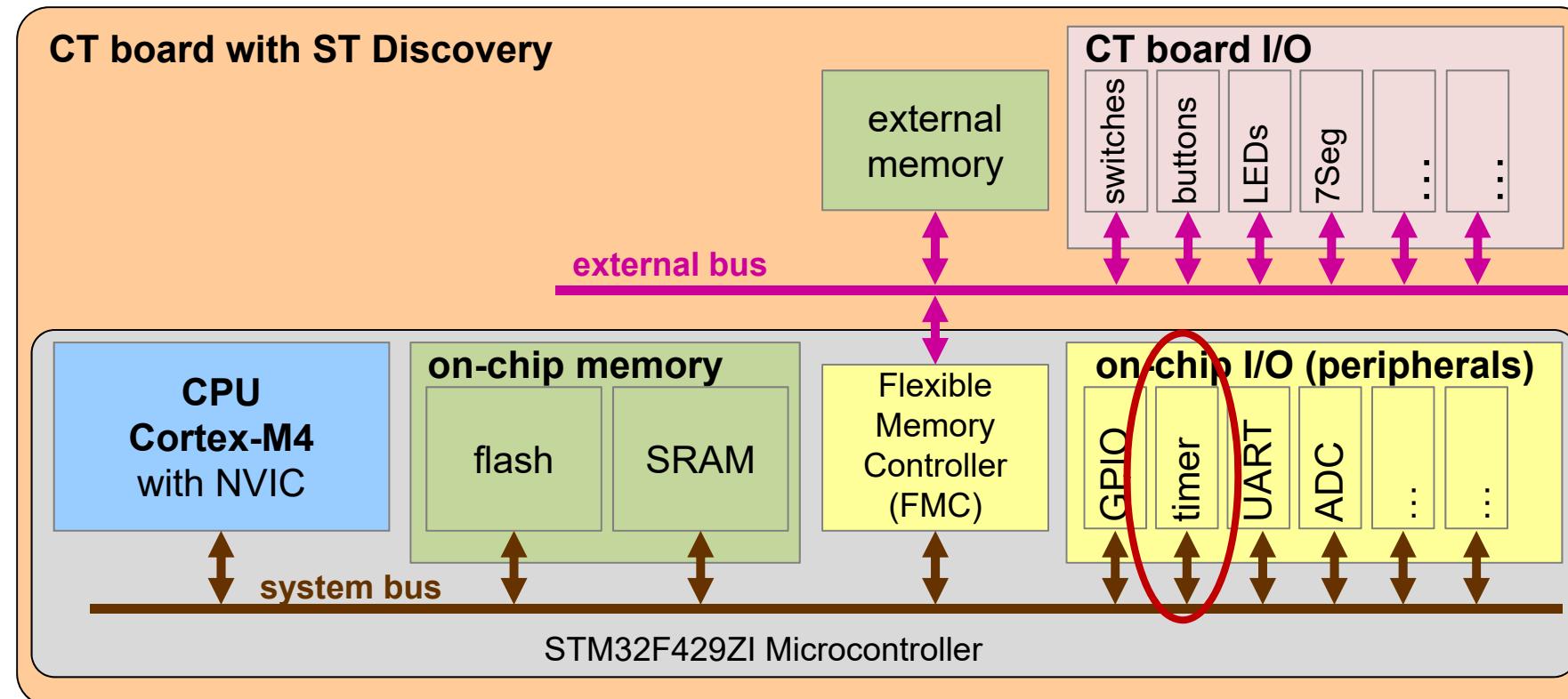
- Synchronous half-duplex transmission (SCL, SDA)
- 7-bit slave addresses

Timer / Counter

Computer Engineering 2

■ Timer / Counter

- Reference Manual pages 576-635



- Timer / Counter – Basic Ideas
- Timers / Counters
- ST32F4xx Timers
- Timer Configuration
- Input Capture
- Pulse-Width-Modulation (PWM)
- Output Compare – Generating PWM Signals
- Capture / Compare Configuration

Learning Objectives

At the end of this lesson you will be able

- to describe the functionality of timers
- to explain the realization of a timer
- to give an overview of timer functions
- to describe the timers of a real microcontroller
- to interpret block diagrams of timers
- to explain the concepts of capture / compare
- to explain the idea of PWM
- to program timers using documents / data sheets

■ **Binary up- or down-counter**

- Counts events / clock pulses or external signals
- Output after a defined number of events (e.g. interrupt)
- Timer: counting clock cycles or processor cycles (periodic)
- Counter: counting events

■ **Use**

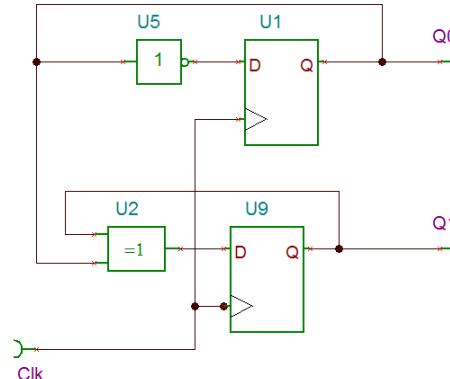
- Count of events
- Measure of time, frequencies, phases, periods
- Generate intervals, row of pulses, interrupts

■ Application examples

- Trigger for periodic software tasks
 - Display refresh
 - Sampling inputs e.g. buttons
- Count number of pulses on input pin
- Measure time between rising edges of an input pin
- Generate defined sequence of pulses on an output pin

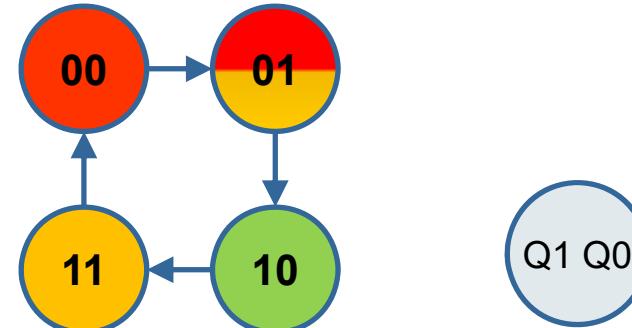
Timer / Counter – Basic Ideas

■ Repetition: 2-bit binary counter



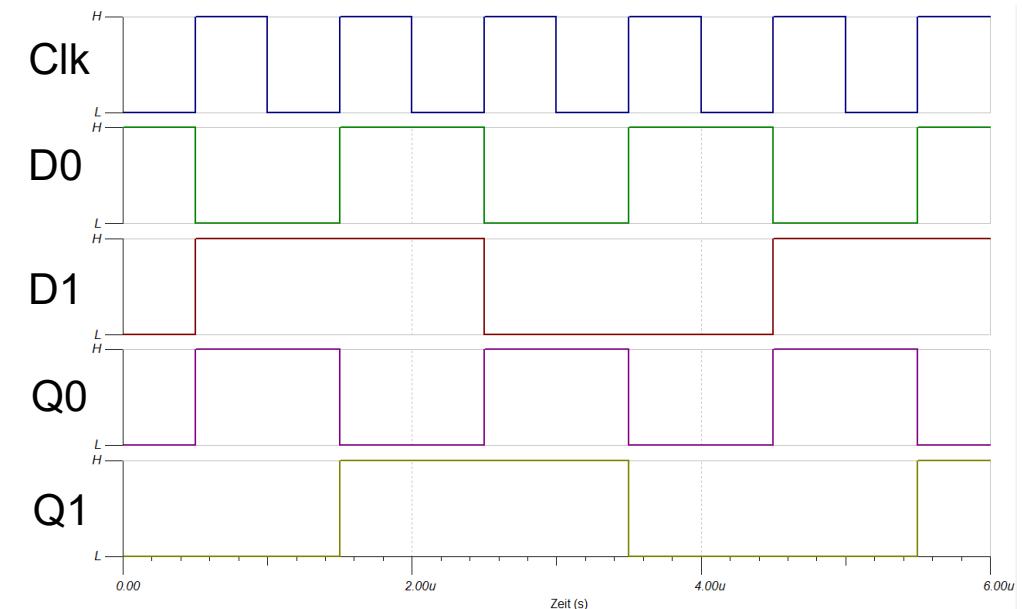
$$\begin{aligned}D1 &= Q0 \oplus Q1 \\D0 &= \overline{Q0}\end{aligned}$$

State diagram



Q1	Q0	D1	D0
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

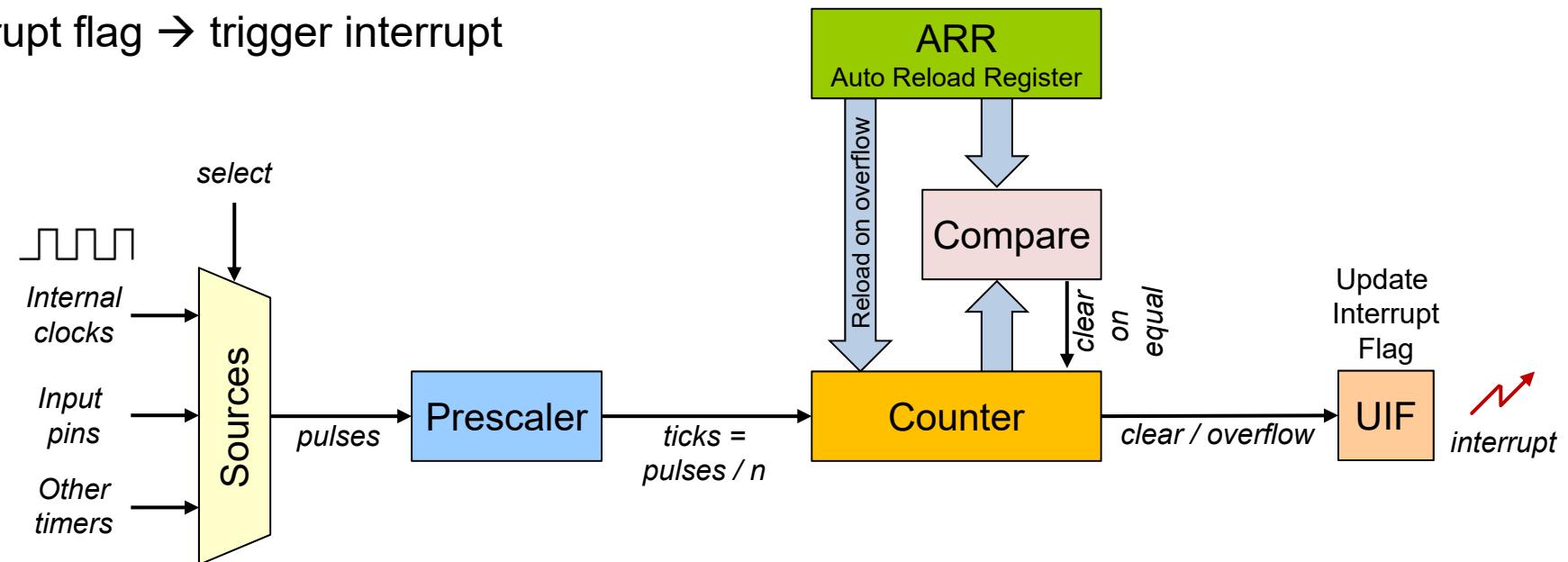
Timing diagram



■ Function

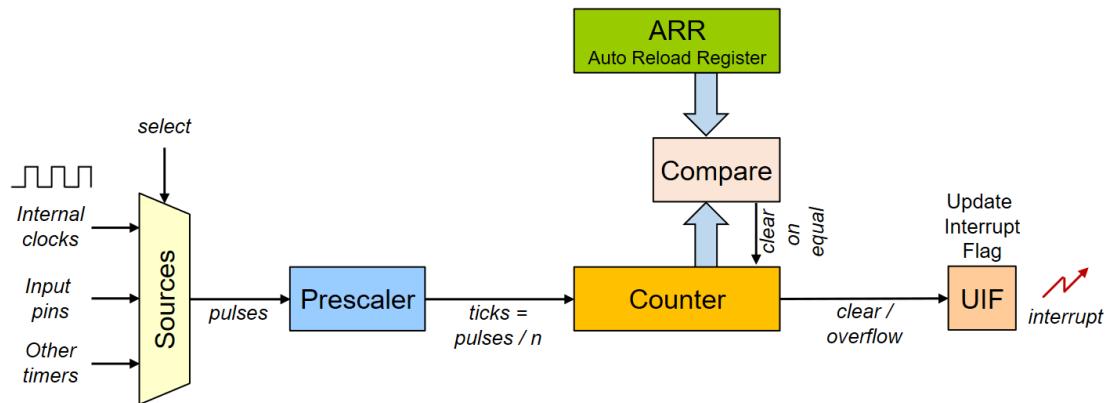
- Configure in up- or down-counting mode
- Select source
- 16-bit / 32-bit counter register
 - Increment / decrement at every tick
- Set interrupt flag → trigger interrupt

16-bit counter	0, 1, 2,	65'535
32-bit counter	0, 1, 2,	4'294'967'295



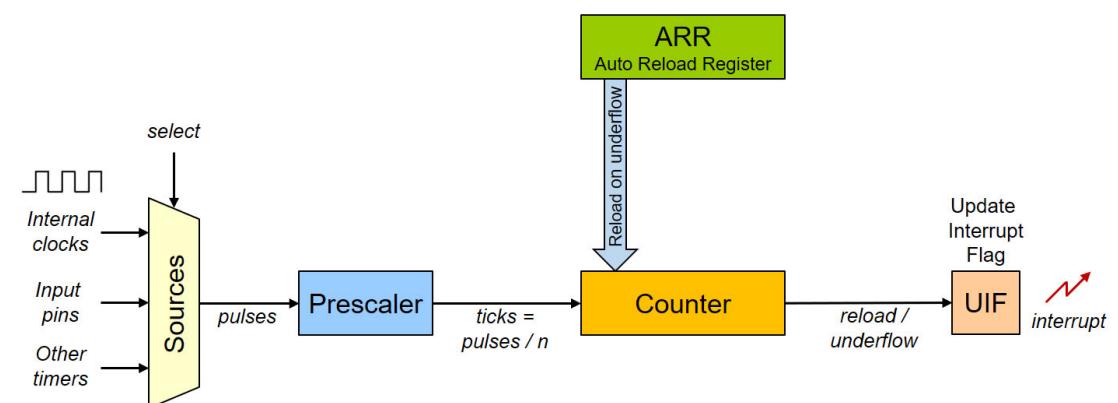
Up-counting mode

- Counts from 0 to the auto-reload value (content of ARR)
- Restarts from 0
- Generates a counter overflow event



Down-counting mode

- Counts from auto-reload value (content of ARR) down to 0
- Restarts from auto-reload value
- Generates a counter underflow event

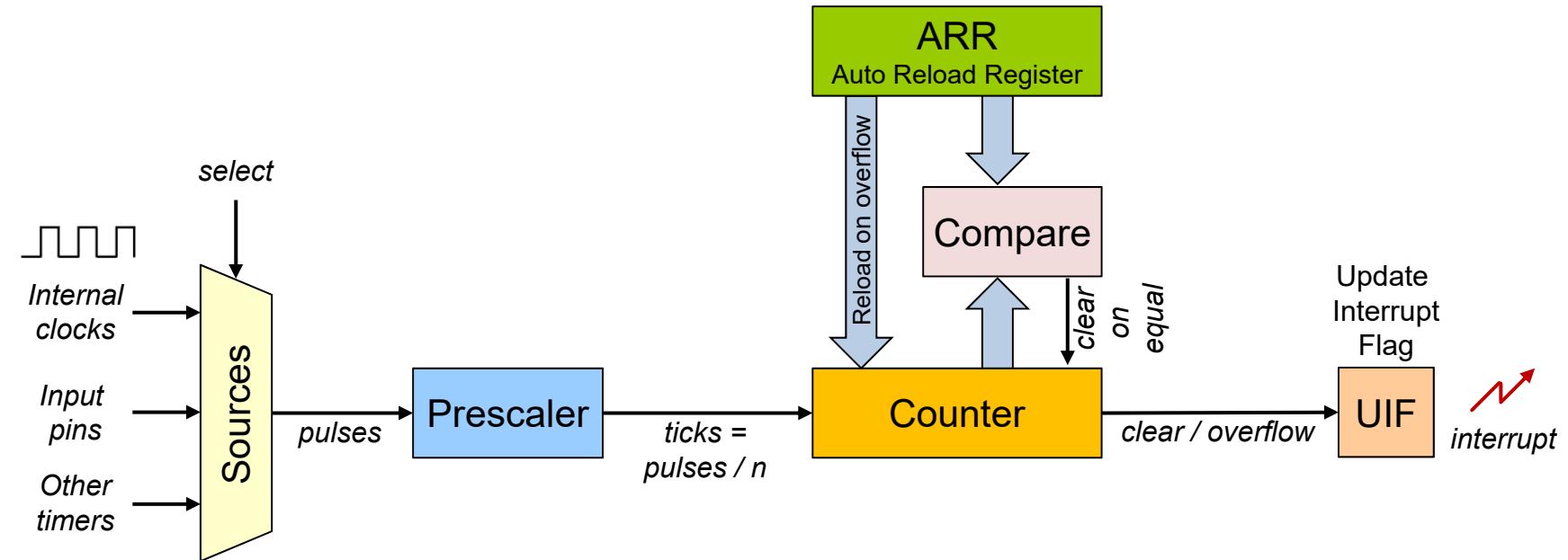


■ Prescaler

- Increase counting range
- Count only every n-th event
 - e.g. $n = \{1, 2, 4, 8, 32, 64, \dots\}$

Example: 16-bit counter

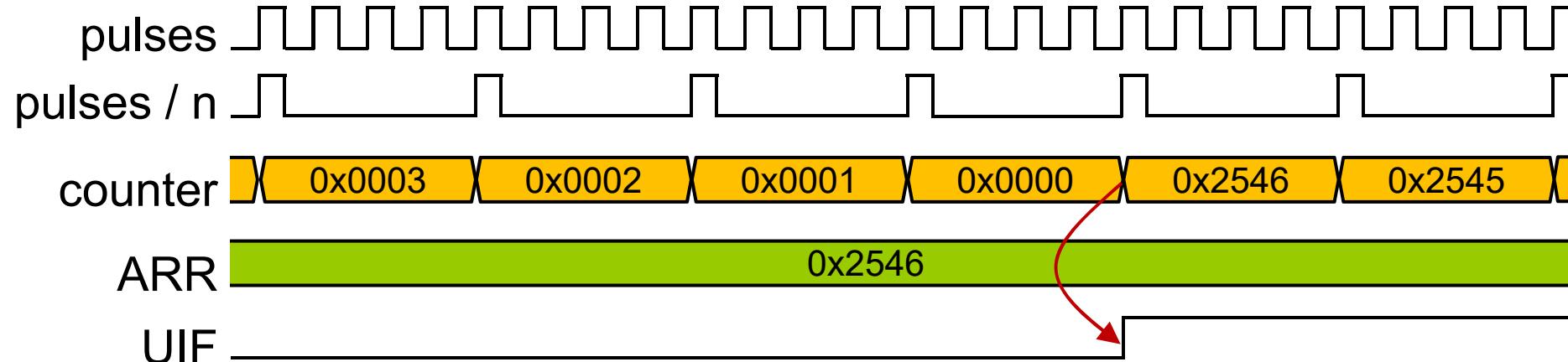
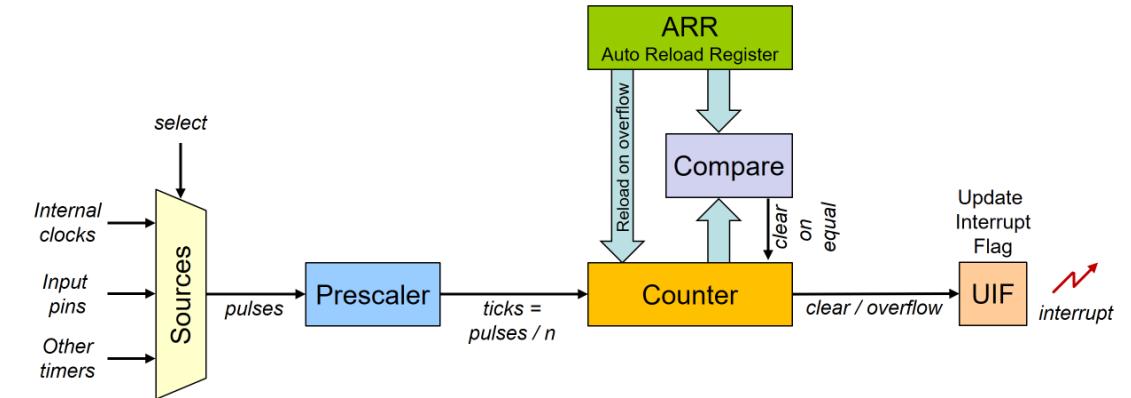
$$\begin{aligned}
 \text{Source } 100 \text{ MHz} &\rightarrow \text{period } T = 1 / (100 \text{ MHz}) = 0.01 \text{ us} \\
 \text{Prescaler} = 1 &\rightarrow 65'536 \cong 65'536 * 0.01 \text{ us} = 655.36 \text{ us} \\
 \text{Prescaler} = 1'000 &\rightarrow 65'536 \cong 65'536 * 0.01 \text{ ms} = 655.36 \text{ ms}
 \end{aligned}$$



Timers / Counters

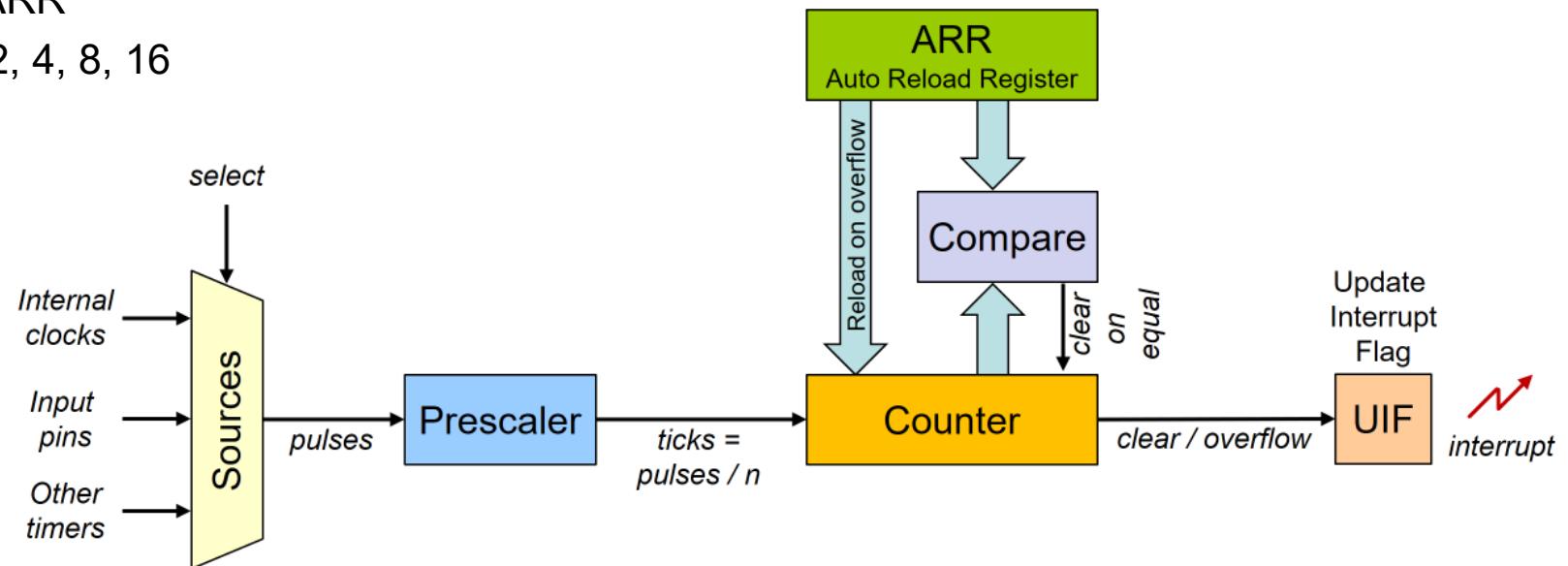
■ Down-counting example

- Prescaler → divide by 4
 - Count down to zero
 - Set interrupt request (UIF)
 - Restart from value in ARR



■ Exercise

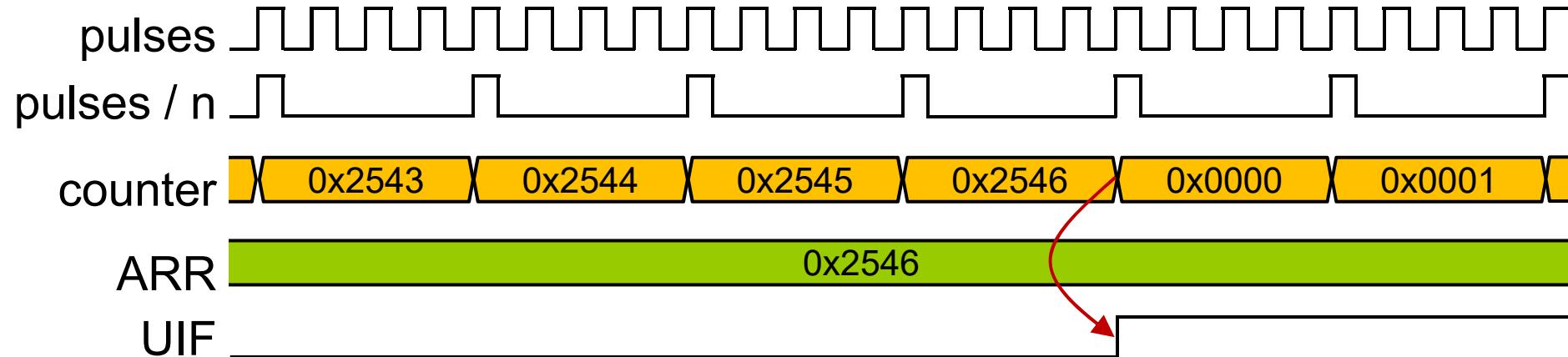
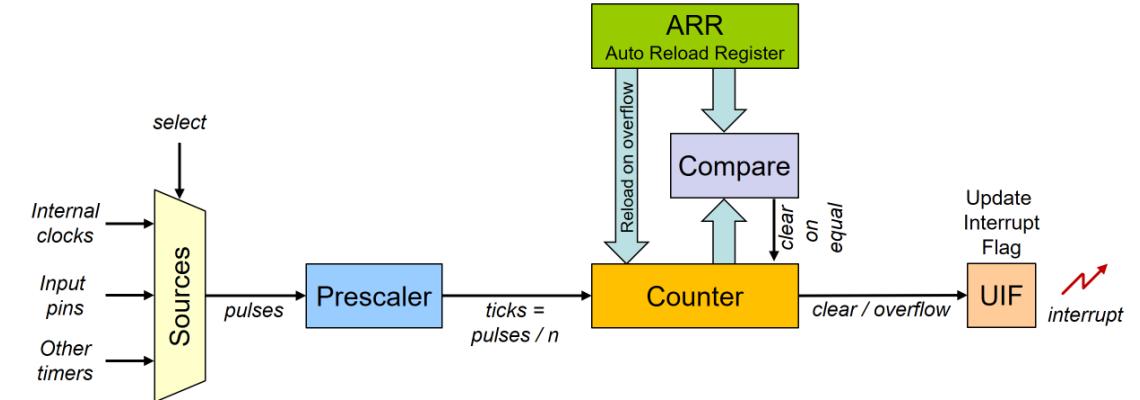
- Source: 1 MHz
- What needs to be set, when we want an interrupt every
 - 50 ms → 20 Hz
 - 1 s → 1 Hz
- Assume
 - **16-bit** counter / ARR
 - Prescaler $n = 1, 2, 4, 8, 16$
 - Down-counter



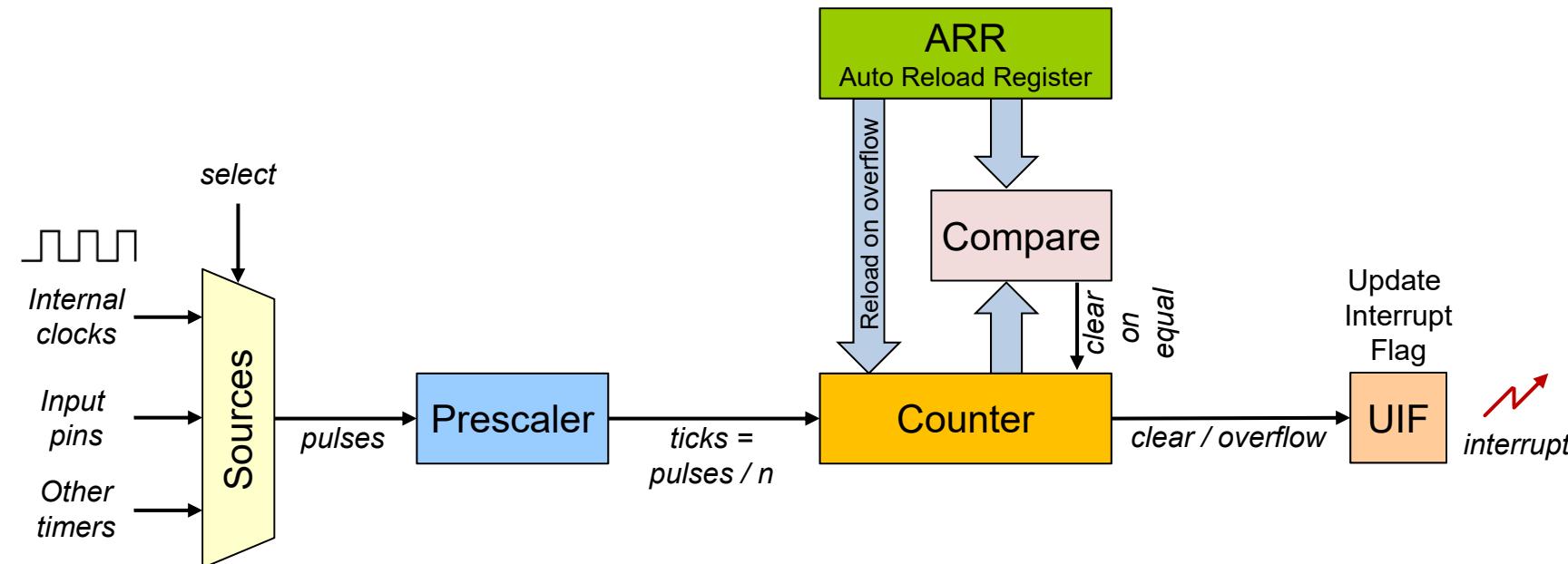
Timers / Counters

■ Up-counting example

- Prescaler → divide by 4
- Count up to the value in reload register
 - Set interrupt request
 - Restart from 0



For a given problem there are often different ways to use the available hardware.
→ E.g. up-counter vs. down-counter.



- **Full-featured general-purpose timers**
 - TIM2, TIM3, TIM4, TIM5
- **General-purpose timers**
 - TIM9, TIM10, TIM11, TIM12, TIM13, and TIM14
- **Advanced-control timers**
 - TIM1, TIM8
- **Basic timers**
 - TIM6, TIM7

*Explained
Here: TIM 2*

*Not explained in detail
see reference manual*

■ Timers TIM2 - TIM5

- 16-bit → TIM3 and TIM4
 - Up, down, up/down
 - Auto-reload
- 16-bit programmable prescaler
 - Dividing counter clock frequency by factor between 1 and 65536
- Up to 4 independent channels for:
 - Input capture
 - Output compare
 - PWM generation
 - One-pulse mode output
- Synchronization circuit
 - Control timer with external signals
 - Interconnect several timers together
- Interrupt/DMA generation based on several events

■ Register address = Base address + Offset

- Offset address is given for each register in Reference Manual
- Base address defined in memory map
→ Reference Manual

Boundary address	Peripheral
0x4000 0C00 - 0x4000 0FFF	TIM5
0x4000 0800 - 0x4000 0BFF	TIM4
0x4000 0400 - 0x4000 07FF	TIM3
0x4000 0000 - 0x4000 03FF	TIM2
0x4002 3800 - 0x4002 3BFF	RCC

RCC: Reset and Clock Configuration

■ Enable timer block

- RCC APB1 peripheral clock enable register (RCC_APB1ENR)
- RCC = Reset and Clock Control

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
UART8 EN	UART7 EN	DAC EN	PWR EN	Reser-ved	CAN2 EN	CAN1 EN	Reser-ved	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART 3 EN	USART 2 EN	Reser-ved
rw	rw	rw	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw	
15	14	13	12		10	9		8	7	6	5	4	3	2	1
SPI3 EN	SPI2 EN	Reserved		WWDG EN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	

Reference Manual

Timer Configuration → Selected Registers

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
0x00	TIMx_CR1	Reserved												CKD [1:0]		ARPE		CMS [1:0]		DIR		OPM		URS		UDIS		CEN													
	Reset value													0 0		0 0		0 0		0 0		0 0		0 0		0 0		0 0													
0x08	TIMx_SMCR	Reserved												ETP	ECE	ETPS [1:0]	ETF[3:0]				MSM	TS[2:0]			SMS[2:0]																
	Reset value														0	0	0																								
0x0C	TIMx_DIER	Enable Interrupts												CTDE	COMDE	CC1DE					0	CC4IE																			
	Reset value														0	0	0						CC3IE																		
0x10	TIMx_SR	Interrupt Flags												CC2OF	CC3OF	CC4OF					0	CC2IF																			
	Reset value														0	0	0						CC3IF																		
0x14	TIMx_EGR	Reserved												TG	TIF	UDE					0	CC4IF																			
	Reset value														0	0	0						CC3IF																		
0x24	TIMx_CNT	CNT[31:16] (TIM2 and TIM5 only, reserved on the other timers)												CNT[15:0]																											
	Reset value	0 0													0 0																										
0x28	TIMx_PSC	Reserved												PSC[15:0]																											
	Reset value													0 0																											
0x2C	TIMx_ARR	ARR[31:16] (TIM2 and TIM5 only, reserved on the other timers)												ARR[15:0]																											
	Reset value	0 0													0 0																										

CR1 – Control Register 1

CMS Center-aligned Mode Selection
DIR Direction (0: up; 1: down)
CEN Counter Enable

SMCR – Slave Mode Control Register

SMS Slave Mode Selection
Usually keep SMS = 000

DIER – DMA/Interrupt Enable Register

UIE Update Interrupt Enable
CCsIE see later

SR – Status Register

UIF Update Interrupt Flag
 Set by HW, cleared by SW
CCxIF see later

EGR – Event Generation Register

UG Update Generation
SW can reinitialize counter and update registers

CNT – Counter

PSC – Prescaler

ARR = Auto Reload Register

Timer Configuration → Selected Registers

```
#define TIM2 ( (reg_tim_t *) 0x40000000 )
#define TIM3 ( (reg_tim_t *) 0x40000400 )
#define TIM4 ( (reg_tim_t *) 0x40000800 )
#define TIM5 ( (reg_tim_t *) 0x40000c00 )
```

```
typedef struct {
    volatile uint32_t CR1;
    volatile uint32_t CR2;
    volatile uint32_t SMCR;
    volatile uint32_t DIER;
    volatile uint32_t SR;
    volatile uint32_t EGR;
    volatile uint32_t CCMR1;
    volatile uint32_t CCMR2;
    volatile uint32_t CCER;
    volatile uint32_t CNT;
    volatile uint32_t PSC;
    volatile uint32_t ARR;
    volatile uint32_t RCR;
    volatile uint32_t CCR1;
    volatile uint32_t CCR2;
    volatile uint32_t CCR3;
    volatile uint32_t CCR4;
    volatile uint32_t BDTR;
    volatile uint32_t DCR;
    volatile uint32_t DMAR;
    volatile uint32_t OR;
} reg_tim_t;
```

Example

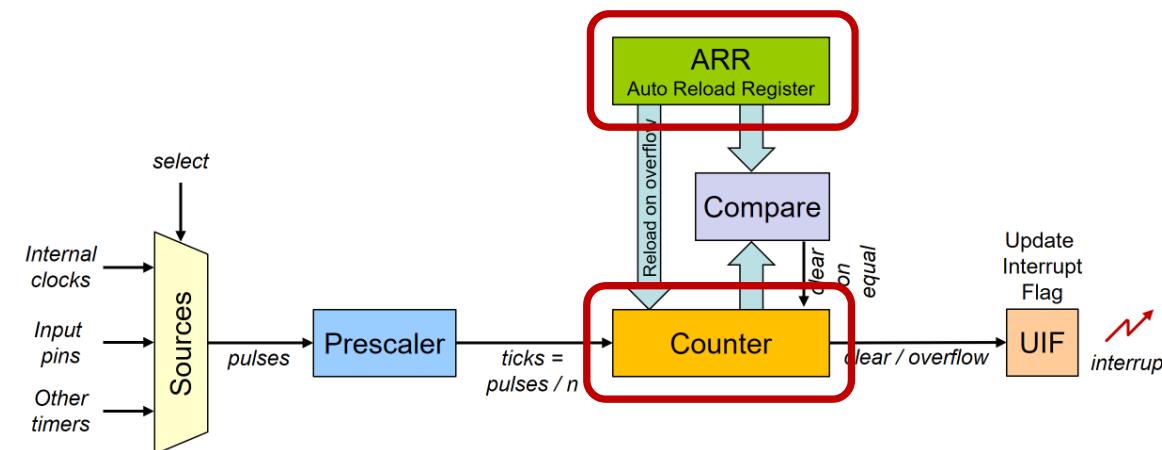
■ TIMx counter (**TIMx_CNT**)

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

■ TIMx auto-reload register (**TIMx_ARR**)

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

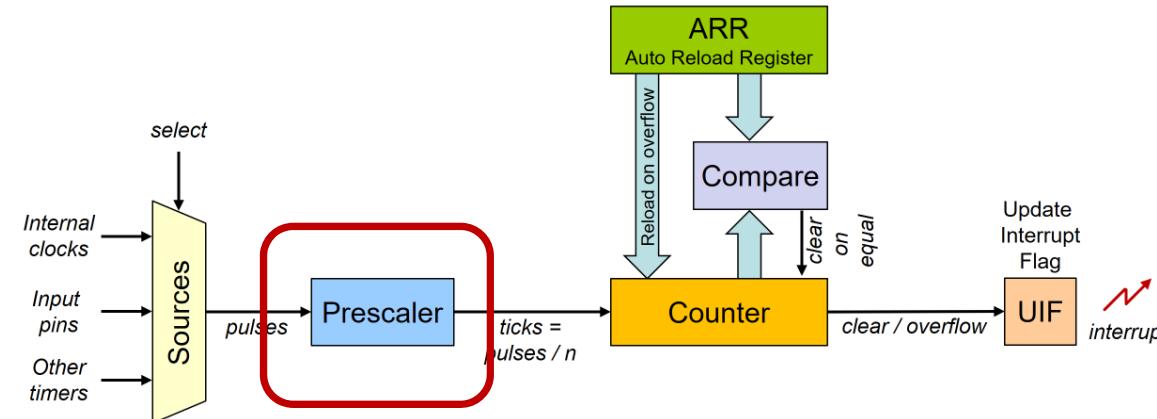
- ARR is the value to be loaded in the actual auto-reload register



■ TIMx prescaler (TIMx_PSC)

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

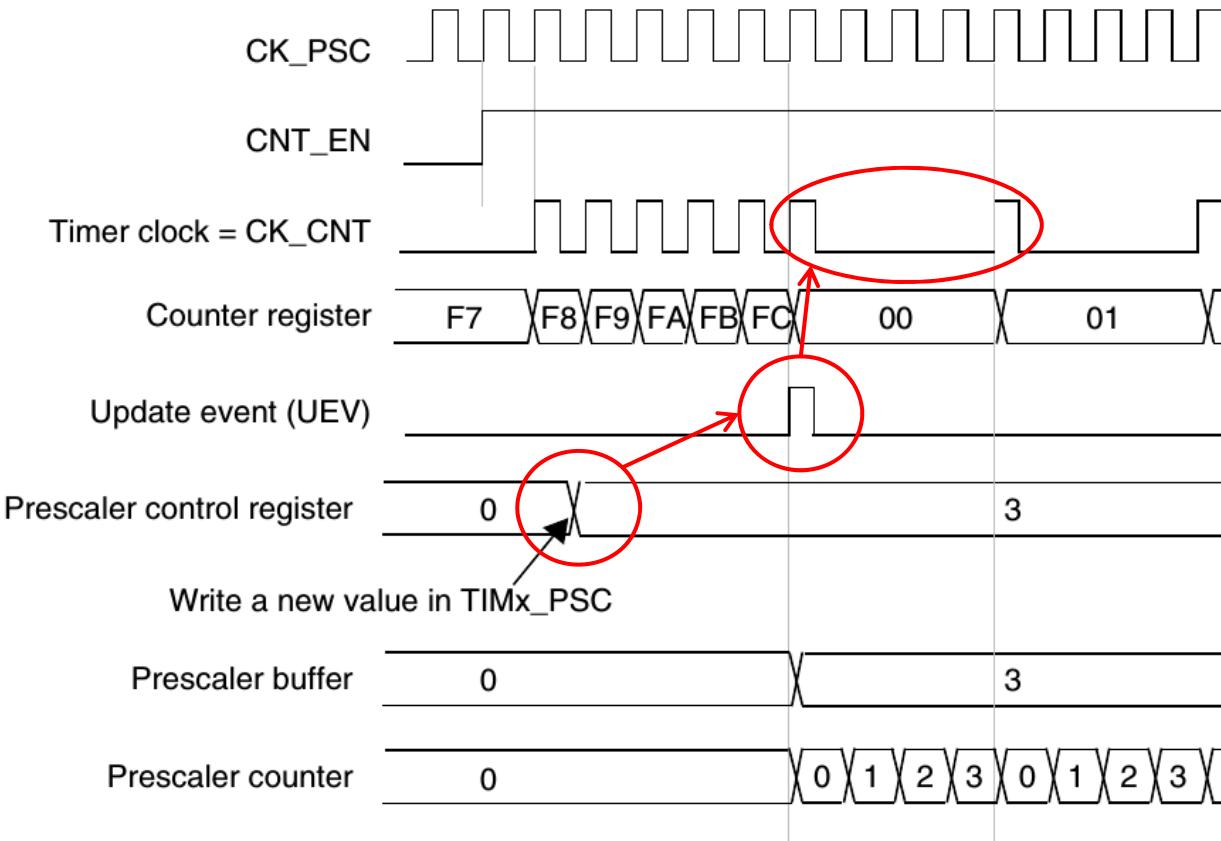
- Divides counter clock frequency by a factor between 1 and 65536
- Clock frequency CK_CNT equal to $f_{CK_PSC} / (PSC[15:0] + 1)$
- TIMx_PSC can be changed on the fly
 - Reason: TIMx_PSC is buffered → see next slide



■ Prescaler

- Example: Changing prescaler division from 1 to 4

Example
 $\text{TIMx_ARR}=0x00FC$



■ TIMx Control Register 1 (**TIMx CR1**)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				CKD[1:0]		ARPE	CMS		DIR	OPM		URS	UDIS		CEN
				rw	rw	rw	rw	rw	rw			rw			rw

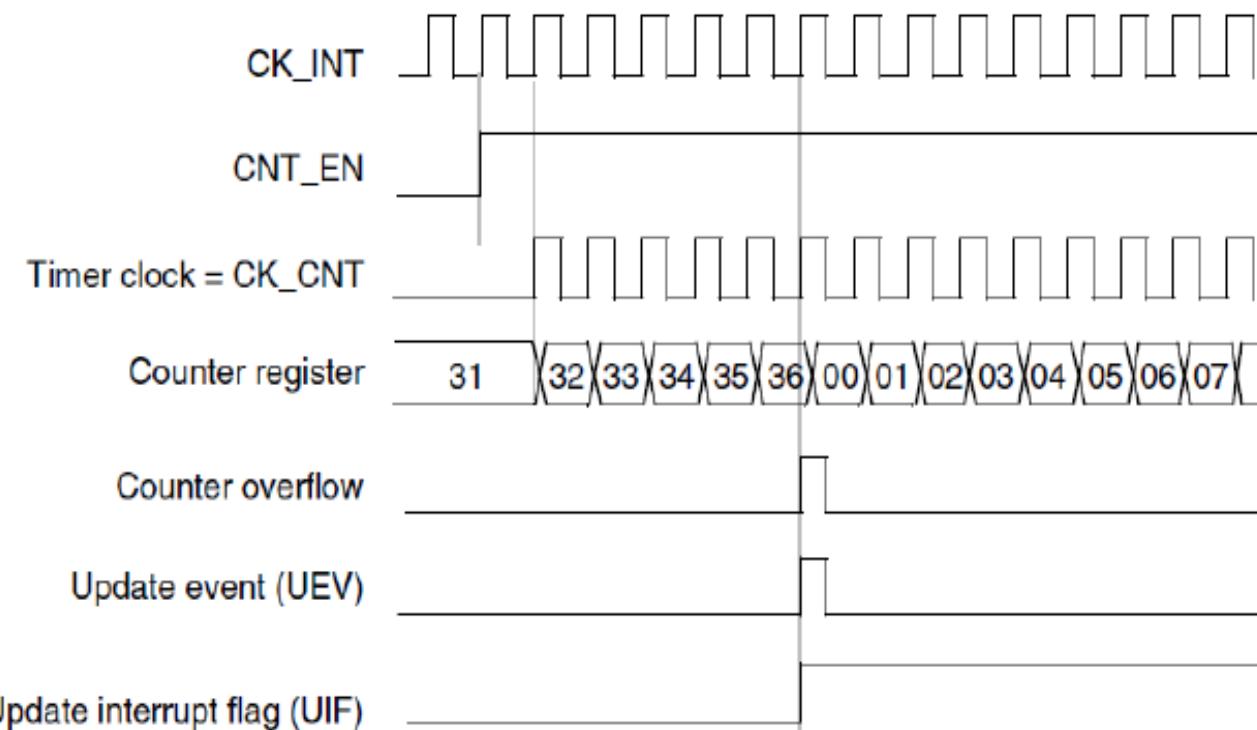
- CMS Center-aligned mode selection
 - 00 count up or down depending on DIR
 - others center-aligned
- DIR Direction
 - 0 up-counter
 - 1 down-counter
- CEN Counter enable
- *Other settings for advanced use -> keep at default values*

■ Up-counting mode

- Counting from 0 to auto-reload value (TIMx_ARR)
- Generates a counter overflow event
- Restarts from 0

Example
 $\text{TIMx_ARR}=0x0036$

Division by 1

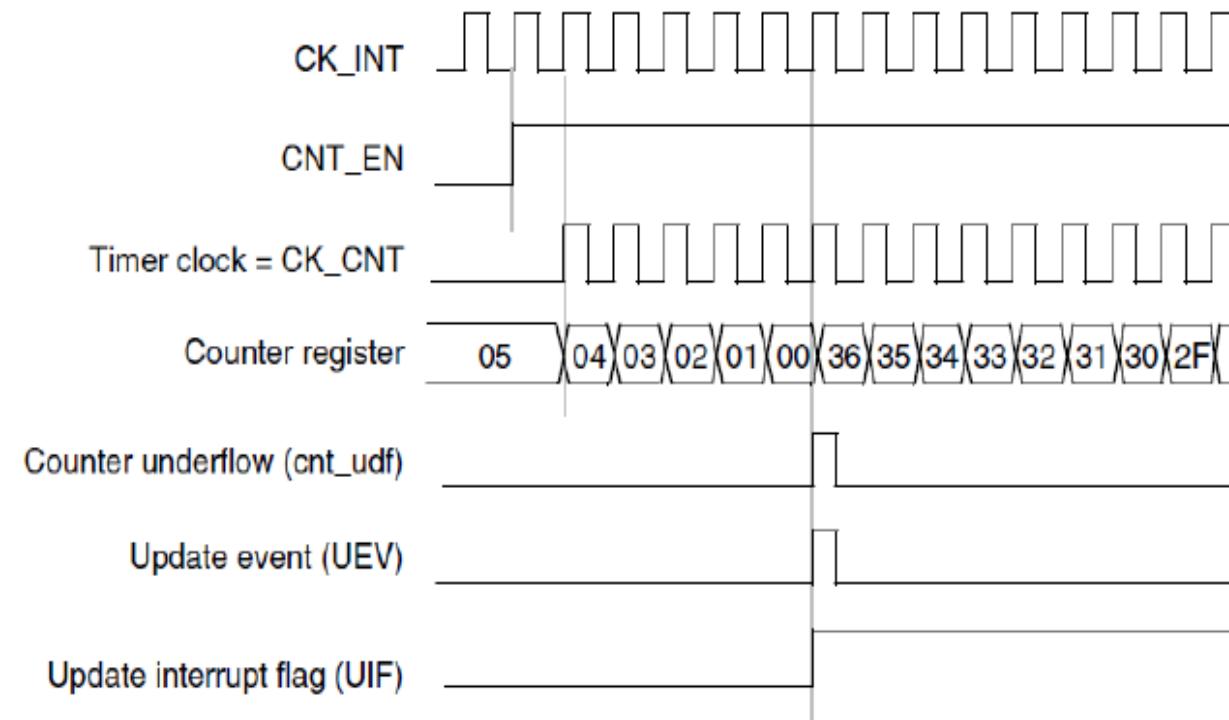


■ Down-counting mode

- Counting from auto-reload value (TIMx_ARR) down to 0
- Generates a counter underflow event
- Restarts from auto-reload value

Example
 $\text{TIMx_ARR}=0x0036$

Division by 1

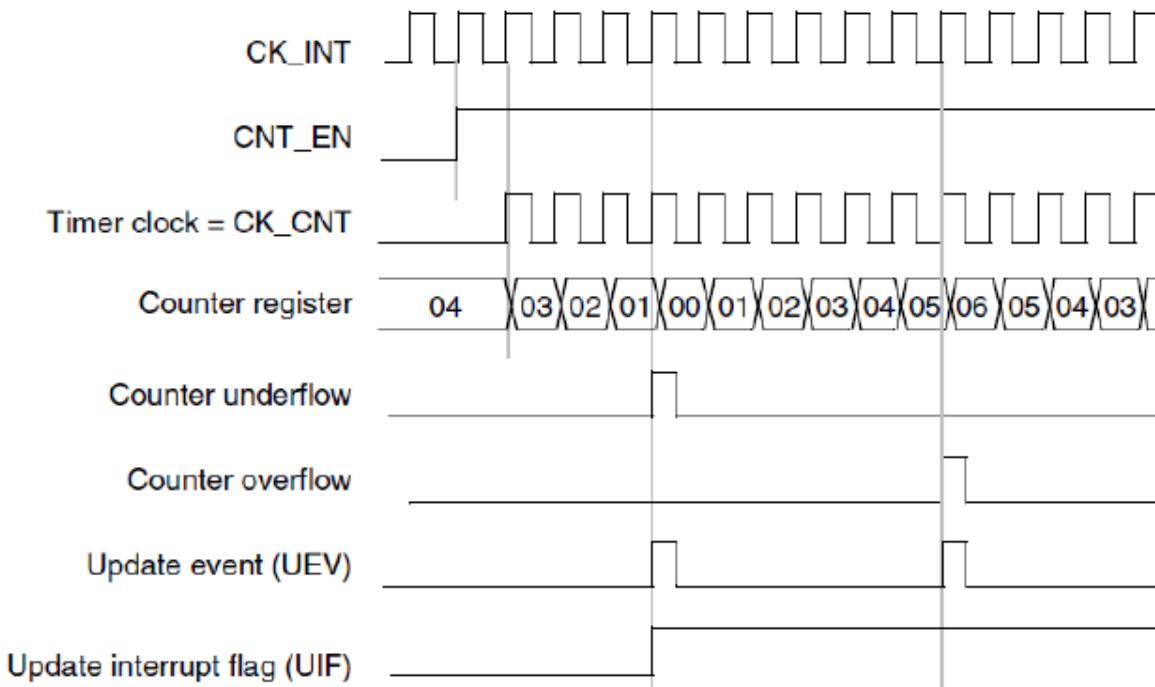


■ Center-aligned mode (up/down counting)

- Counts from 0 to auto-reload value ($\text{TIMx_ARR} - 1$)
- Generates a counter overflow event
- Counts from auto-reload value down to 1
- Generates a counter underflow event
- Restarts counting from 0

Example
 $\text{TIMx_ARR}=0x0006$

Division by 1



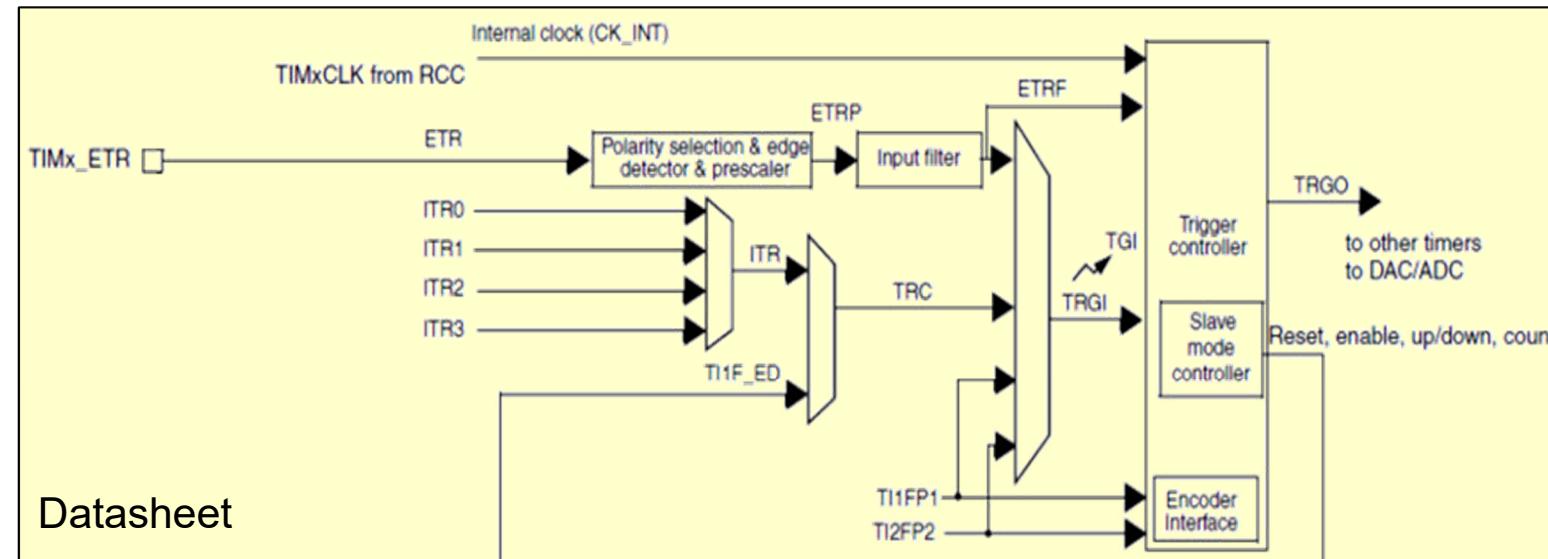
■ TIMx slave mode control register (**TIMx_SMCR**)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			Res.	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- This register is used to select the various clock and trigger sources → see next page
- SMS: Master/Slave mode
 - 000: Slave Mode off -> using internal clock (CK_INT)
- *Other settings for advanced use -> keep at default value*

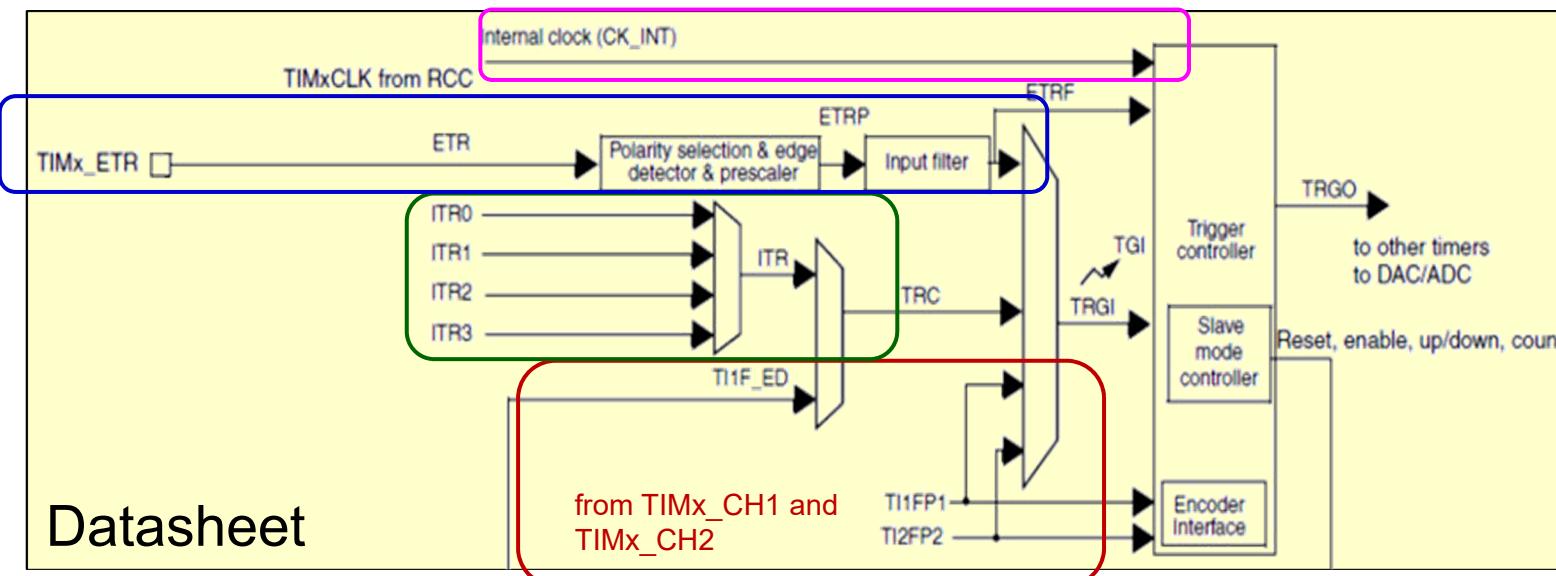
■ Clock sources for TIM2-TIM5

- Internal clock (**CK_INT**)
- External input pins (**TIMx_CH1** and **TIMx_CH2**)
- External trigger input (**TIMx_ETR**)
- Internal trigger inputs (**ITRx**)
 - Using one timer as prescaler for another timer



■ Clock sources for TIM2-TIM5

- Internal clock (**CK_INT**)
- External input pins (**TIMx_CH1** and **TIMx_CH2**)
- External trigger input (**TIMx_ETR**)
- Internal trigger inputs (**ITRx**)
 - Using one timer as prescaler for another timer



■ Given

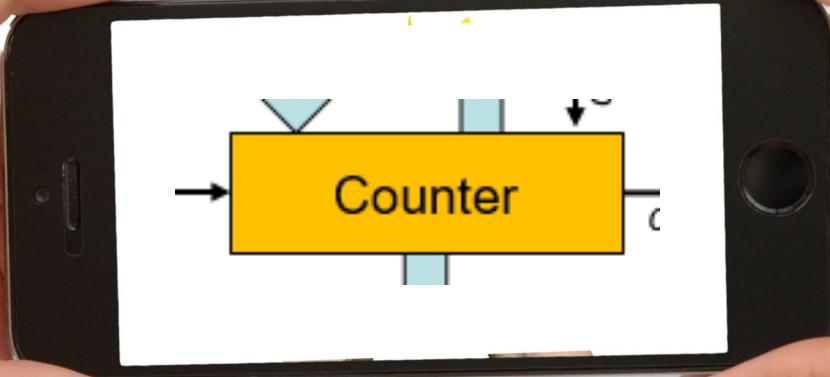
- CK_INT is already configured to 84 MHz

■ Task

- Generate an interrupt every 1 s
- Use Timer 3 (16-bit) in up-counting mode

■ Wanted

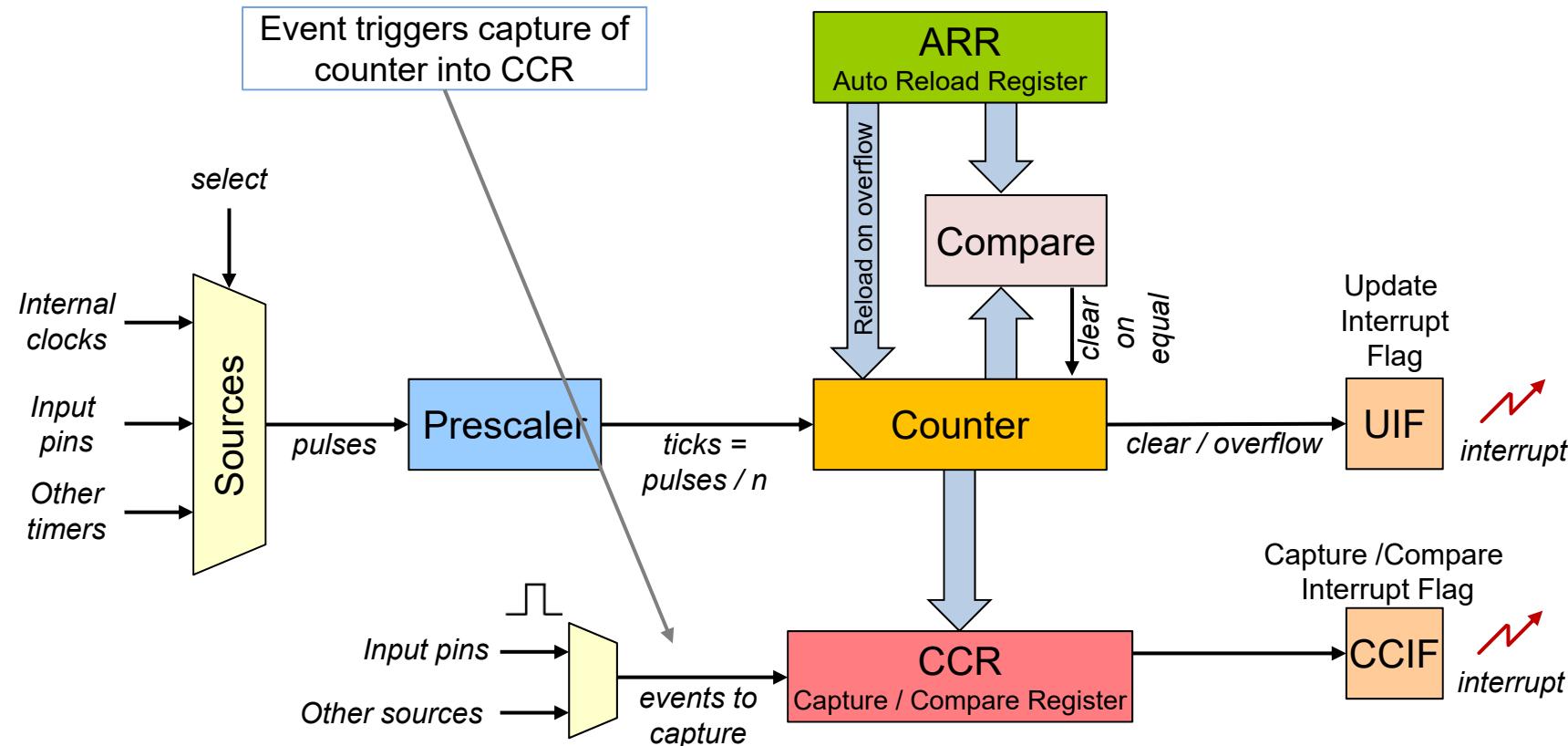
- Names and addresses of configuration registers
- Settings for configuration registers



Input Capture

■ Measuring intervals → pulse lengths and periods

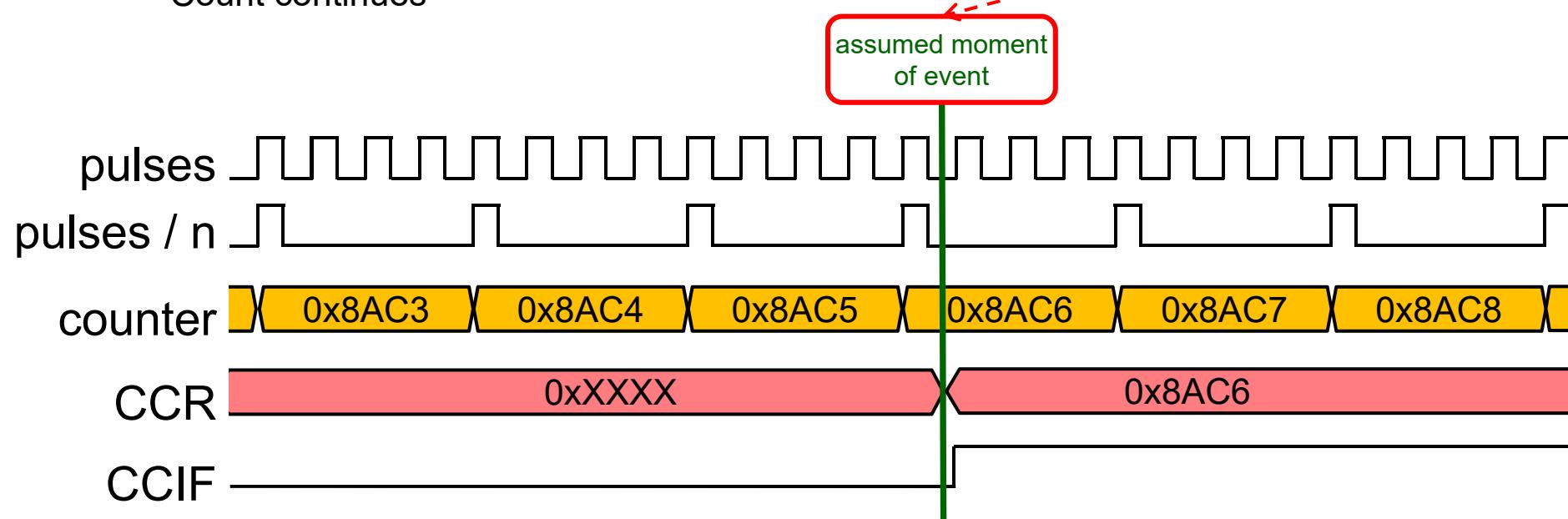
- Count ticks between timer start and an event



Input Capture

■ Capture example

- Stop watch
- At which moment in time does the user push the button?
 - Event = rising edge on input pin
 - Time of event is captured
 - Count continues

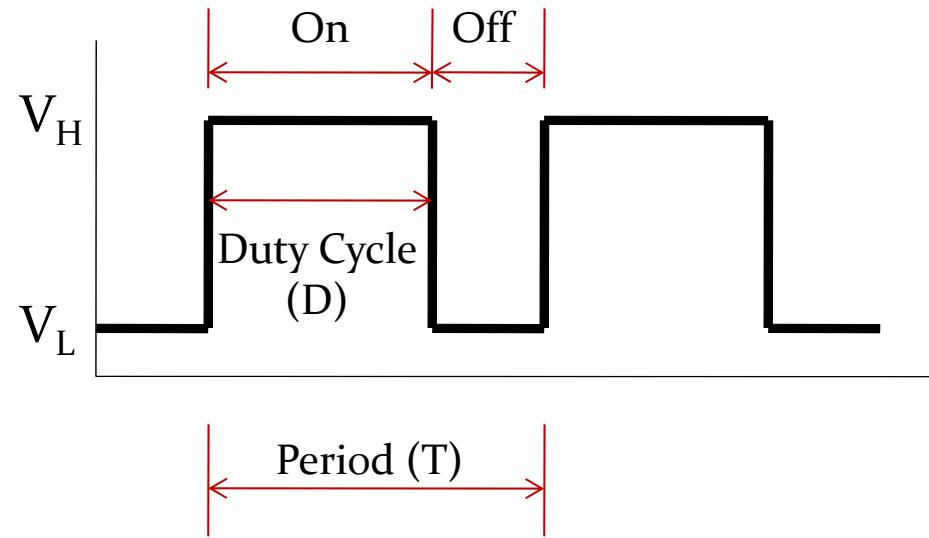




Pulse Width Modulation (PWM)

Pulse-Width-Modulation (PWM)

■ Duty Cycle – Definition



$$\text{Duty Cycle} = \frac{\text{On Time}}{\text{Period}} \times 100\%$$

Average signal

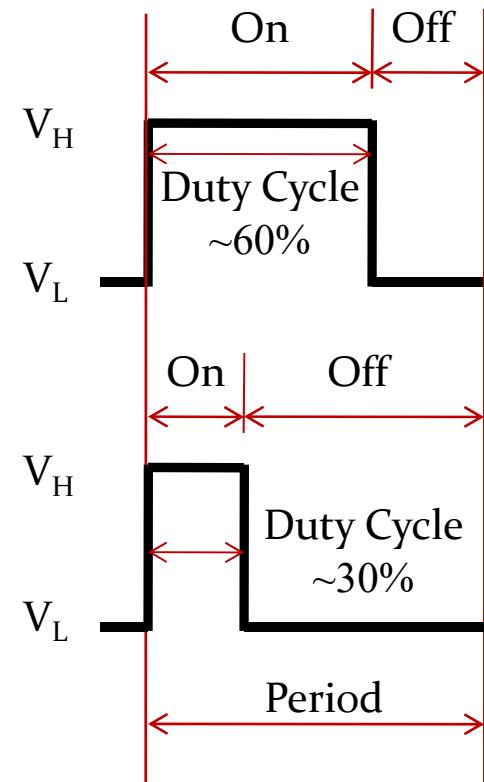
$$V_{avg} = D \cdot V_H + (1 - D) \cdot V_L$$

Usually, V_L is taken as zero volts for simplicity.

Pulse-Width-Modulation (PWM)

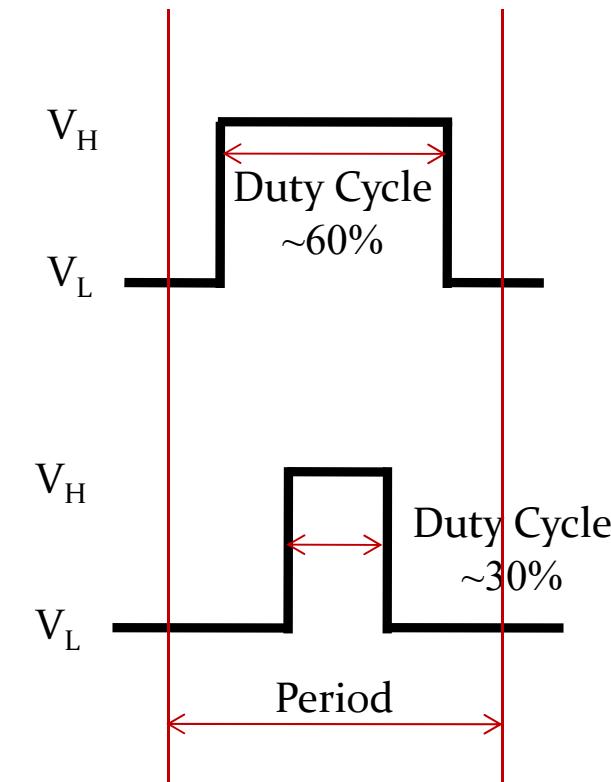
Left Aligned

- Left edge fixed, trailing edge modulated



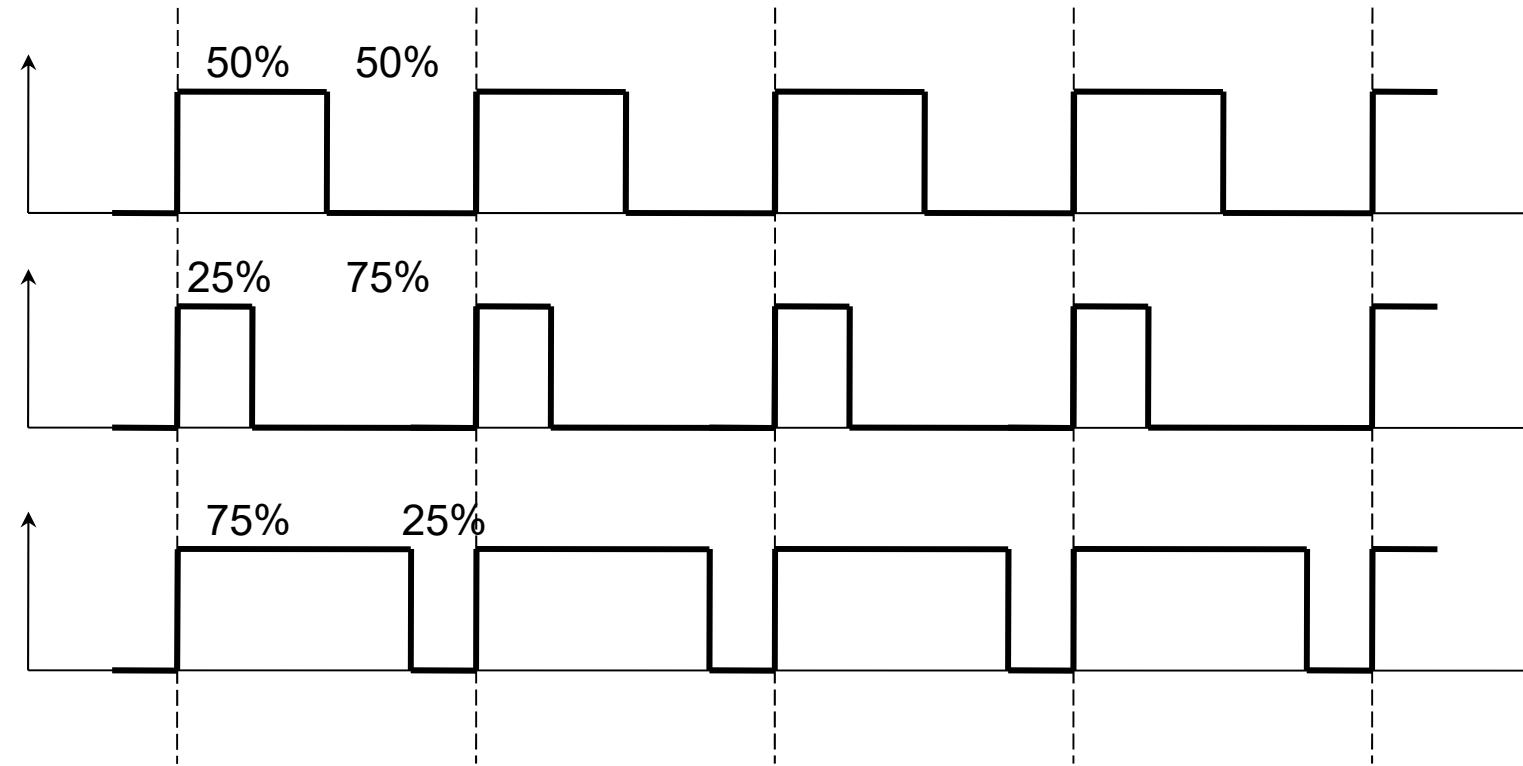
Center Aligned

- Center of signal fixed, both edges modulated



Pulse-Width-Modulation (PWM)

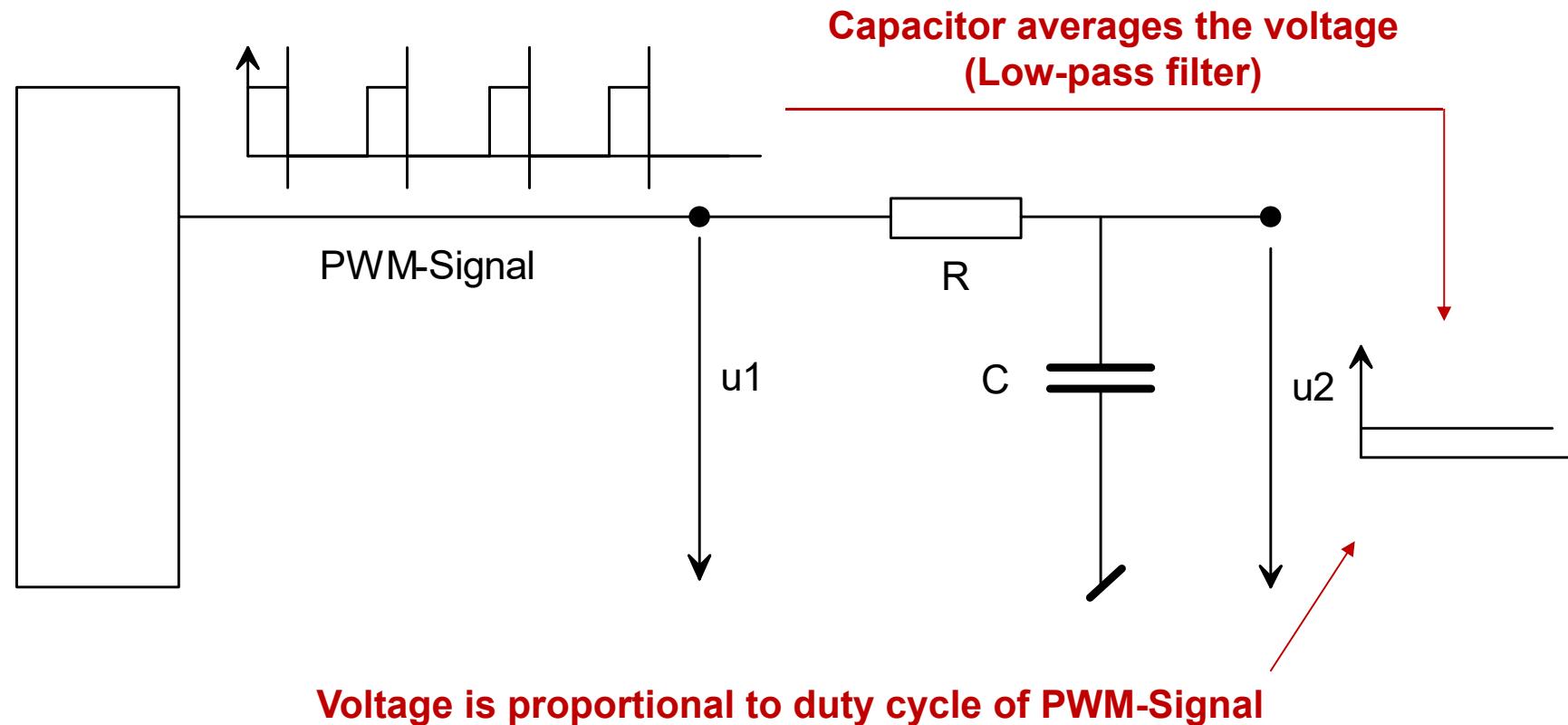
- PWM-Signals are digital signals (0/1) with a defined frequency and variable pulse width



Pulse-Width-Modulation (PWM)

■ Application

- Dimming LED with variable on / off
- Digital/Analog-Converters (DAC)



Pulse-Width-Modulation (PWM)

■ Sine wave approximation

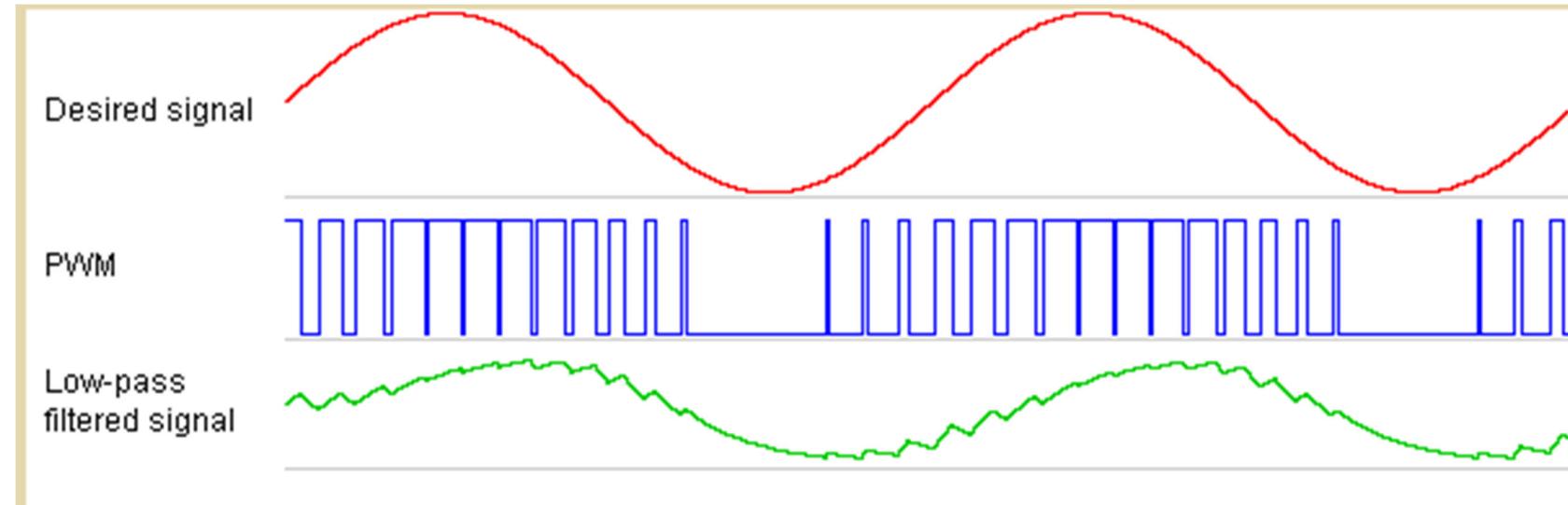
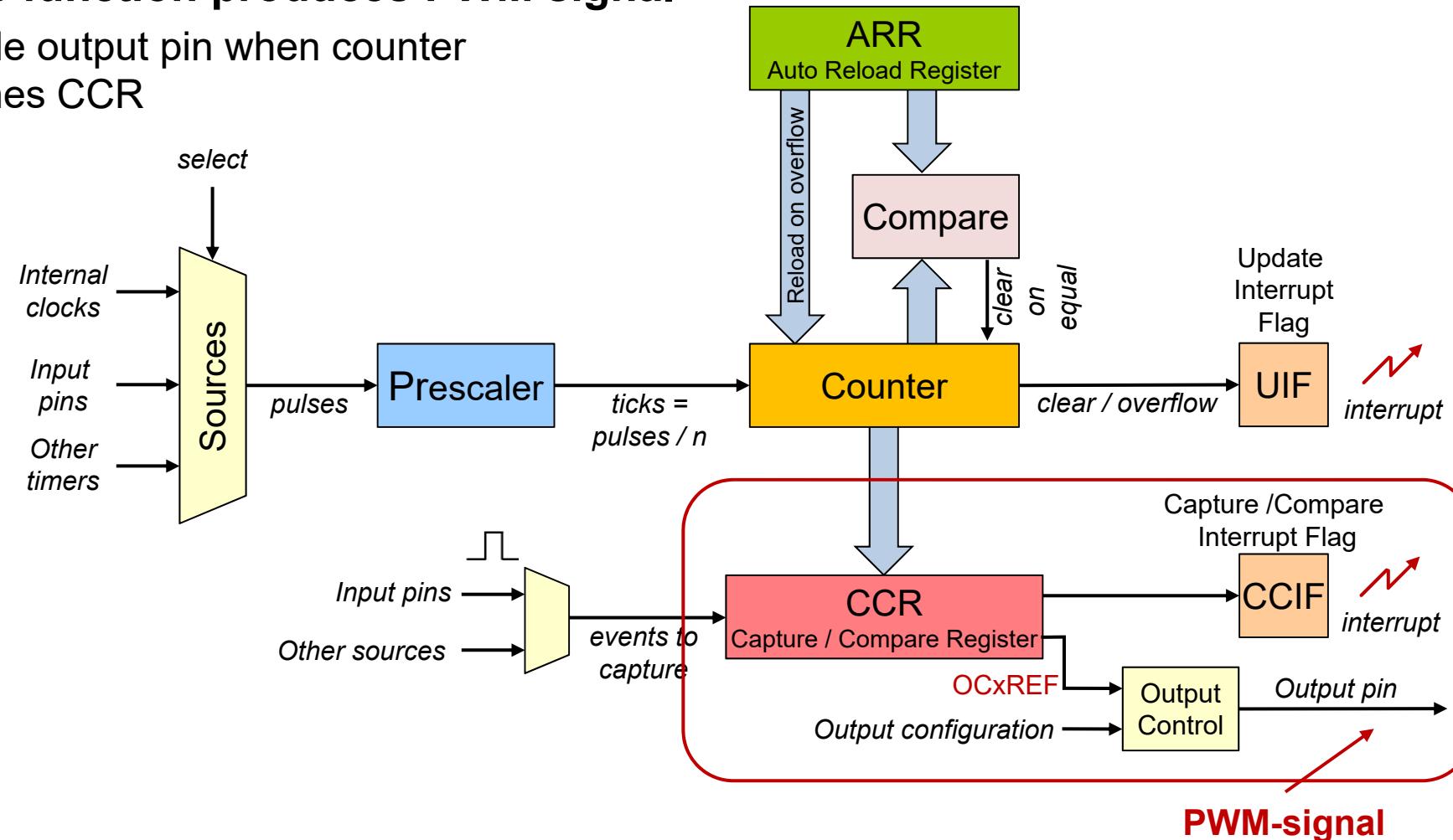


Image: Zak Ahmad

Output Compare – Generating PWM Signals

■ Compare function produces PWM signal

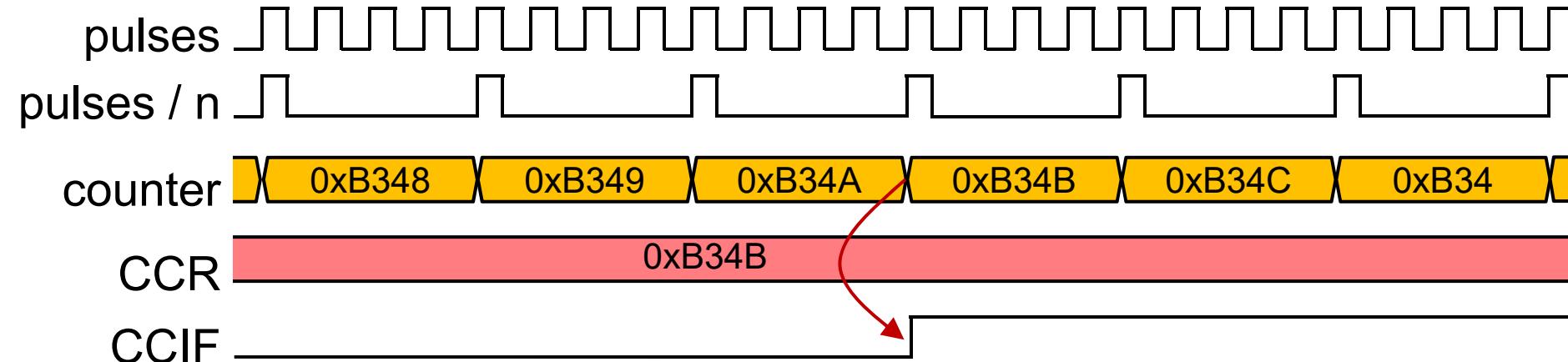
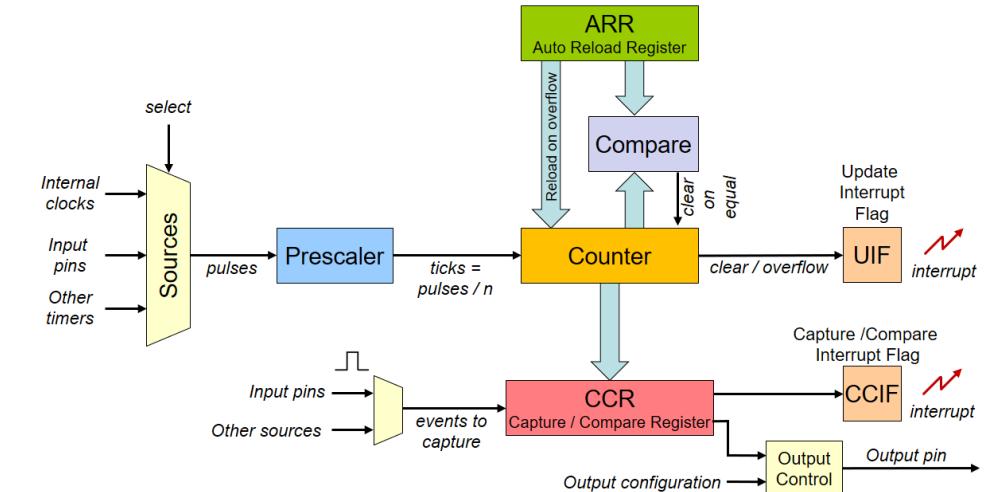
- Toggle output pin when counter reaches CCR



Output Compare – Generating PWM Signals

■ Compare example

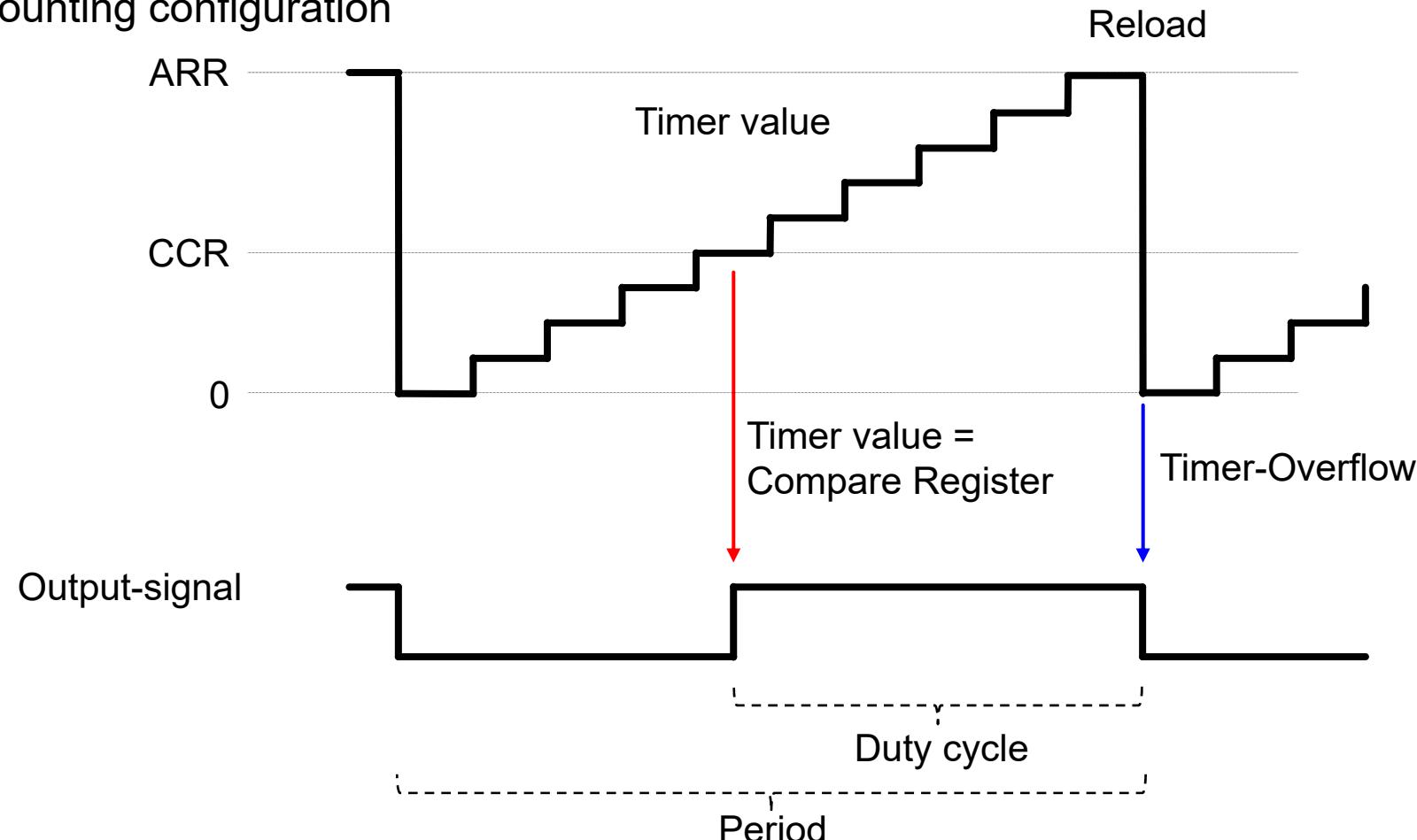
- Raise an alarm when specified count is reached or exceeded
- Continuously compare counter value to a reference value



Output Compare – Generating PWM Signals

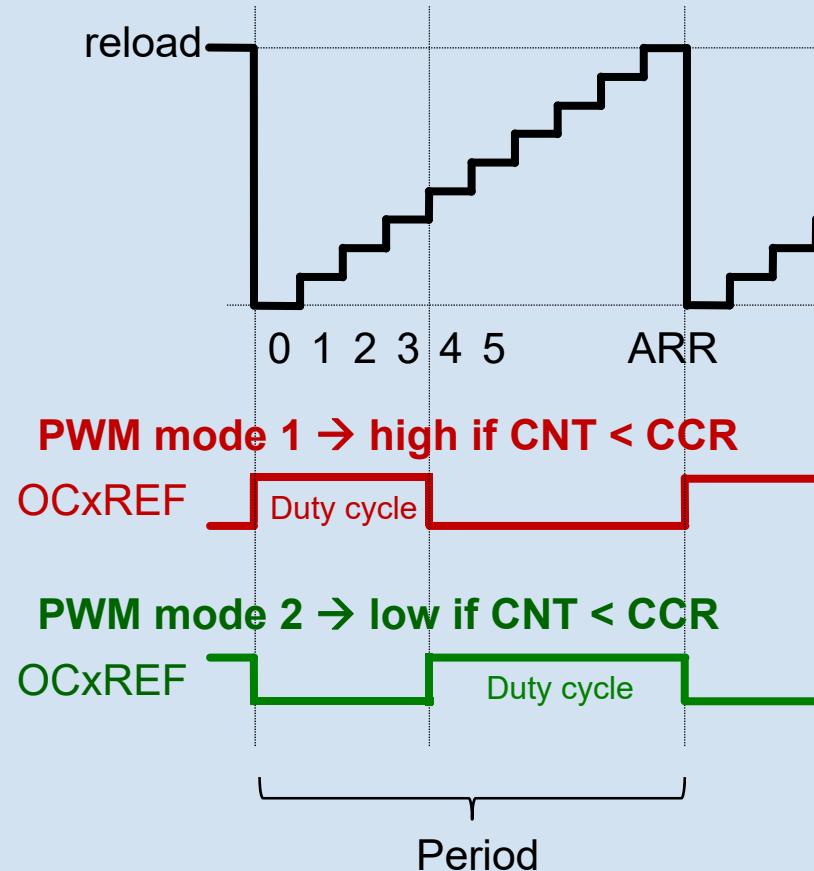
■ Edge-aligned mode

- Up-counting configuration



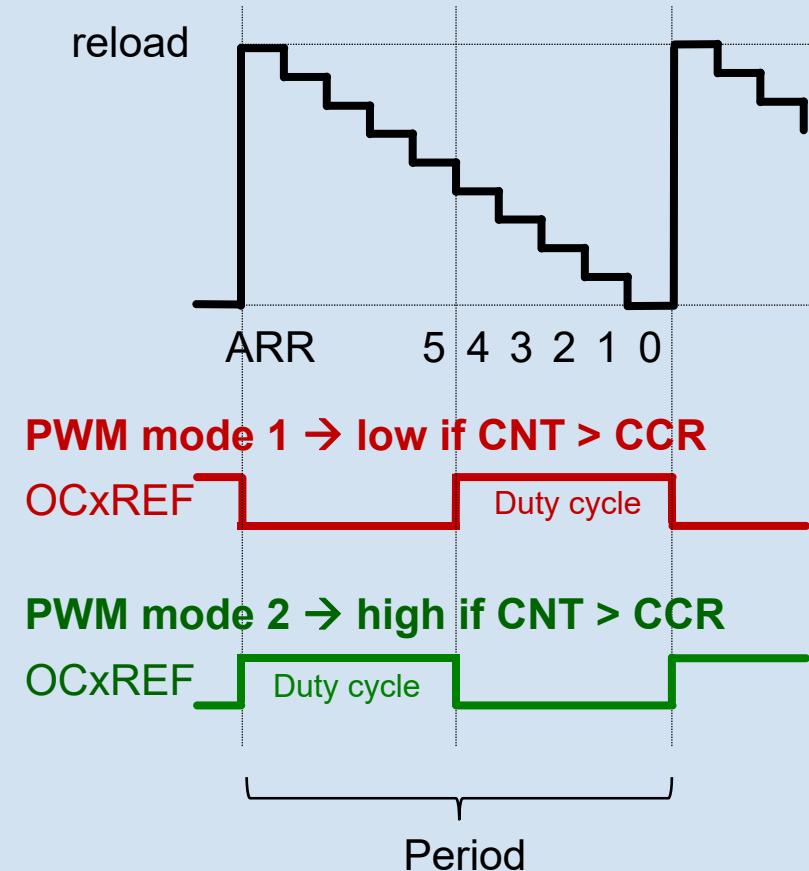
Output Compare – Generating PWM Signals

■ Up-counting



assuming Capture Compare Register (CCR) = 4

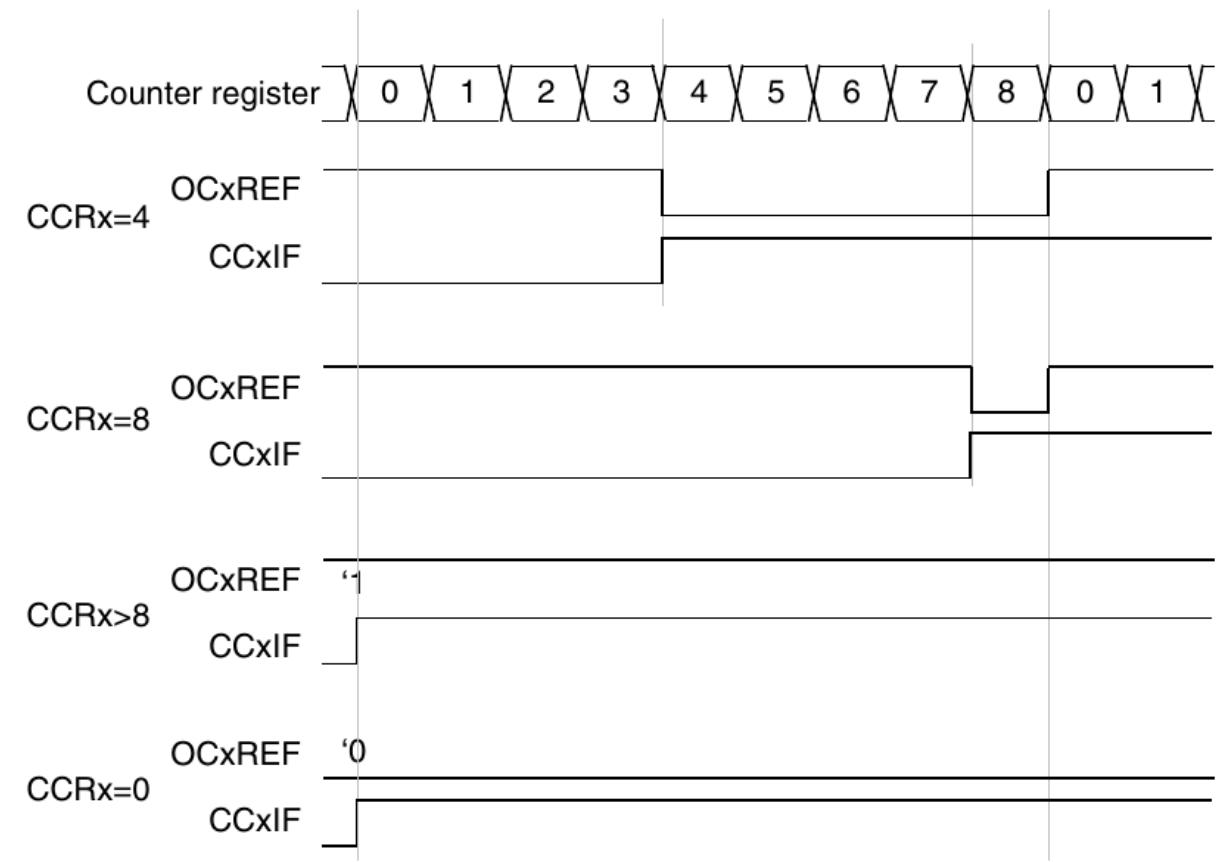
■ Down-counting



Output Compare – Generating PWM Signals

■ Edge-aligned mode

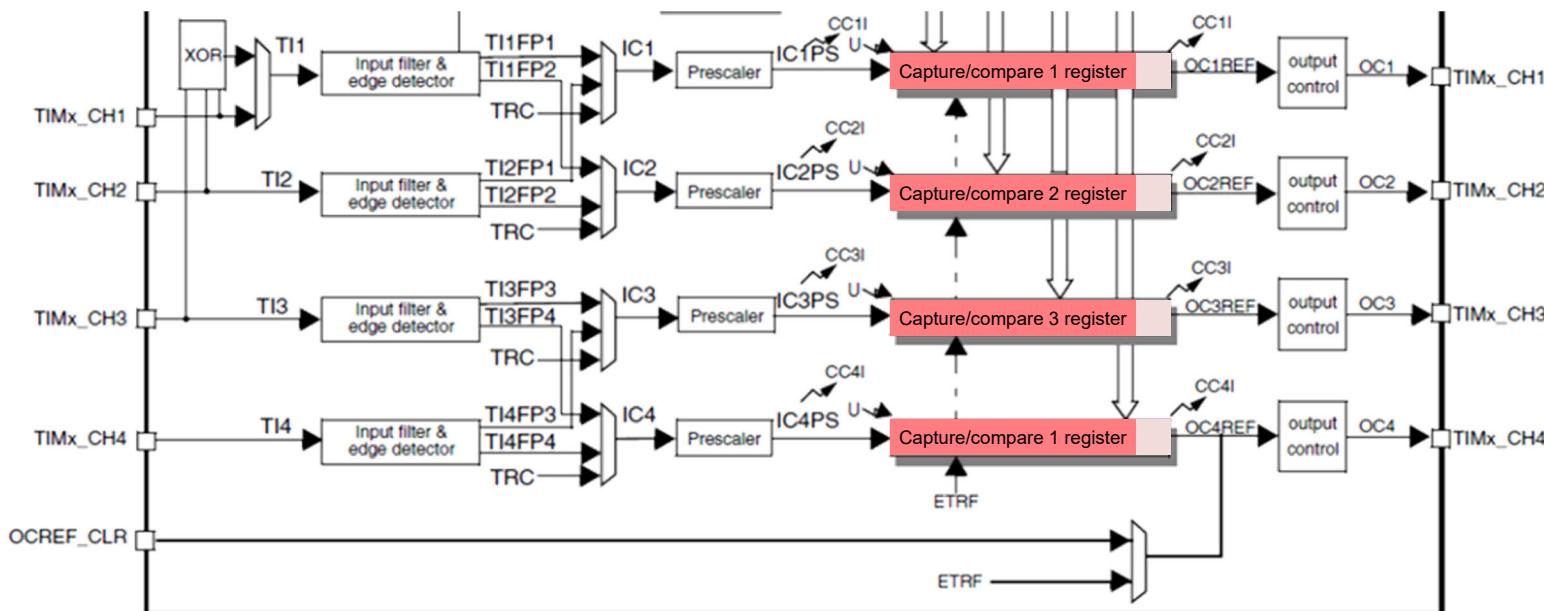
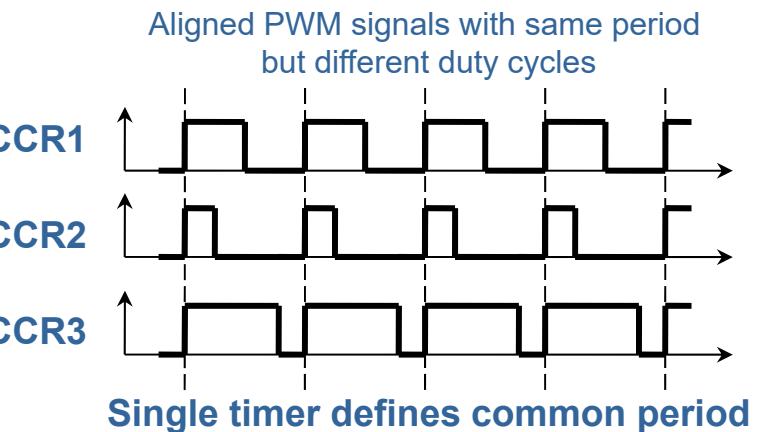
- Up-counting configuration → 4 examples for different CCR values
- $\text{TIMx_ARR} = 8$
- PWM mode 1



Input Capture / Output Compare

■ 4 independent channels for

- Input capture
- Output compare
- **PWM generation**
- One-pulse mode output



Capture / Compare Configuration

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x18	TIMx_CCMR1 Output Compare mode	Reserved										OC2CE	OC2M [2:0]		OC2PE	OC2FE	CC2S [1:0]	OC1CE	OC1M [2:0]		OC1PE	OC1FE	CC1S [1:0]										
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
	TIMx_CCMR1 Input Capture mode	Reserved										IC2F[3:0]	IC2 PSC [1:0]		CC2S [1:0]	IC1F[3:0]		IC1 PSC [1:0]	CC1S [1:0]														
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x1C	TIMx_CCMR2 Output Compare mode	Reserved										OC24CE	OC4M [2:0]		OC4PE	OC4FE	CC4S [1:0]	OC3CE	OC3M [2:0]		OC3PE	OC3FE	CC3S [1:0]										
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
	TIMx_CCMR2 Input Capture mode	Reserved										IC4F[3:0]	IC4 PSC [1:0]		CC4S [1:0]	IC3F[3:0]		IC3 PSC [1:0]	CC3S [1:0]														
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x20	TIMx_CCER	Output enable of CC →										CC4NP	CC4E		CC3NP	CC3E		CC2NP	CC2E		CC1NP	CC1E											
0x34	TIMx_CCR1	CCR1[31:16] (TIM2 and TIM5 only, reserved on the other timers)											CCR1[15:0]																				
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x38	TIMx_CCR2	CCR2[31:16] (TIM2 and TIM5 only, reserved on the other timers)											CCR2[15:0]																				
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x3C	TIMx_CCR3	CCR3[31:16] (TIM2 and TIM5 only, reserved on the other timers)											CCR3[15:0]																				
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x40	TIMx_CCR4	CCR4[31:16] (TIM2 and TIM5 only, reserved on the other timers)											CCR4[15:0]																				
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		

CCMR1 – Capture Compare Mode Register 1
CCMR2 – Capture Compare Mode Register 2

OC1M Output Compare Mode 1
 OC2M Output Compare Mode 2
 OC3M Output Compare Mode 3
 OC4M Output Compare Mode 4
 → 110 = PWM mode 1
 → 111 = PWM mode 2

CCER – Capture Compare Enable Register

CC1E Capture Compare 1 output enable
 CC2E Capture Compare 2 output enable
 CC3E Capture Compare 3 output enable
 CC4E Capture Compare 4 output enable

CCR1 – Capture Compare Register 1

CCR2 – Capture Compare Register 2

CCR3 – Capture Compare Register 3

CCR4 – Capture Compare Register 4

Other settings for advanced use
 → keep at default value

■ PWM output cookbook

- Select counter clock (internal, external, prescaler)
- Write desired data to TIMx_ARR register
→ defines common period of PWM signals
- Write desired data to TIMx_CCRx registers
→ defines duty cycles of PWM signals
- Set **CCxE** bits if **interrupts** are to be generated (in TIMx_DIER register)
- Select the output mode (registers CCMRx / CCER)
- Enable counter by setting the CEN bit in the TIMx_CR1 register

Capture / Compare Configuration

■ Use macros and structs from “reg_stm32f4xx.h”

```
/**  
 * \struct reg_tim_t  
 * \brief Representation of Timer register.  
 *  
 * Described in reference manual p.507ff.  
 */  
typedef struct {  
    volatile uint32_t CR1;           /**< Configuration register 1. */  
    volatile uint32_t CR2;           /**< Configuration register 2. */  
    volatile uint32_t SMCR;          /**< Slave mode control register. */  
    volatile uint32_t DIER;          /**< DMA/interrupt enable register. */  
    volatile uint32_t SR;            /**< Status register. */  
    volatile uint32_t EGR;           /**< Event generation register. */  
    volatile uint32_t CCMR1;          /**< Capture/compare mode register 1. */  
    volatile uint32_t CCMR2;          /**< Capture/compare mode register 2. */  
    volatile uint32_t CCER;           /**< Capture/compare enable register. */  
    volatile uint32_t CNT;            /**< Count register. */  
    volatile uint32_t PSC;            /**< Prescaler register. */  
    volatile uint32_t ARR;            /**< Auto reload register. */  
    volatile uint32_t RCR;            /**< Repetition counter register. */  
    volatile uint32_t CCR1;           /**< Capture/compare register 1. */  
    volatile uint32_t CCR2;           /**< Capture/compare register 2. */  
    volatile uint32_t CCR3;           /**< Capture/compare register 3. */  
    volatile uint32_t CCR4;           /**< Capture/compare register 4. */  
    volatile uint32_t BDTR;          /**< Break and dead-time register. */  
    volatile uint32_t DCR;            /**< DMA control register. */  
    volatile uint32_t DMAR;          /**< DMA address for full transfer. */  
    volatile uint32_t OR;             /**< Option register. */  
} reg_tim_t;
```

Example: **TIM3->CCMR2 = 0;**

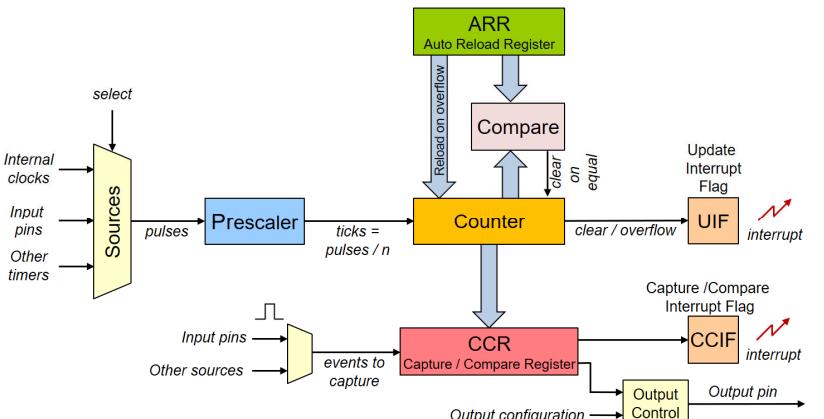
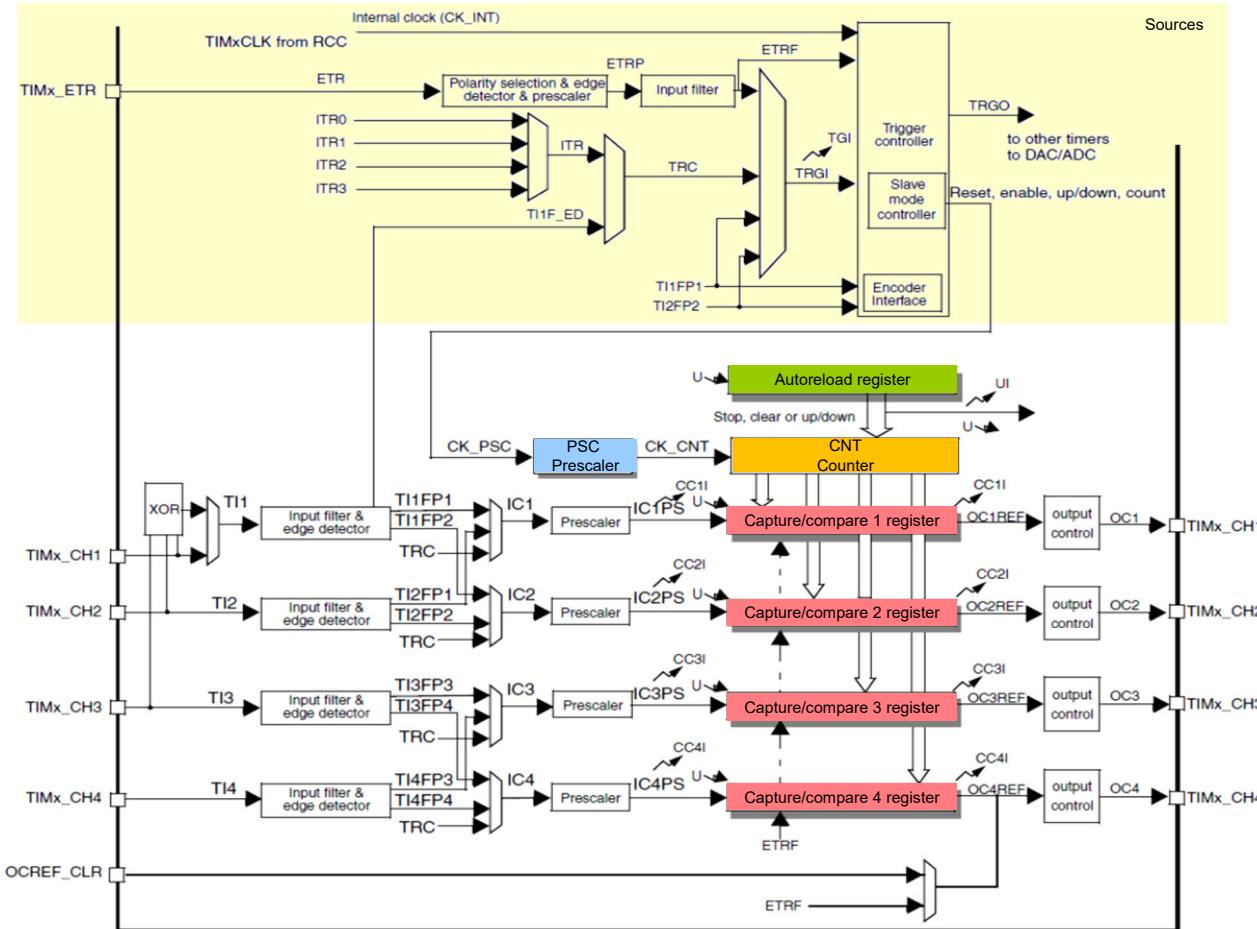
```
#define TIM2 ( (reg_tim_t *) 0x40000000 )  
#define TIM3 ( (reg_tim_t *) 0x40000400 )  
#define TIM4 ( (reg_tim_t *) 0x40000800 )  
#define TIM5 ( (reg_tim_t *) 0x40000c00 )
```

Exercise: Capture / Compare Configuration

- **Timer 2 already configured**
 - CK_INT is configured to 84 MHz
 - Timer 2 (see exercise “Timer”)
 - Up-counting, Period = 1s
 - TIM2_ARR = (10000 – 1)

- **Configure PWM with Capture/Compare 1**
 - Duty cycle 25%
 - PWM mode 1

■ Timers TIM2 - TIM5: Reference manual block diagram



■ Feature comparison

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels	Complementary output	Max interface clock (MHz)	Max timer clock (MHz) (1)
Advanced -control	TIM1, TIM8	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	Yes	90	180
General purpose	TIM2, TIM5	32-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	45	90/180
	TIM3, TIM4	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	Yes	4	No	45	90/180
	TIM9	16-bit	Up	Any integer between 1 and 65536	No	2	No	90	180
	TIM10, TIM11	16-bit	Up	Any integer between 1 and 65536	No	1	No	90	180
	TIM12	16-bit	Up	Any integer between 1 and 65536	No	2	No	45	90/180
Basic	TIM13, TIM14	16-bit	Up	Any integer between 1 and 65536	No	1	No	45	90/180
	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	Yes	0	No	45	90/180

Conclusion

■ Timer / counter functionality → TIM2 – TIM5 ST32F4xx

- Up and Down-counter with Prescaler
- Programmable count sources
- Auto Reload Register (ARR)
- Update Interrupt Flag (UIF)

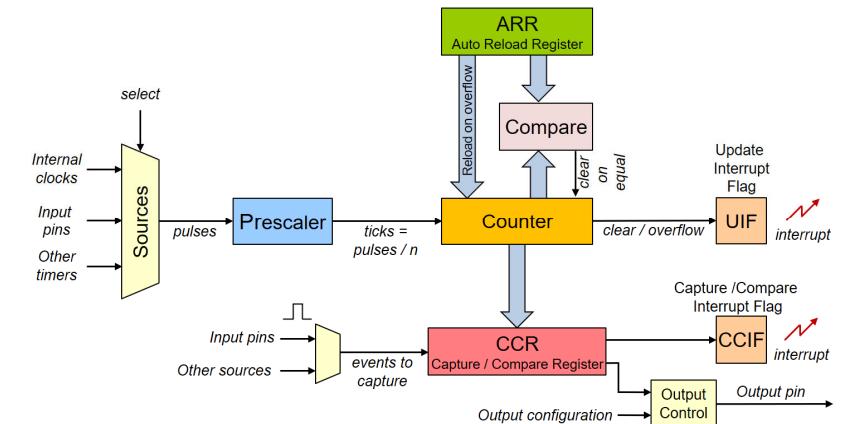
■ Capture / Compare Unit

- Measure input signals
- Generate PWM signals
- Capture / Compare Interrupt Flags (CCxIF)

■ Programming example

■ Literature

- STM32F4xx Reference Manual

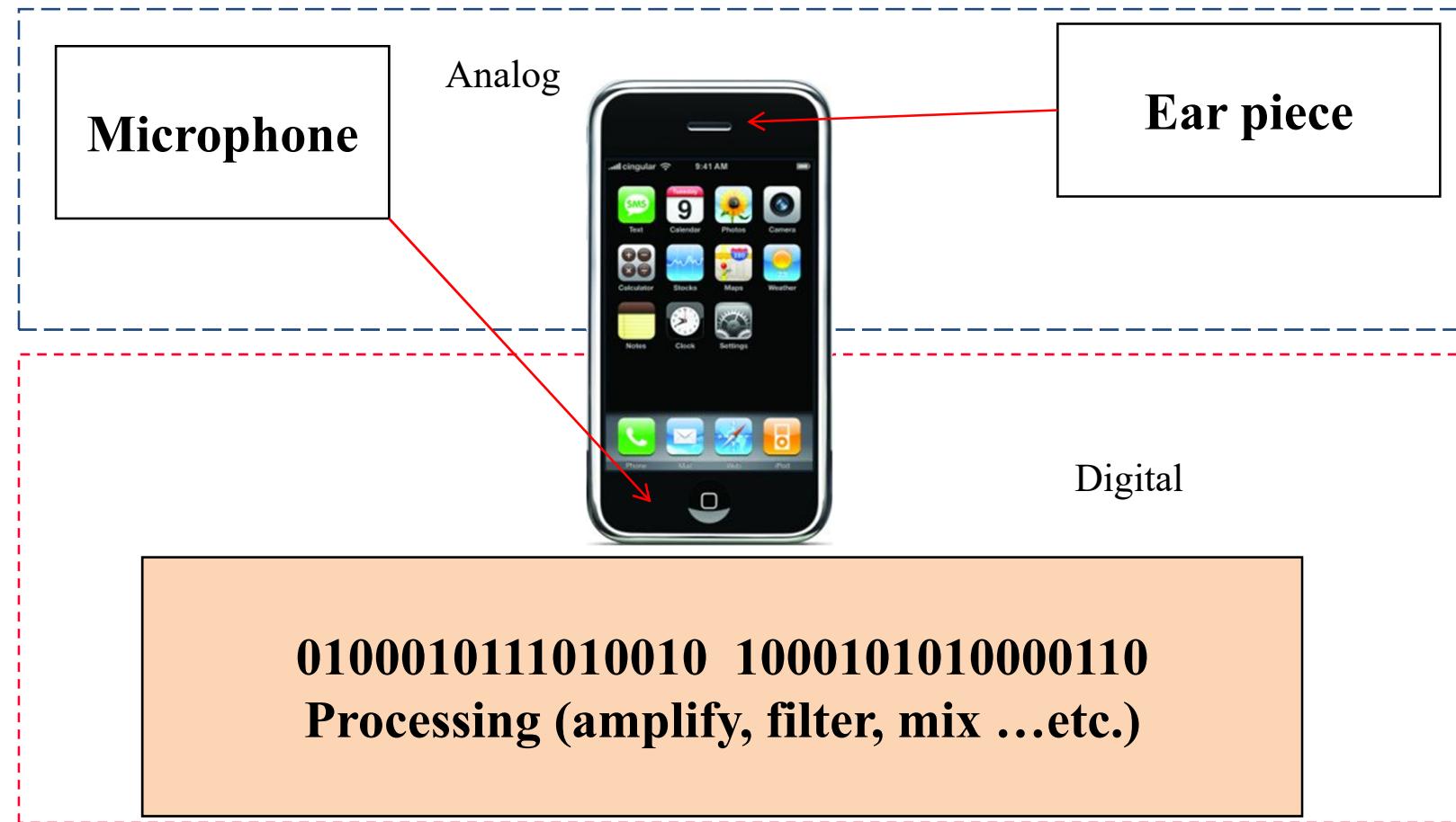


ADC / DAC

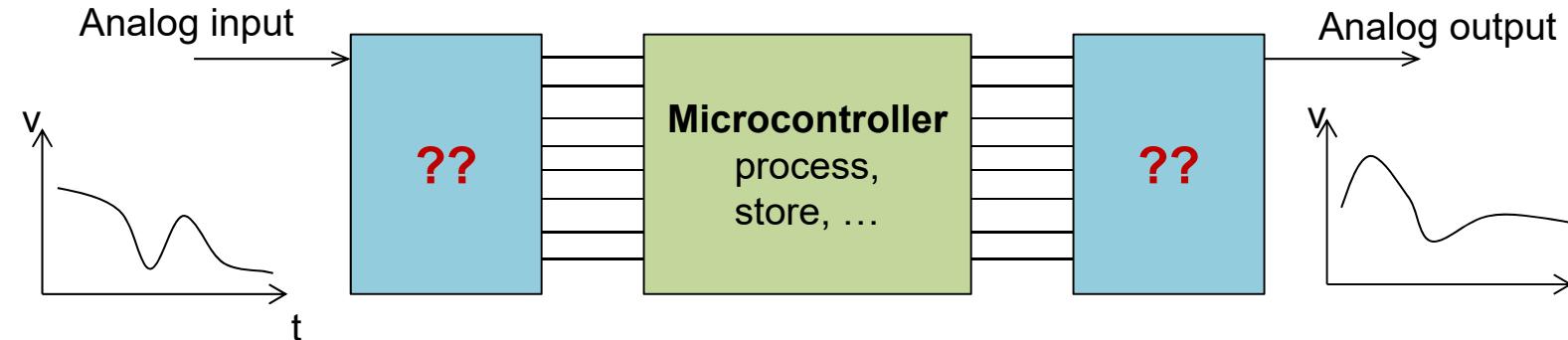
Analog-to-digital converter
Digital-to-analog converter

Computer Engineering 2

- Analog ↔ digital conversion needed in many applications



- **Modern information processing done in digital domain**
 - Easier to: process, store, make copies, ... etc.
- **But, the real world is analog (not digital)**
 - Signals are continuous and not discrete
 - Pressure sensor that continuously delivers a voltage
 - The music you hear
- **Need for devices to convert analog to digital (and vice-versa)**



- **ADC and DAC**
 - What it is
 - How it works
 - Characteristics
 - Types of error
- **ADCs on STM32F429**
 - Features and functionality
 - Programming the ADC
- **DACs on STM32F429 (optional)**
 - Features and registers
 - DAC configuration example
- **Conclusions**

At the end of this lesson you will be able

- To explain what an ADC/DAC is and what it is used for
- To describe how a (simple) Flash ADC/DAC works
- To name some application examples of ADC/DAC
- To name and explain some important characteristics/error sources
 - Sampling rate, voltage reference, offset and gain error ...
- To name basic features of ADC in the STM32F429
- To set up and use simple features of ADC in STM32F429
- To use the device documentation
 - To understand features and functionality
 - Interpret simple parameters (e.g. energy consumption, sampling rate, gain/offset error)
 - To derive simple configuration and control of ADC using the CPU
 - To find out, understand and use other features of the ADC

■ Modern microcontrollers have lots of features

- Impossible to discuss all of them in a lesson
- Important that you learn to interpret the information in the documentation
- In this lesson you will do that, under the guidance of your lecturer
- Lecture focuses on application of ADCs in microcontrollers and not on ADC design

■ Advanced modes STM32F4

- Advanced features of ADC/DAC are not described in this class
 - e.g. injected mode, dual/triple modes, ... etc.
- Interested? See application notes and datasheets (references)

■ ADC – Analog to Digital Converter

- Converts input signal (voltage) to a digital value (N-bit)
- Conversion results in one of 2^N possible numerical levels
- Raw input signal can be dynamic¹⁾ or static²⁾
 - Dynamic signal (green) sampled at specific time intervals
 - Samples transformed into series of discrete values (blue)

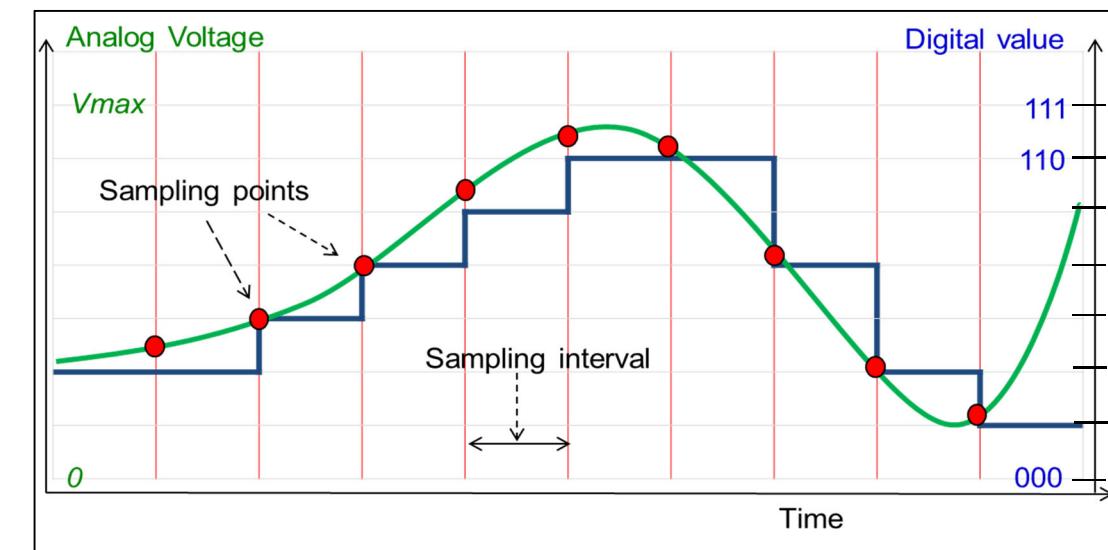
Example

3-bit ADC

- 8 possible levels (000 – 111)
- Each conversion corresponds to one out of 8 levels

1) changing over time

2) time-invariant



■ Input signals

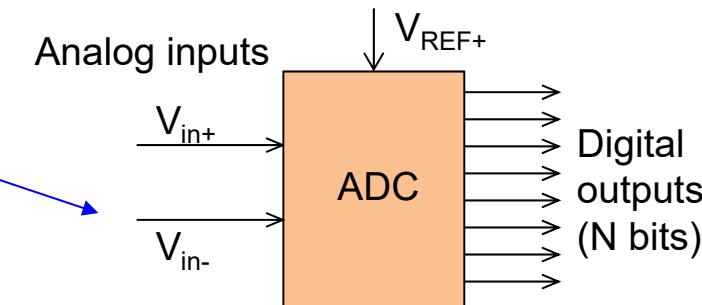
- Differential inputs
- V_{in+} signal to convert (non-inverting input)
- V_{in-} signal to convert (inverting input)

$$V_{in} = V_{in+} - V_{in-}$$

■ Single ended mode

- Only V_{in+} used
- V_{in-} is grounded

Connect to ground
for single ended



■ Reference voltage V_{REF+}

- Internal or external stable voltage
- Needed to weight input voltage

$$V_{in} = (\text{digital value}) * V_{REF+} / (2^N)$$

We will concentrate on single ended

ADC: What it is

■ Resolution

- Number of bits N
- Size of digital word

■ LSB¹

- $1 \text{ LSB} \triangleq V_{\text{REF}} / (2^N)$

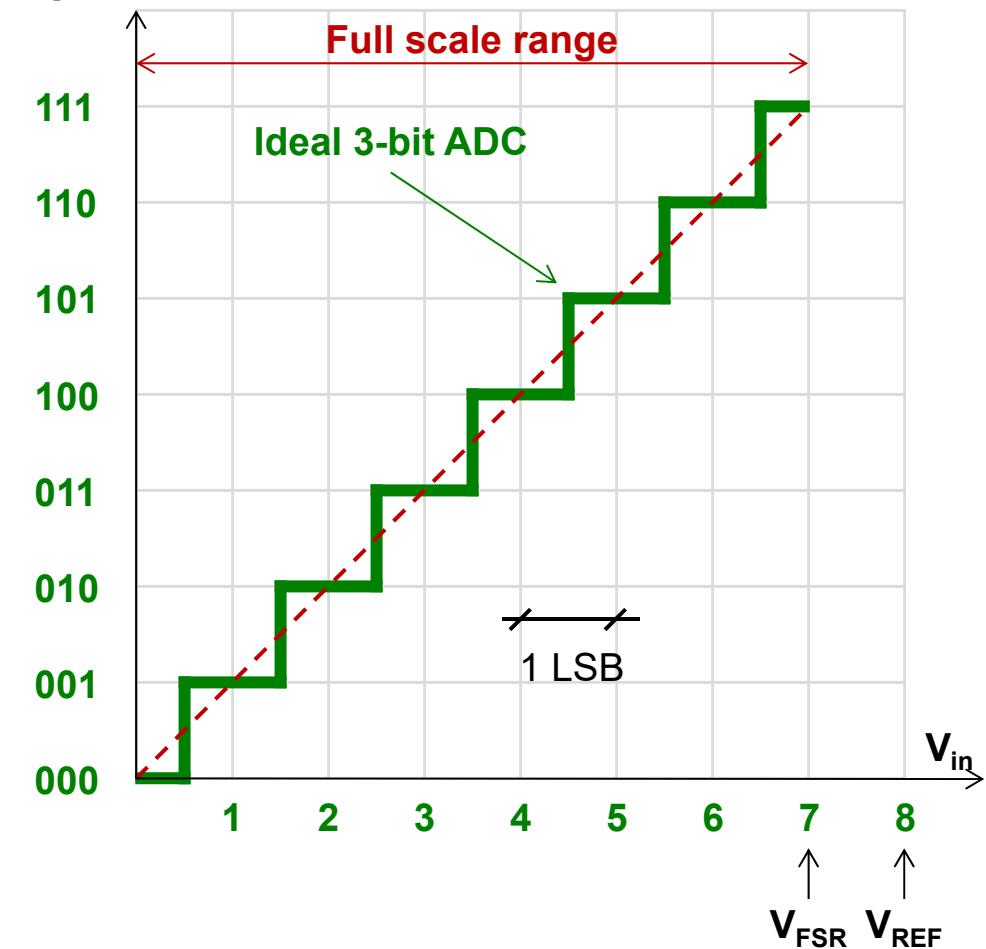
■ Full Scale Range (FSR)

- Range between analog levels of minimum and maximum digital codes
- V_{FSR} is one LSB less than V_{REF}

Example

$$V_{\text{REF}} = 8 \text{ V}, N = 3 \text{ bits} \quad \rightarrow 1 \text{ LSB} = 8 \text{ V} / 8 = 1 \text{ V}$$
$$\rightarrow \text{FSR from } 0 \text{ V to } 7 \text{ V}$$

Digital value



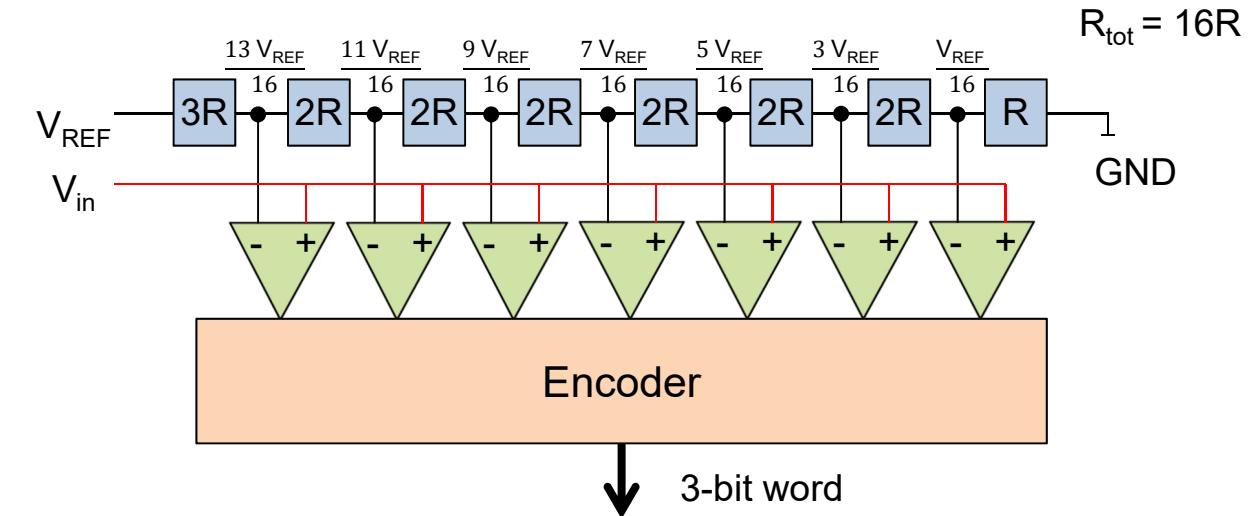
1) Least Significant Bit

■ Example Flash ADC

- Network of 2^N resistors to divide V_{REF} into 2^N levels
- $2^N - 1$ analog comparators
 - Compare input signal to divided reference voltages
- Encoder transforms digital comparator results into N-bit word

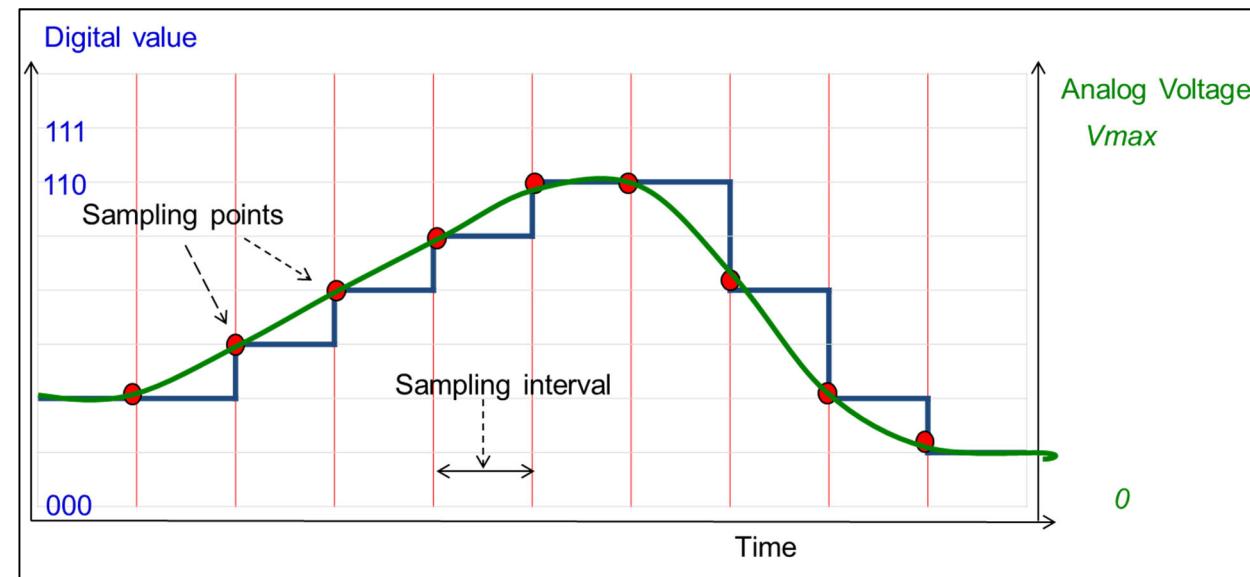
3-bit ADC
 Example: $V_{REF} = 8V$, $V_{in} = 2.3V$
 $V_{REF} / 16 = 0.5 V$
 $3 V_{REF} / 16 = 1.5 V$
 $5 V_{REF} / 16 = 2.5 V$

 Comparator stream = 0000011
 3-bit output word = 010



■ DAC – Digital to Analog Converter

- Converts N-bit digital input to analog voltage level
- E.g. music from your MP3 player is read and converted back to sound
 - A series of different values in the digital domain leads to a series of steps in the analog domain. The result is a dynamic output signal
 - “Play-back” time depends on time between conversions (sampling interval)



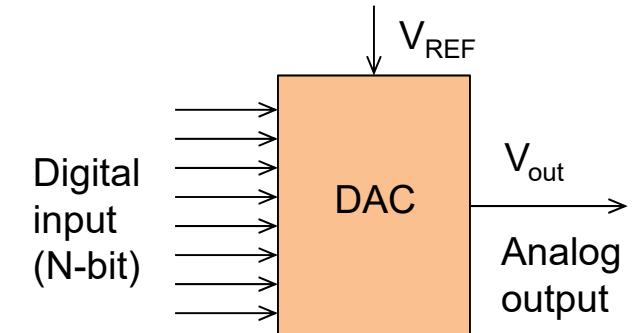
■ Reference voltage V_{REF}

- Accurate reference voltage (from internal or external source)
- Needed to relate digital value to a voltage

■ Output signal V_{out}

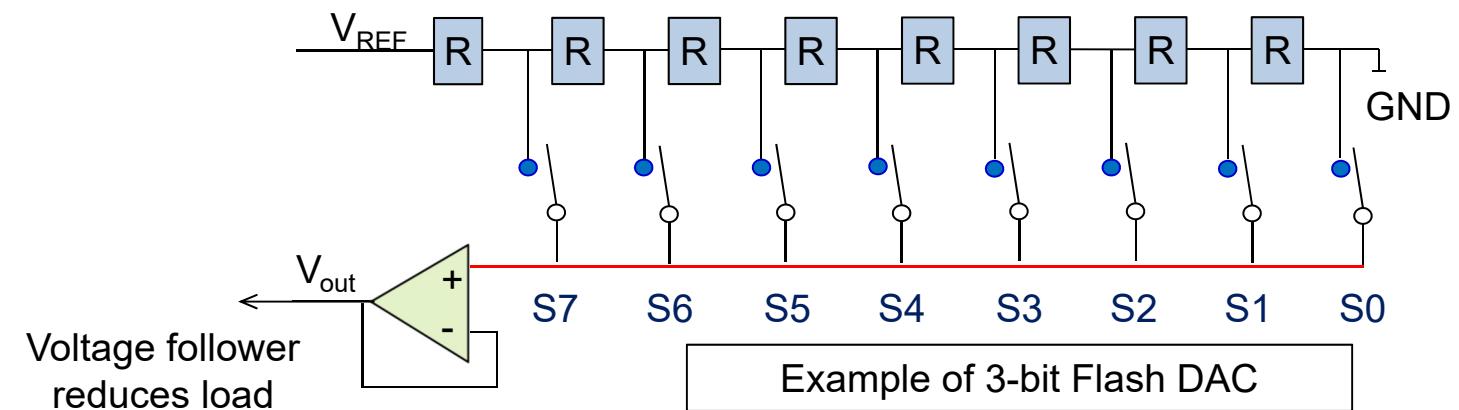
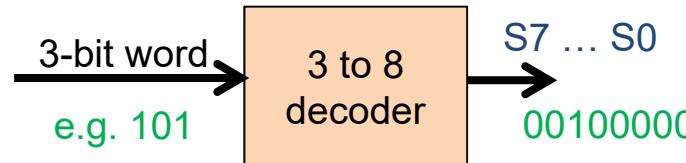
- Analog output
 - Unipolar (only positive)
 - Bipolar (positive or negative)
- Conversion yields approximation of digital signal

$$V_{out} = (\text{digital value}) * V_{REF} / (2^N)$$



■ Example Flash DAC

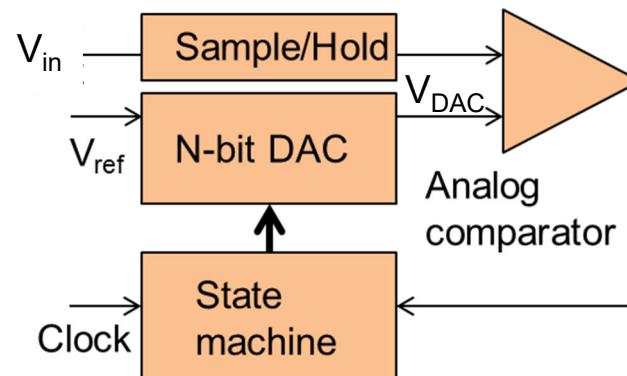
- Network of resistors (of same value) creates 2^N voltage levels
- N-bit digital input decoded into 2^N values ($S_0 \dots S_x$)
 - Select single voltage level as DAC output



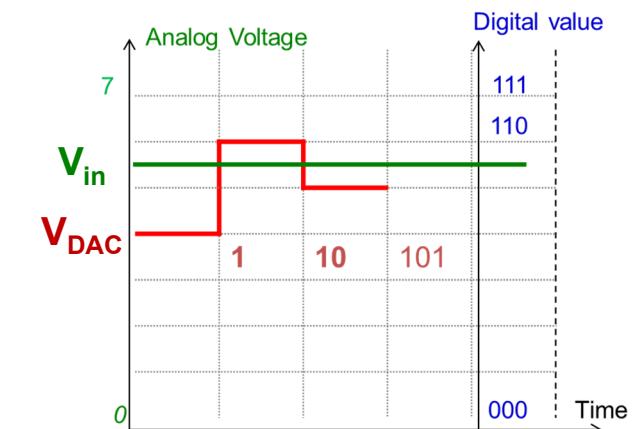
■ Successive Approximation Register (SAR) ADC

- Approach V_{in} with successive division by 2
 - Binary search
- Start with half the digital value
 - MSB = 1, all other bits at 0
- DAC generates analog value V_{DAC} that is compared to V_{in}
 - If $V_{DAC} < V_{in}$ → keep MSB at 1, otherwise set MSB to 0
- Continue with other N bits in same way (N steps)

SAR ADC as an alternative to the previously introduced Flash ADC



1. Higher than 100 → 1
 2. Lower than 110 → 10
 3. Higher than 101 → 101
- 3-bit result = 101**



■ Flash ADC

- Fast conversion
- Requires many elements
 - e.g. 255 comparators for 8-bit resolution
 - Power hungry
 - Consumes large chip area

■ SAR ADC

- Used on most microcontrollers
- Good trade-off between speed, power and cost
 - Up to 5 Msps
 - Resolution from 8 to 16 bits

■ Sampling rate

- Input signal sampled at discrete points in time → discontinuities
- Should be at least twice the highest frequency component of input signal
 - Nyquist–Shannon sampling theorem

■ Conversion time

- Time between start of sampling and digital output available
- Programming a higher resolution may increase conversion time

■ Monotonicity

- Increase of V_{in} results in increase or no change of digital output and vice-versa

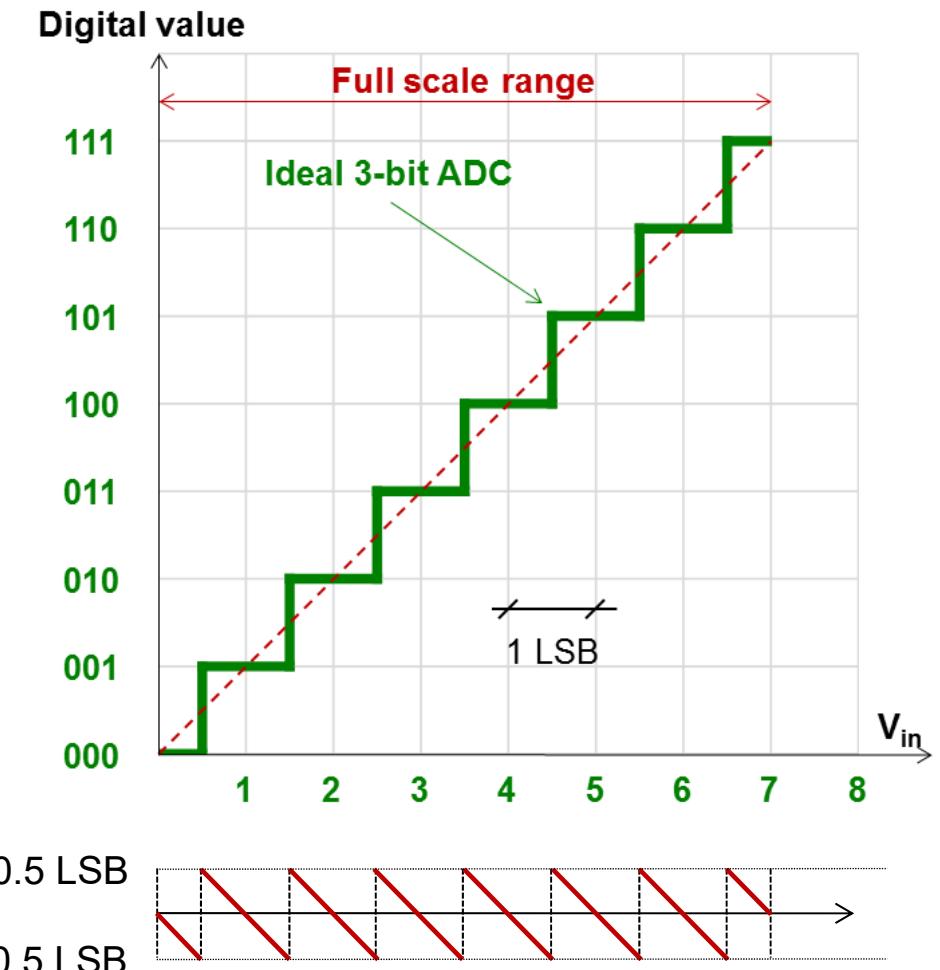
ADC: Types of Error

■ Quantization error

- Analog input is continuous
 - Infinite number of states
- Digital output is discrete
 - Finite number of states
- Introduces an error between -0.5 LSB and +0.5 LSB

Quantization error can be reduced by reducing LSB,
e.g. either by increasing number of bits (resolution)
or by reducing V_{REF}

Reducing V_{REF} also reduces full scale range

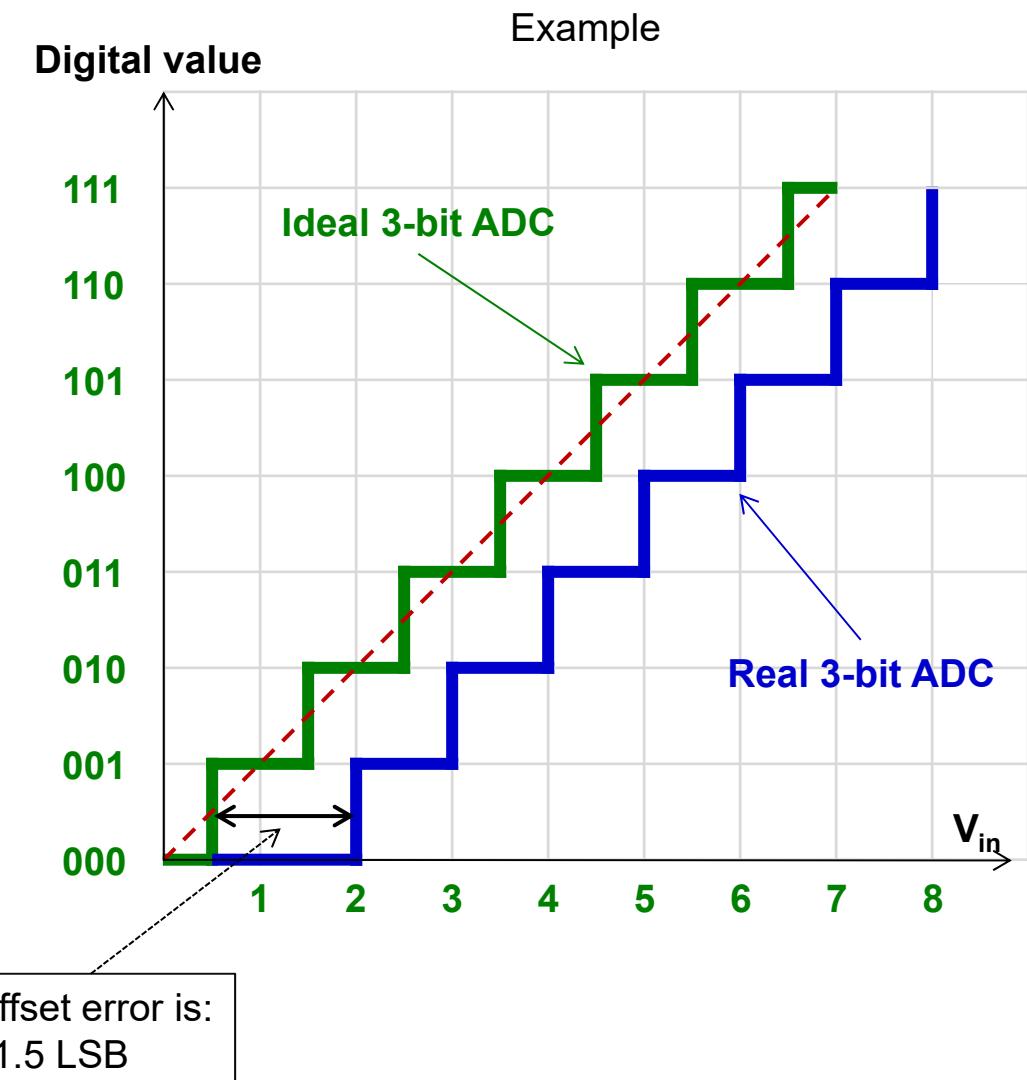


ADC: Types of Error

■ Offset error

- Also called zero-scale error
- Deviation of real N-bit ADC from ideal N-bit ADC at input point zero
- For an ideal N-bit ADC, the first transition occurs at 0.5 LSB above zero
- Can be corrected using the microcontroller

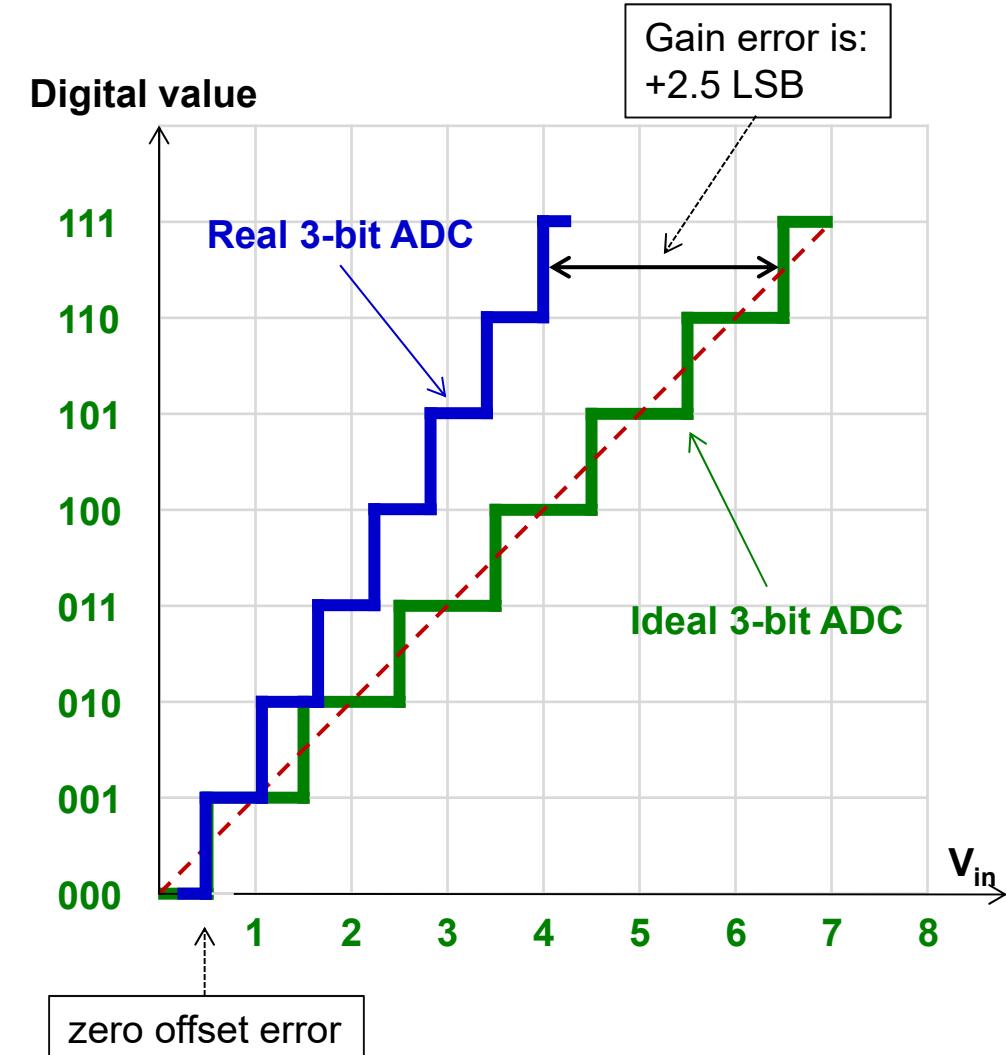
Measuring the offset error:
Zero-scale voltage is applied to analog input and is increased until first transition occurs



■ Gain error

- Indicates how well the slope of an actual transfer function matches the slope of the ideal transfer function
- Expressed in LSB or as a percent of full-scale range (%FSR)
- Calibration with hardware or software possible

$$\text{full-scale error} = \text{offset error} + \text{gain error}$$



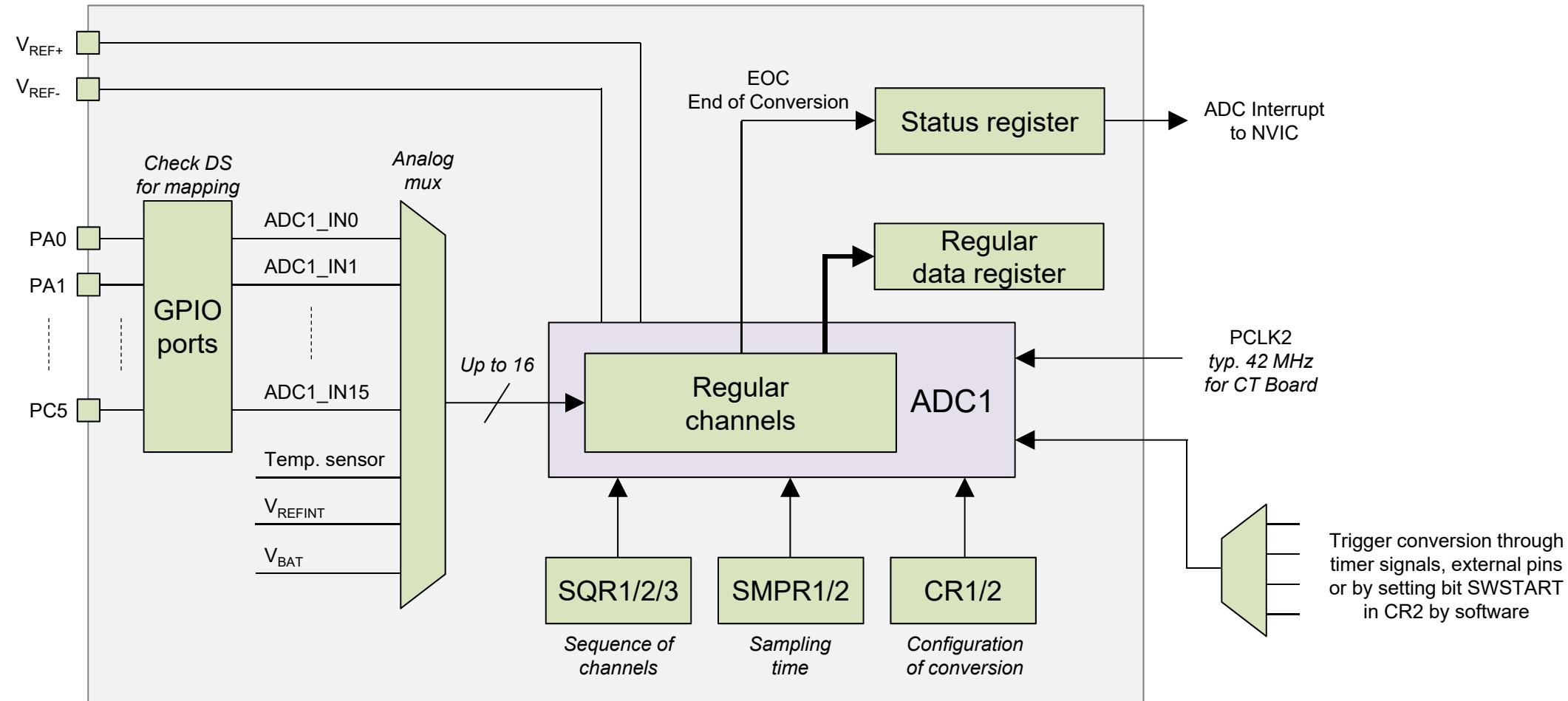
Features and Functionality

ADCs on STM32F429

ADC Exercise: Reading the Datasheet

- **Q1:** How many ADCs are there in the STM32F429?
- **Q2:** How many sources for each ADC?
 - What is meant by internal/external source?
 - How many external sources?
 - How many internal sources? What are they?
 - What is temperature monitoring?
- **Q3:** How many bits can the result of a conversion have?
- **Q4:** How is the result of the conversion stored?
 - Can it be overwritten? What are the consequences?
 - What is meant by left aligned/right aligned?
- **Q5:** How can the result of conversion be transferred to memory?
- **Q6:** To which ADC can pins PA5/PA6 be connected?
- **Q7:** What is meant by single/continuous conversion vs single channel/scan mode?
- **Q8:** How does the CPU know that the conversion is done?

Simplified ADC Diagram

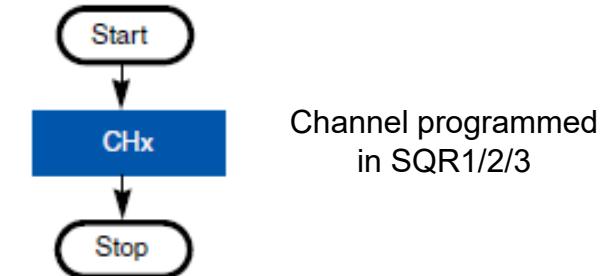


■ **Single channel / multi-channel** vs **Single / continuous conversion**

	Single channel	Multi-channel (scan mode)
Single conversion	Convert 1 channel, then stop. This is the simplest mode.	Convert all channels in group, one after the other, then stop. The group of channels is in a sequence that can be programmed.
Continuous conversion	Continuously convert 1 channel until stop order is given. Minimal CPU intervention.	Continuously convert a group of several channels until stop order is given. The group of channels is in a sequence that can be programmed.

■ Single channel single conversion

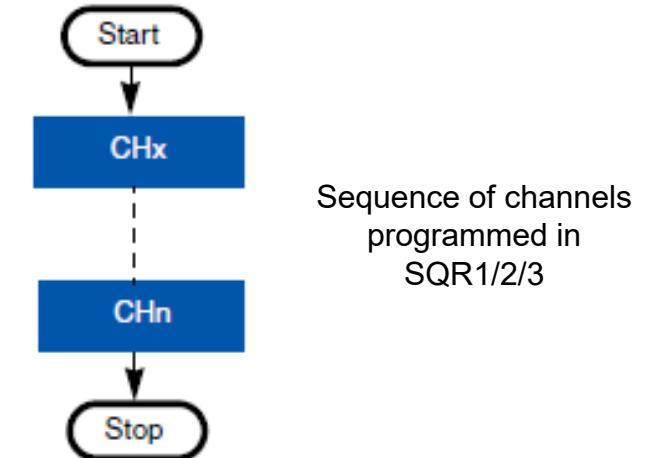
- Possible use
 - Sensor tied to ADC input
 - On a push button you read and display it



Channel programmed
in SQR1/2/3

■ Multi-channel single conversion

- ADC sequencer used to set up order of conversions.
- Possible use
 - Group of sensors need to be monitored
 - For each sensor there is an ADC input
 - Sensors are grouped and scanned one after the other every time a reading needs to be done



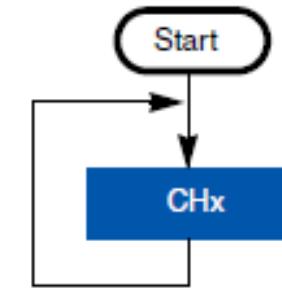
Sequence of channels
programmed in
SQR1/2/3

■ Usefulness

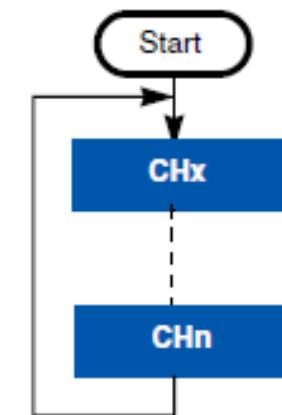
- Minimal CPU intervention for setups → ADC does not require CPU intervention to restart.
- Beware overwriting previous results

■ Single channel continuous conversion

- Possible use: Continuously monitor a single sensor
 - Result can be compared in background to specific value and action taken only when that value reaches limit



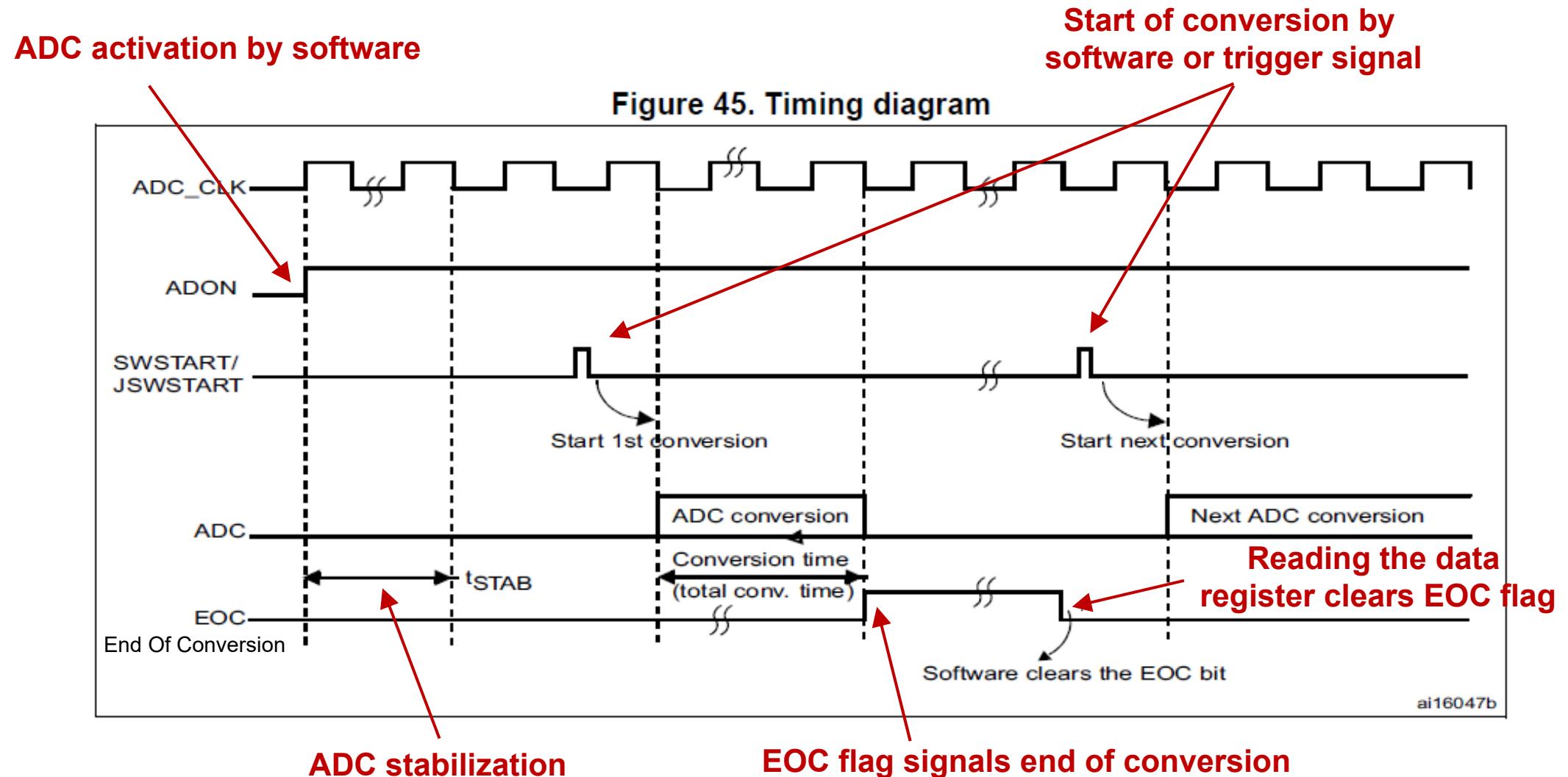
Channel programmed
in SQR1/2/3



Sequence of channels
programmed in
SQR1/2/3

■ Multi channel continuous conversion

- Possible use: Continuously monitor several sensors



■ Total conversion time

- Depends on time for sampling and conversion
- T_{sample} individually programmable for each channel
 - Registers ADC_SMPR1 and ADC_SMPR2
 - Between 3 and 480 cycles
- T_{conv} depends on resolution
 - 12 bits 12 ADCCLK cycles
 - 10 bits 10 ADCCLK cycles
 - 8 bits 8 ADCCLK cycles
 - 6 bits 6 ADCCLK cycles

$$T_{\text{total}} = T_{\text{sample}} + T_{\text{conv}}$$

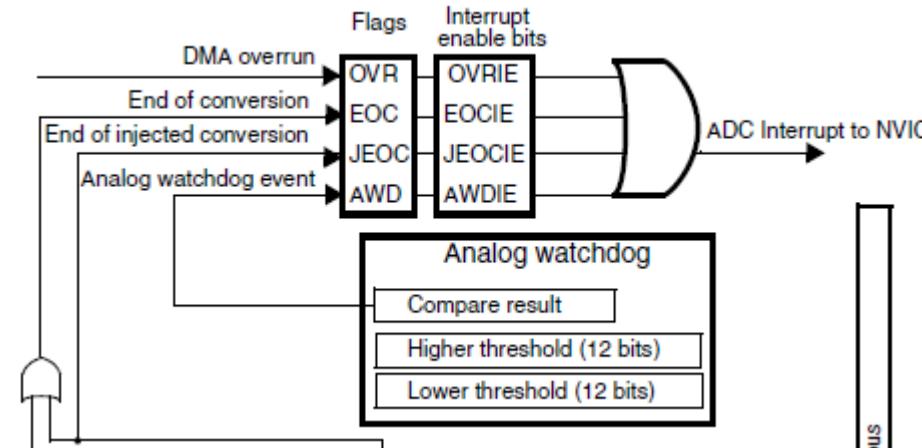
Example

given: APB2 clock = 48 MHz
Prescaler 2 → ADCCLK = 24 MHz
3 cycles sampling time
12 bit resolution

$$T_{\text{total}} = (3 + 12) * 1 / 24 \text{ MHz} = 0.625 \text{ us}$$

$$\text{sampling rate} < 1 / T_{\text{total}} = 1.6 \text{ Msps}$$

- Allows guarded monitoring of one or more channels with very little CPU overhead
- Converted value is compared to programmable guards (min and max)
- Flag (interrupt) if monitored signal outside limits
- Example
 - potentiometer can be guarded so that an activity is started only if knob is in certain area



STM32F429 ADC: Characteristics

■ Datasheet gives information about important parameters

- Current consumption: Useful for the system power budget
 - Goes up by 1.6 mA for each ADC that is active
- Max. sampling rate
 - Useful for defining the max. speed of the input signal
 - Sampling at up to 2.4 Msps possible
- Offset error (max = +-2.5 LSB)
- Gain error (typ = +-1.5 LSB)

Table 78. ADC static accuracy at $f_{ADC} = 30 \text{ MHz}^{(1)}$

Symbol	Parameter	Test conditions	Typ	Max ⁽²⁾	Unit
ET	Total unadjusted error		±2	±5	
EO	Offset error	$f_{ADC} = 30 \text{ MHz}$, $R_{AIN} < 10 \text{ k}\Omega$,	±1.5	±2.5	LSB
EG	Gain error	$V_{DDA} = 2.4 \text{ to } 3.6 \text{ V}$, $V_{REF} = 1.8 \text{ to } 3.6 \text{ V}$	±1.5	±3	
ED	Differential linearity error	$V_{DDA} - V_{REF} < 1.2 \text{ V}$	±1	±2	
EL	Integral linearity error		±1.5	±3	

Operating power supply range	ADC operation
$V_{DD} = 1.8 \text{ to } 2.1 \text{ V}^{(3)}$	Conversion time up to 1.2 Msps
$V_{DD} = 2.1 \text{ to } 2.4 \text{ V}$	Conversion time up to 1.2 Msps
$V_{DD} = 2.4 \text{ to } 2.7 \text{ V}$	Conversion time up to 2.4 Msps
$V_{DD} = 2.7 \text{ to } 3.6 \text{ V}^{(5)}$	Conversion time up to 2.4 Msps

STM32F429

Programming the ADC

■ ADC registers

- Some registers are common for all ADCs (common registers)
- Each ADC also has specific registers
 - Structure of specific registers similar for all 3 ADCs

ADC region		Offset		Address of register
0x4001 2000 - 0x4001 23FF	ADC1	0x000 - 0x04C	Specific registers	0x4001 2000 + 0x000 + register offset
			Reserved	
	ADC2	0x100 - 0x14C	Specific registers	0x4001 2000 + 0x100 + register offset
			Reserved	
	ADC3	0x200 - 0x24C	Specific registers	0x4001 2000 + 0x200 + register offset
			Reserved	
	Common	0x300 - 0x308	Common registers	0x4001 2000 + 0x300 + register offset

Programming the ADC

■ Common registers

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0x00	ADC_CSR	Reserved										OVR	0	STRT	0	JSTRT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Reset value	ADC3										0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x04	ADC_CCR	Reserved										TSVREFE	0	VBATE	0	Reserved	0	ADCPRE[1:0]	0	DMA[1:0]	0	DDS	0	0	0	0	0	0	0	0	0	0	0	0	0	
	Reset value	0 0										0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x04	ADC_CCR	Reserved										TSVREFE	0	VBATE	0	Reserved	0	ADCPRE[1:0]	0	DMA[1:0]	0	DDS	0	0	0	0	0	0	0	0	0	0	0	0	0	
	Reset value	0 0										0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x08	ADC_CDR	Regular DATA2[15:0]										0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	Reset value	0 0 0 0 0 0 0 0 0 0										0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		

ADC CCR ADC Common Control Register

TSVREFE Enable/disable temp sensor and V_{REFINT}

VBATE Enable/disable V_{BAT}

ADCPRE[1:0] Prescaler for ADCCLK: 00/01/10/11 → APB2 clock divided by 2/4/6/8
 APB2 clock = 42 MHz on CT_Board

All other positions kept at 0.
 Features not relevant for this class

Specific Registers for each ADC

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	ADC_SR	Reserved																															
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x04	ADC_CR1	Reserved				OVRIE	RES[1:0]	AWDEN	JAWDEN	Reserved				DISC NUM [2:0]	JDISCEN	DISCEN	JAUTO	AWD SGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]					JSTART					
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x08	ADC_CR2	Re se rv ed	SWSTART	EXTEN[1:0]	EXTSEL [3:0]			Re se rv ed	JSWSTART	JEXTEN[1:0]	JEXTSEL [3:0]			Reserved	ALIGN	EOCS	DDS	DMA	Reserved					CONT	ADON	0							
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x0C	ADC_SMPR1	Sample time bits SMPx_x																															
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x10	ADC_SMPR2	Sample time bits SMPx_x																															
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x2C	ADC_SQR1	Reserved				L[3:0]			Regular channel sequence SQx_x bits																								
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
0x30	ADC_SQR2	Reserved	Regular channel sequence SQx_x bits																														
	Reset value		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
0x34	ADC_SQR3	Reserved	Regular channel sequence SQx_x bits																														
	Reset value		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
0x4C	ADC_DR	Reserved												Regular DATA[15:0]																			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		

status register

control registers

sample time registers

sequence registers

data register

← other registers

← other registers

Specific Registers for each ADC

■ Register Bits

In our cases keep all other bits to '0'

ADC_SR ADC Status Register

OVR	Overrun. 1 → Result data overwritten
STRT	Conversion started (regular channel)
EOC	End Of Conversion. Cleared by reading result of conversion

ADC_CR1 ADC Control Register 1

OVRIE	OVR Interrupt Enable
EOCIE	EOC Interrupt Enable
RES[1:0]	Conversion resolution: 00/01/10/11 → 12/10/8/6-bit
SCAN	Enable scan mode

ADC_CR2 ADC Control Register 2

SWSTART	Start conversion of regular channel by software. Cleared by HW
EXTEN[1:0]	Ext. trigger for regular channels 00/01/10/11 → disabled/pos edge/neg edge/pos & neg edge
EXTSEL	External event select to trigger conversion of a regular group
ALIGN	Data alignment : '0' → right aligned, '1' → left aligned
EOCS	EOC selection. '0' / '1' → EOC shows end of each sequence of conversions/end of each conversion
DMA	Enable DMA
CONT	Continuous mode: 0 → Single conversion mode 1 → Continuous conversion mode
ADON	ADC On

Specific Registers for each ADC

■ Sample Time

ADC SMPR1 ADC Sample Time Register1															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				SMP18[2:0]			SMP17[2:0]			SMP16[2:0]			SMP15[2:1]		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
ADC SMPR2 ADC Sample Time Register2															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]		
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
SMPx[2:0]		Sampling time for channel x													
'000'		3 cycles			'001'		15 cycles			'010'		28 cycles			
'011'		56 cycles			'100'		84 cycles			'101'		112 cycles			
'110'		144 cycles			'111'		480 cycles								

Specific Registers for each ADC

■ Sequence of Channels

ADC SQR1 ADC Sequence Register 1

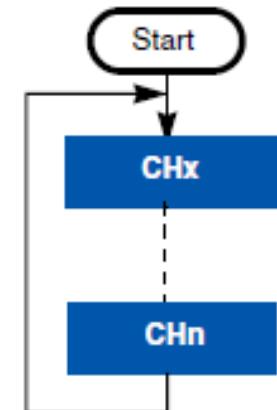
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved										L[3:0]		SQ16[4:1]			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

ADC SQR2 ADC Sequence Register 2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SQ12[4:0]					SQ11[4:0]					SQ10[4:1]			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ10_0	SQ9[4:0]					SQ8[4:0]					SQ7[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

ADC SQR3 ADC Sequence Register 3

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SQ6[4:0]					SQ5[4:0]					SQ4[4:1]			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ4_0	SQ3[4:0]					SQ2[4:0]					SQ1[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

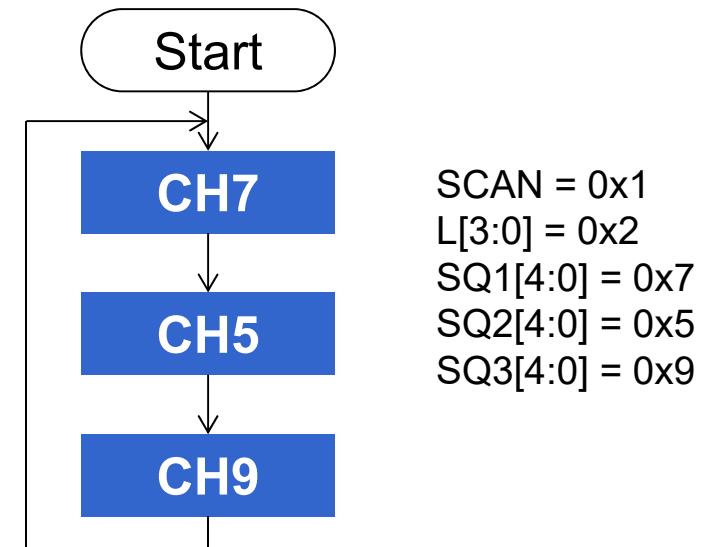


Specific Registers for each ADC

■ Sequence of Channels

<u>ADC_SQR1/2/3 ADC Sequence Register</u>	
L[3:0]	Sequence length: L+1 = Number of conversions in regular sequence '0000' → 1 conversion upto '1111' → 16 conversions
SQx[4:0]	Channel number of xth conversion in sequence, $1 \leq x \leq 16$ e.g. SQ3[4:0] = 0x7 means the 3 rd conversion takes place on channel 7

Example



Programming the ADC → Example for PF8

```
/* Setup GPIO and ADC3 */
hal_rcc_set_peripheral(PER_GPIOF, ENABLE);
hal_rcc_set_peripheral(PER_ADC3, ENABLE);

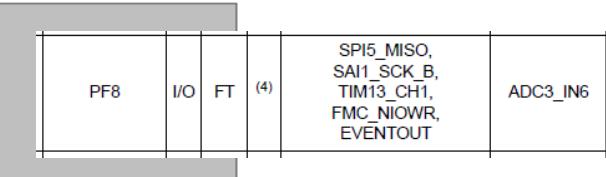
GPIOF->MODER |= (0x3 << 16);      // analog pin conf on PF8
ADCCOM->CCR = (0x3 << 16);        // ADC prescaler 8

ADC3->CR1 = 0x0;                   // 12 bit resolution, no scan
ADC3->CR2 = 0x1;                   // single conv., enable ADC, right align

ADC3->SMPR1 = 0x0;
ADC3->SMPR2 = (0x6 << 12);       // ch4: 144 cycles sampling time

ADC3->SQR1 = 0x0;                  // L = '0000' -> sequence length: 1
ADC3->SQR2 = 0x0;
ADC3->SQR3 = 0x6;                  // ch6 is first in sequence

while (1) {
    ADC3->CR2 |= (0x1 << 30);    // start conversion
    while(!(ADC3->SR & 0x2)) {     // wait while conversion not finished
    }
    CT_SEG7->BIN.HWORD = ADC3->DR; // show on 7-segment display
}
```



What is the max. achievable sampling rate with these settings and APB2 clock = 42 MHz?

Functional Summary of ADC

3) Signals used to inform the CPU of the state of the conversion process. Status register can be checked or interrupts generated.

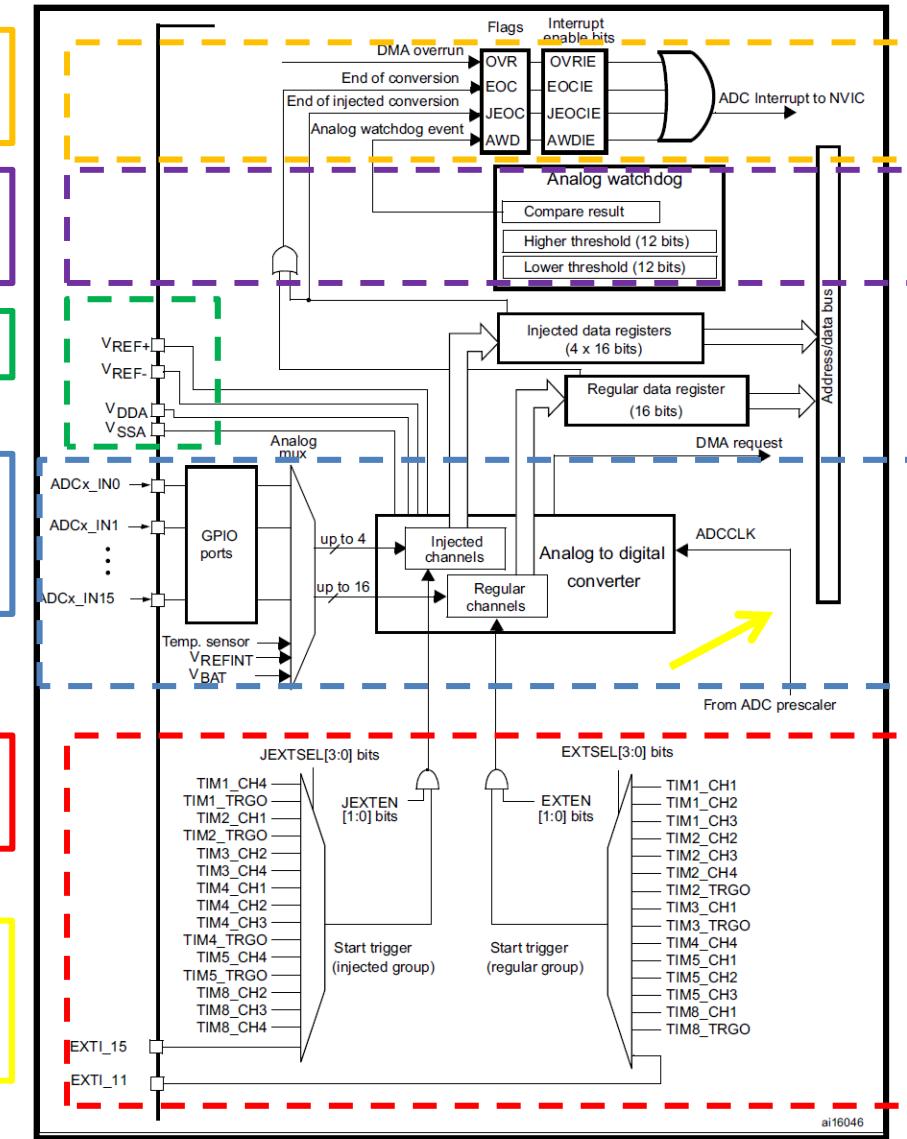
4) Analog watchdog compares conversion results with programmed min/max limits.

5) Analog pins to set the references

1) 3 ADCs (capable of 12/10/8/6 bit resolution)
- 16 possible external sources (pins) each
- 3 possible internal sources

2) Conversion triggered by CPU, by internal sources or by external sources

6) ADCCLK is used as clock for conversions.
Generated by dedicated prescaler that allows APB2 clock to be scaled down by 2/4/6/8



Features and Functionality

DACs on STM32F429 (Optional)

Features of STM32F429 DAC (optional)

2 voltage output DACs. Can be combined with DMA

- Independent output each
- Both support 12-bit and 8-Bit modes. Left or right data alignment in 12-bit mode

Conversion can be triggered by:

- SW
- internal timers
- external event

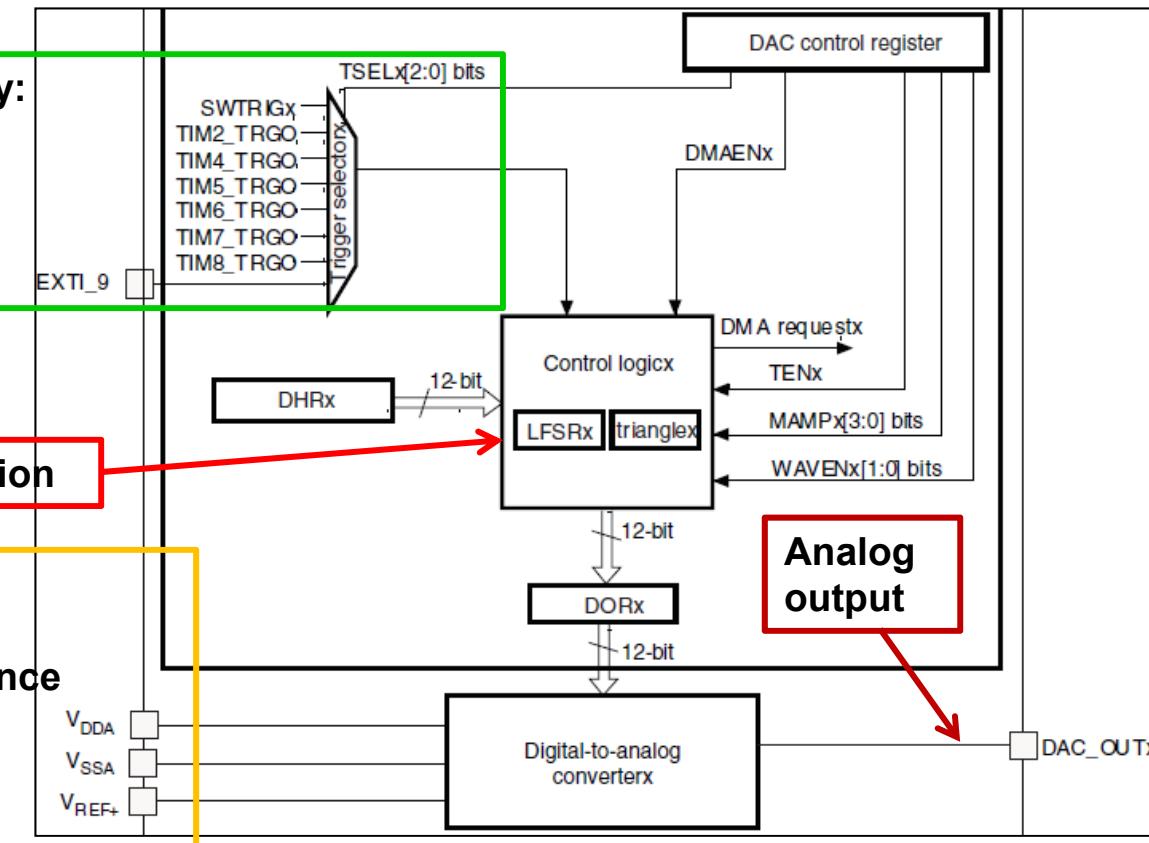
Source is SW selectable

Noise/Triangular-wave generation

$$1.8V \leq V_{DDA} \leq 3.6V$$

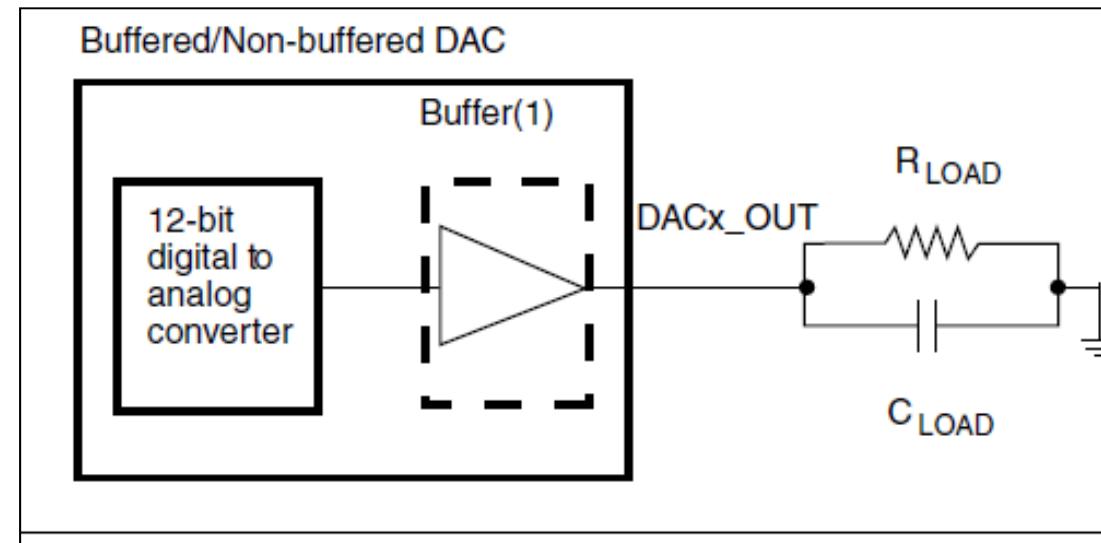
$$1.8V \leq V_{REF+} \leq V_{DDA}$$

- V_{REF+} : External voltage reference
- V_{DDA} : Analog power supply
- V_{SSA} : Analog GND



■ Overview

- The 2 DACs can be grouped for synchronous update
 - Independent or simultaneous modes
- Integrated output buffers can be used to reduce the output impedance, and to drive external loads directly
 - No need for external operational amplifier



■ Overview of some DAC electrical parameters

- The use of the output buffer reduces external components and helps improve performance if the load is important

Table 87. DAC characteristics

Symbol	Parameter	Min	Typ	Max	Unit	Comments
$R_{LOAD}^{(2)}$	Resistive load with buffer ON	5	-	-	kΩ	
$R_O^{(2)}$	Impedance output with buffer OFF	-	-	15	kΩ	When the buffer is OFF, the Minimum resistive load between DAC_OUT and V _{SS} to have a 1% accuracy is 1.5 MΩ
$C_{LOAD}^{(2)}$	Capacitive load	-	-	50	pF	Maximum capacitive load at DAC_OUT pin (when the buffer is ON).
DAC_OUT min ⁽²⁾	Lower DAC_OUT voltage with buffer ON	0.2	-	-	V	<p>It gives the maximum output excursion of the DAC.</p> <p>It corresponds to 12-bit input code (0x0E0) to (0xF1C) at V_{REF+} = 3.6 V and (0x1C7) to (0xE38) at V_{REF+} = 1.8 V</p>
DAC_OUT max ⁽²⁾	Higher DAC_OUT voltage with buffer ON	-	-	$V_{DDA} - 0.2$	V	

■ DAC output voltage

- Digital values are converted to output voltages on a linear conversion between 0 and V_{REF+}
- Analog output voltages on each DAC channel pin are determined by the following equation:

$$DAC_{output} = V_{REF+} * DOR / 4095$$

- DOR is the digital value that should be converted
- Use of an external V_{REF+} can help to improve results.
 - ▶ By choosing a smaller V_{REF+} , the minimal “analog step” is reduced
 - ▶ Using a dedicated pin allows the use of less noisy references

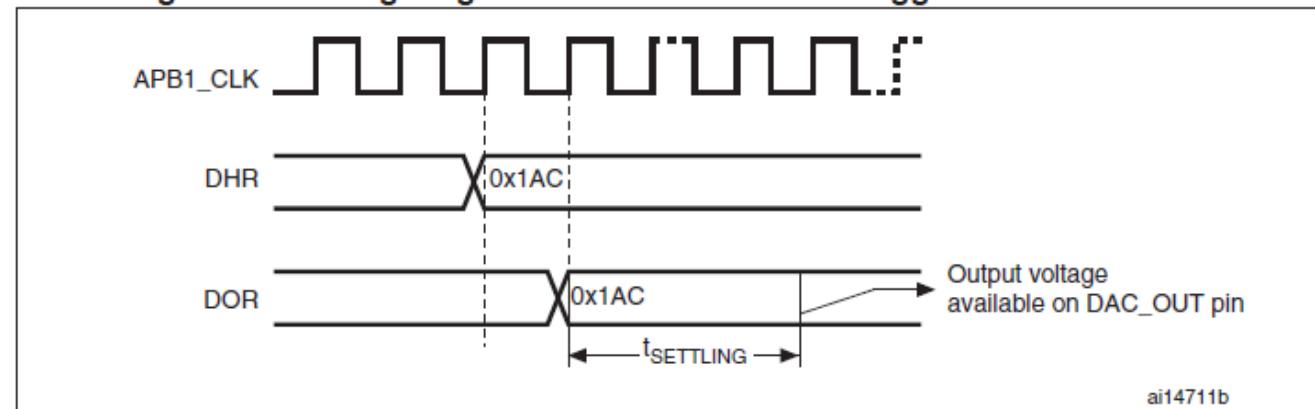
■ Automatic waveforms generation

- DAC can be set to produce noise/triangular signals
 - Useful for testing

■ Conversion timing

- DAC_DORx cannot be written directly
 - Data transfer to the DAC performed by loading DAC_DHRx register
 - Data in DAC_DHRx transferred to DAC_DORx after one APB1 clock cycle
 - If HW trigger selected, transfer performed 3 APB1 clock cycles after trigger
- When DAC_DORx loaded with DAC_DHRx contents, the analog output voltage available after a time $t_{SETTLING}$
 - $t_{SETTLING}$ depends on power supply voltage and analog output load.

Figure 67. Timing diagram for conversion with trigger disabled $TEN = 0$



■ Conversion timing

- $t_{SETTLING}$ is given in the table. Value higher when many bits change
- With changes of 1LSB, update rates of 1MS/s are possible

Table 87. DAC characteristics (continued)

Symbol	Parameter	Min	Typ	Max	Unit	Comments
$t_{SETTLING}^{(4)}$	Settling time (full scale: for a 10-bit input code transition between the lowest and the highest input codes when DAC_OUT reaches final value ± 4 LSB	-	3	6	μs	$C_{LOAD} \leq 50 \text{ pF}$, $R_{LOAD} \geq 5 \text{ k}\Omega$
$THD^{(4)}$	Total Harmonic Distortion Buffer ON	-	-	-	dB	$C_{LOAD} \leq 50 \text{ pF}$, $R_{LOAD} \geq 5 \text{ k}\Omega$
Update rate ⁽²⁾	Max frequency for a correct DAC_OUT change when small variation in the input code (from code i to i+1LSB)	-	-	1	MS/s	$C_{LOAD} \leq 50 \text{ pF}$, $R_{LOAD} \geq 5 \text{ k}\Omega$

Some DAC characteristics

Current consumption, offset error are examples of important characteristics

Table 87. DAC characteristics (continued)

Symbol	Parameter	Min	Typ	Max	Unit	Comments
$I_{DDA}^{(4)}$	DAC DC VDDA current consumption in quiescent mode ⁽³⁾	-	280	380	μA	With no load, middle code (0x800) on the inputs
		-	475	625	μA	With no load, worst code (0xF1C) at $V_{REF+} = 3.6$ V in terms of DC consumption on the inputs
$DNL^{(4)}$	Differential non linearity Difference between two consecutive code-1LSB)	-	-	± 0.5	LSB	Given for the DAC in 10-bit configuration.
		-	-	± 2	LSB	Given for the DAC in 12-bit configuration.
$INL^{(4)}$	Integral non linearity (difference between measured value at Code i and the value at Code i on a line drawn between Code 0 and last Code 1023)	-	-	± 1	LSB	Given for the DAC in 10-bit configuration.
		-	-	± 4	LSB	Given for the DAC in 12-bit configuration.
$Offset^{(4)}$	Offset error (difference between measured value at Code (0x800) and the ideal value = $V_{REF+}/2$)	-	-	± 10	mV	Given for the DAC in 12-bit configuration
		-	-	± 3	LSB	Given for the DAC in 10-bit at $V_{REF+} = 3.6$ V
		-	-	± 12	LSB	Given for the DAC in 12-bit at $V_{REF+} = 3.6$ V

DAC Registers

Table 76. DAC register map

Offset	Register name	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	DAC_CR	Reserved		DMAUDRIE2	DMAEN2	MAMP2[3:0]	WAVE2[2:0]	TSEL2[2:0]	TEN2	BOFF2	EN2	Reserved		DMAUDRIE1	DMAEN1	MAMP1[3:0]	WAVE1[2:0]	TSEL1[2:0]	TEN1	SWTRIG2	BOFF1	EN1											
0x04	DAC_SWTRIGR																																
0x08	DAC_DHR12R1																																
0x0C	DAC_DHR12L1																																
0x10	DAC_DHR8R1																																
0x14	DAC_DHR12R2																																
0x18	DAC_DHR12L2																																
0x1C	DAC_DHR8R2																																

Register with offset <0x20 - 0x34> are not relevant for this class

- Once the DAC channelx is enabled, the corresponding GPIO pin (PA4 or PA5) is automatically connected to the analog converter output (DAC_OUTx)
- In order to avoid parasitic consumption, the PA4 or PA5 pin should first be configured to analog (AIN)

DAC Registers

Table 76. DAC register map

Offset	Register name	31	30	29	DMAUDRIE2	28	DMAEN2	27	26	25	24	23	22	21	20	19	18	17	BOFF2	16	EN2	15	14	DMAUDRIE1	13	DMAEN1	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	DAC_CR	Reserved		DMAUDRIE2	MAMP2[3:0]	WAVE2[2:0]	TSEL2[2:0]	TEN2	BOFF2	EN2	Reserved		MAMP1[3:0]	WAVE1[2:0]	TSEL1[2:0]	TEN1	SWTRIG2	BOFF1	EN1																				
0x04	DAC_SWTRIGR	SWTRIG software trigger DAC1/2 1 → SW trigger enabled								Reserved	Cleared by HW once DHR loaded into DOR. Also disabled by SW								SWTRIG1																				
0x08	DAC_	Reserved												DACC1DHR[11:0]																									

Command Register

EN (Enable) 1 → DAC enabled

BOFF (Buffer Off) 1 → output buffer disabled, 0 → output buffer enabled

TEN (Trigger enable) DAC Channel trigger 0 → disabled, 1 → enabled,

TSEL (Select ext trigger, TEN must be 1) 111 → select Swtrigger

WAVE sel wave generator keep at 00 → Wave generation disabled

MAMP select mask/amplitude in waveform generation keep at 000 if no wavegeneration

DMAEN, DMAUDRIE Keep at 0. not relevant for this class

■ Steps needed to get DAC running

- Address of register: DAC Base Address + Register Offset
 - DAC range is <0x4000 7400 - 0x4000 77FF>
- Choosing a DAC
 - Choose the DAC you want to use
 - Configure port pin(s) accordingly. Do you need the output buffer?
- Setting some parameters (conversion mode, clocking, ...)
 - Decide how to clock the DAC. Set trigger source
 - ▶ Set up sources to trigger conversion start
 - ▶ Or CPU to control the conversion
- Starting the DAC, CPU mode
 - Switch on the DAC, write the data in the data register.
 - Wait (the time interval you want)
 - Write the next sample

Using the DAC (example in assembler)

```
; Configuration
init_dac    ; Clock configuration
LDR R6, =REG_RCC_AHB1ENR
LDR R7, =0x1      ; Enable GPIOA clock
BL set_sfr

LDR R6, =REG_RCC_APB1ENR
LDR R7, =0x20000000 ; Enable DAC clock
BL set_sfr

; Analog pin configuration (PA.4)
LDR R6, =REG_GPIOA_MODER
LDR R7, =0x300
BL set_sfr

; DAC configuration
LDR R6, =REG_DAC_CR
LDR R7, =0x1      ; Enable ch. 1
BL set_sfr

LDR R6, =REG_DAC_DHR8R1
LDR R7, =0x0      ; Set initial value (=0)
BL set_sfr

; Starting conversion
LDR R6, =REG_DAC_DHR8R1
LDR R0, [R6]
LDR R1, =0xff
BICS R0, R0, R1      ; Clear bits
STR R0, [R6]
BL set_sfr           ; Set value in R7
```

- **ADC/DAC used to interface the analog and digital world**
 - Applications such as instrumentation, audio, control
- **Rate at which data is converted and number of bits used are important parameters**
- **Devices introduce errors that affect results of conversion**
- **STM32F429 provides ADC and DAC with several features**
 - Those features and the way to use them are found in the datasheet and reference manual
- **ADC and DAC have analog and digital parts**
 - They are known as mixed-signal devices

- Full of ADCs and DACs (DACs and CADs) (dogs and cats)
- When you have forgotten everything about ADCs and DACs

Just remember the music



Sources: <http://www.recordproduction.com/multimedia.htm> <http://rayhiltz.com/wp-content/uploads/2011/07/69-Drunk-Karaoke-Cats-568x384.jpg>

■ References

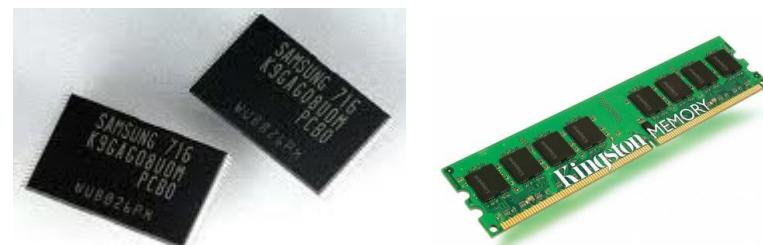
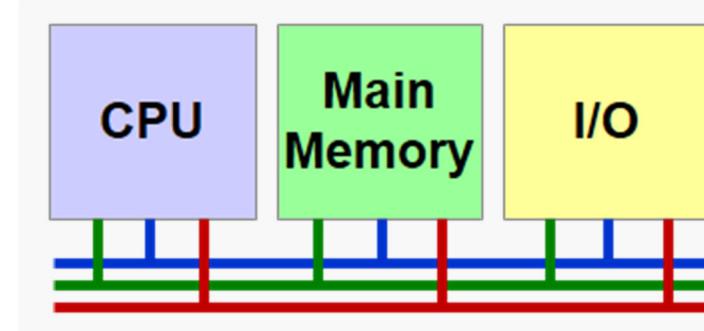
Documents used in this lesson

- [1] STM32F42xxx Datasheet
- [2] STM32F42xxx Reference manual (RM0090)
- [3] AN3116 “STM32™’s ADC modes and their applications”
- [4] <http://www.maximintegrated.com/en/app-notes/index.mvp/id/641>
- [5] Atmel AVR127: Understanding ADC Parameters

Memory

Computer Engineering 2

Storing and retrieving data



- **Memory Technologies**
 - PROM, EEPROM and flash, SRAM, SDRAM
- **On-CHIP Memories STM32F429ZISRAM**
 - SRAM and Flash
- **External Memory (Off-Chip)**
 - Flexible Memory Controller
- **Appendix: Trends and Figures**

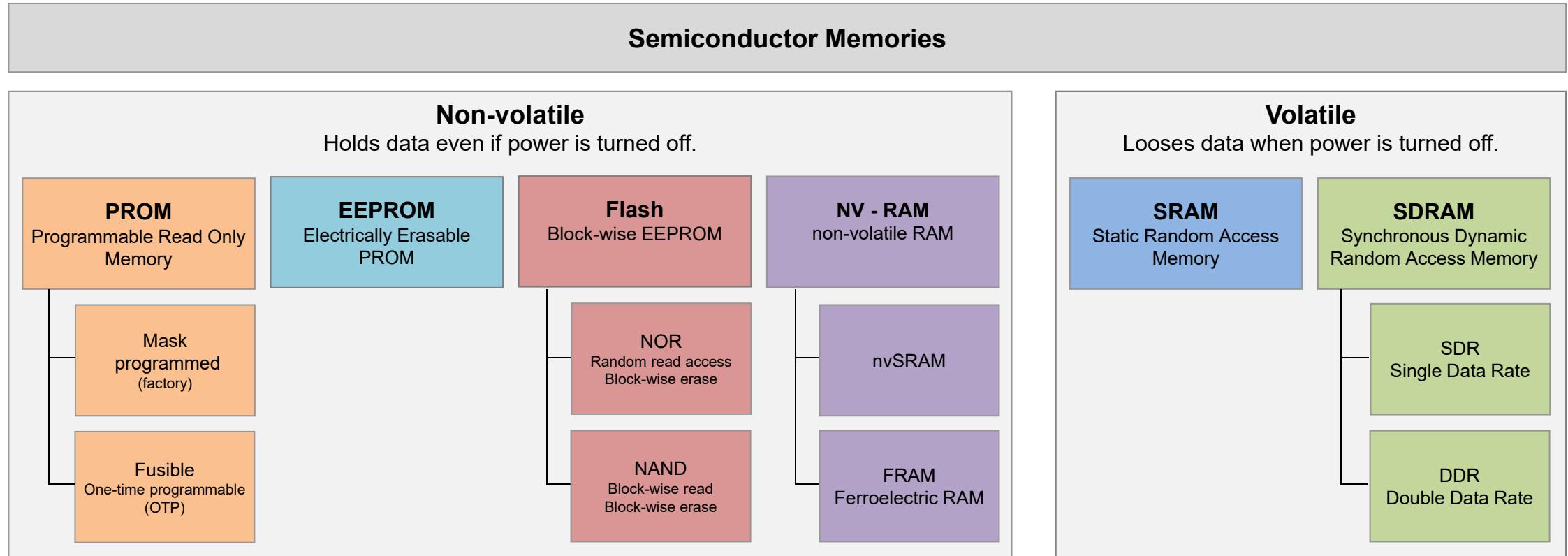
Learning Objectives

At the end of this lesson you will be able

- to classify widely used memory technologies
- to discuss the structure and function of an SRAM (static RAM)
- to discuss the structure and function of flash memory
- to outline the structure and function of an asynchronous SRAM device
- to outline how an external asynchronous SRAM device can be connected through the flexible memory controller (FMC)
- to explain how an internal 32-bit access is partitioned into several external half-word or byte accesses
- to interpret timing diagrams for read and write accesses to external, asynchronous SRAMs
- to summarize the differences between a NOR and a NAND flash
- to summarize the differences between a static RAM (SRAM) and a dynamic RAM (SDRAM)

Semiconductor Fundamentals

MEMORY TECHNOLOGIES



■ Unit Symbols

- b = bit B = Byte

■ Memory Chips

- Binary prefixes according to JEDEC¹⁾ and IEC²⁾
 - Kilo K = 1024
 - Mega M = 1024×1024 = 1'048'510
 - Giga G = $1024 \times 1024 \times 1024$ = 1'073'741'824

■ Hard Disks

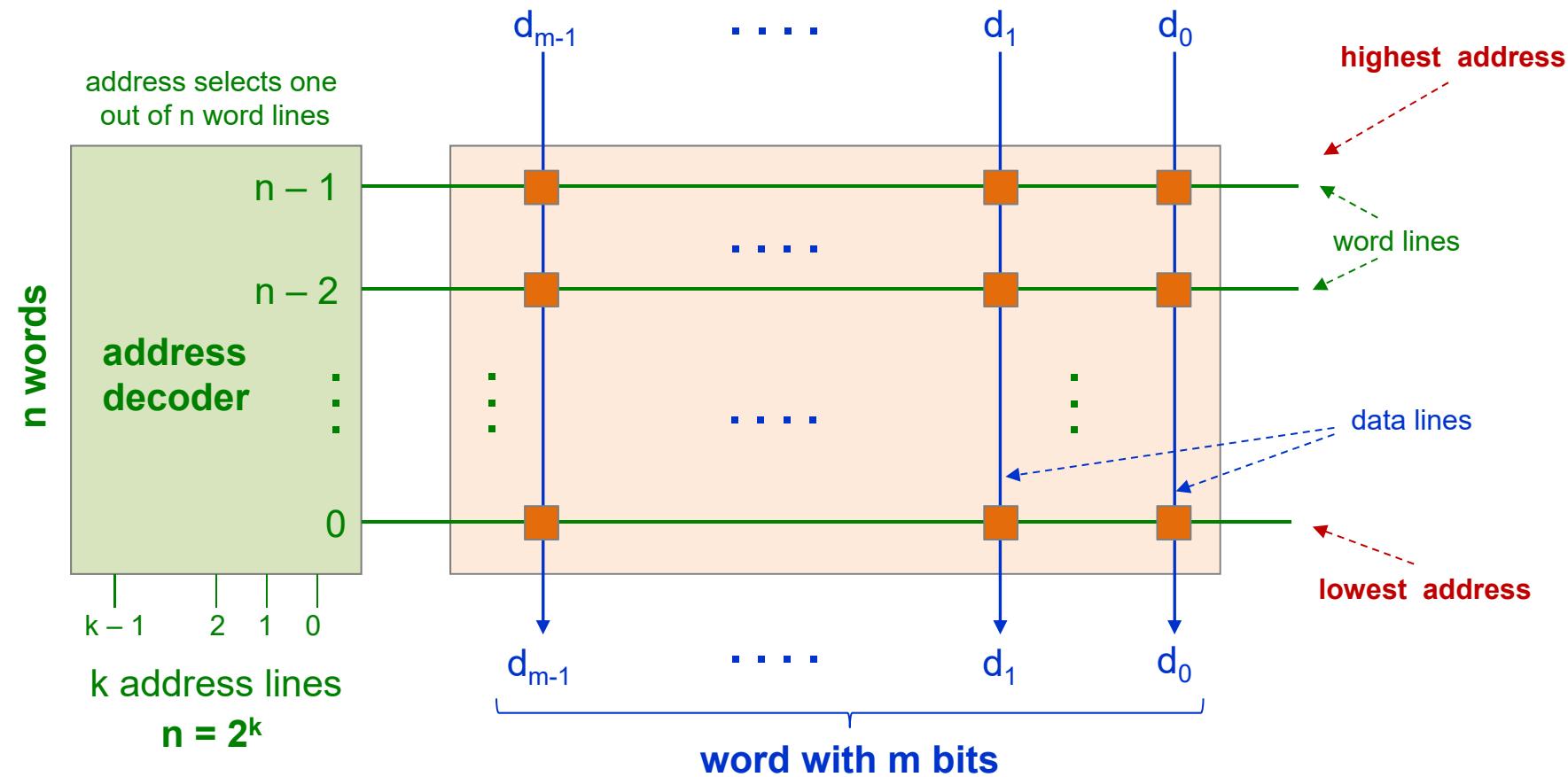
- Often use SI (or metric) prefixes
 - Kilo k = 1000
 - Mega M = 1000×1000
 - Giga G = $1000 \times 1000 \times 1000$

1) JEDEC Solid State Technology Association
2) International Electrotechnical Commission

Memories Are Arrays of Bit Cells

■ Memory Architecture → $n \times m$ array

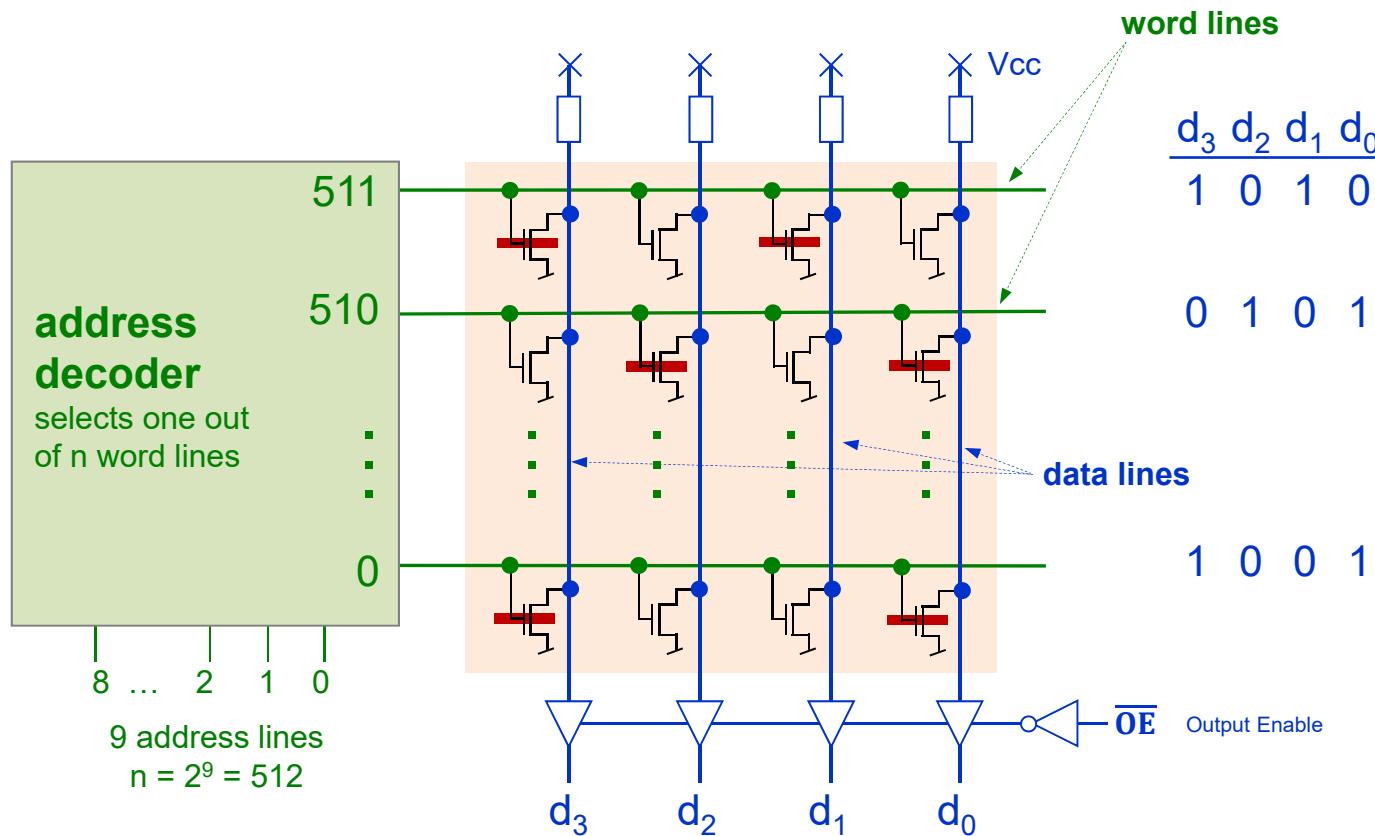
- n words with m data bits



PROM – Programmable Read Only Memory

■ $n \times m$ array

n = number of word lines
 m = number of bit lines



→ n addresses with m data bits

Example 512 x 4 bit

Fusible Transistors

Programming applies higher voltage
to destroy transistors (blow fuses)
Process is not reversible

Word line = '1'
→ transistor shortens bit line to GND

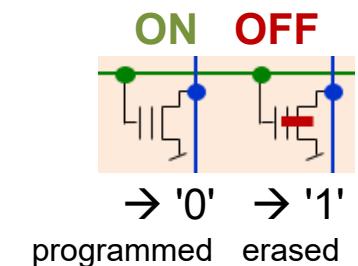
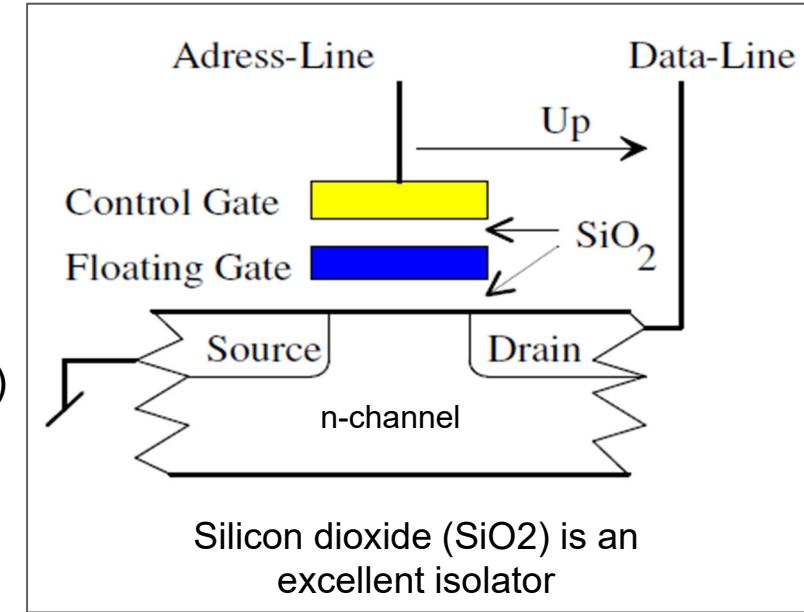
Word line = '0'
→ transistor is open; pull-up pulls bit line to Vcc

Transistor destroyed (fused)
→ always open, i.e. pull-up pulls bit line to Vcc

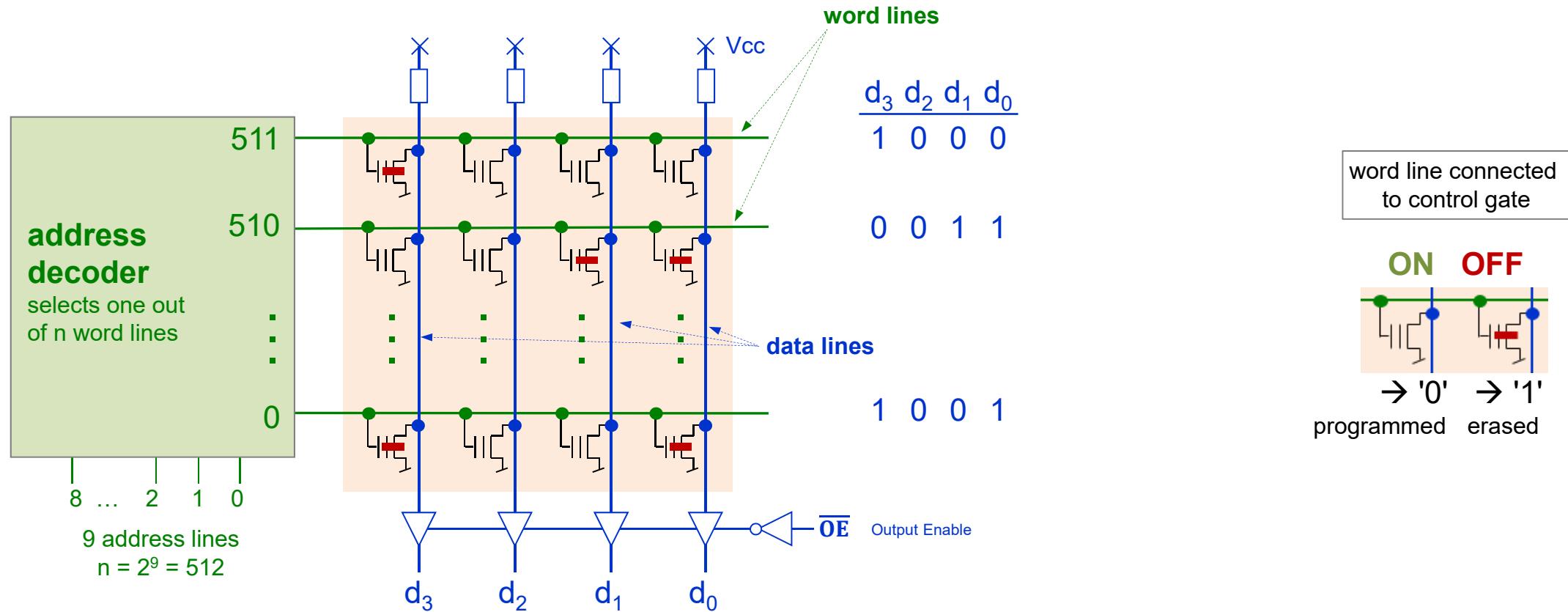
EEPROM and Flash

■ Making PROMs Reprogrammable

- "Floating Gate" transistor
 - Replace fusing by reprogrammable "Floating Gate"
- Write cell to '0' → ON
 - High voltage Up deposits charge on floating gate (isolated by SiO₂)
 - Transistor ON (conducting) if control gate equal '1'
- Erase cell to '1' → OFF
 - Discharge floating gate with negative Up
 - Transistor is OFF, i.e. blocking independent of value on control gate
- EEPROM
 - High cell area → low density, high cost per bit
- Flash
 - Erasing can only be done for whole sectors
→ small cell area, high density, low cost per bit

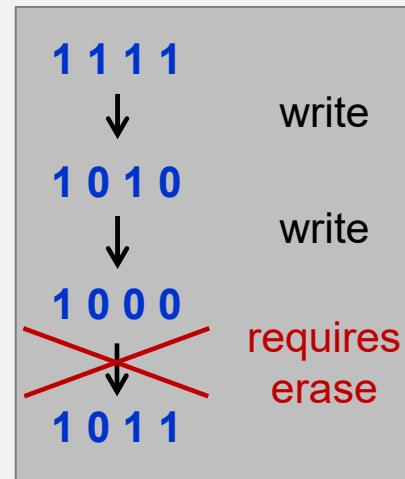


- Use ‘Floating Gates’ instead of ‘Fusible Transistors’



■ Write Operations (Programming)

- Can only change bits from '1' to '0'
 - Otherwise an erase operation is required
- Word, half-word or byte access possible
- Writing a double word ~16 us
 - I.e. around 1000 times slower than SRAM



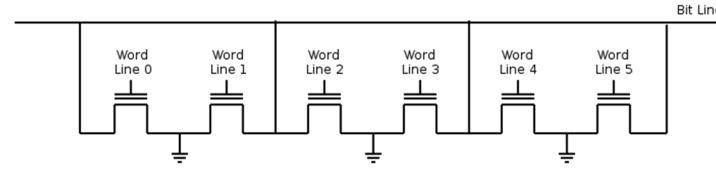
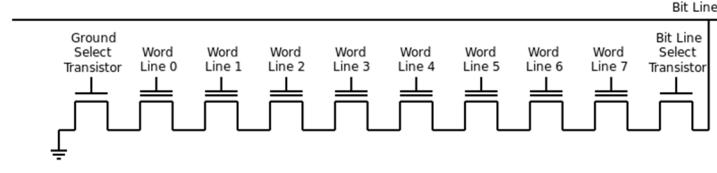
■ Erase Operations

- Change all bits from '0' to '1'
 - **Only possible by sector or by bank**, not on a word
 - Typical sector sizes of 16
- Erase of a 128 Kbytes sector takes between 1 and 2 seconds ¹⁾
- Endurance: 10'000 erase cycles ²⁾
- Sector may not be accessed (write or read) during erase
 - I.e. execute program from another sector or from SRAM during erase

1) Depending on supply voltage and configuration parameters

2) Value from STM32F429ZI datasheet

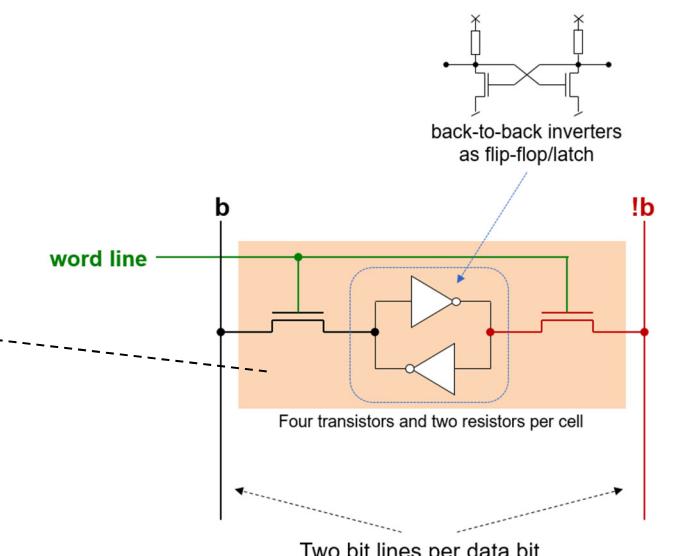
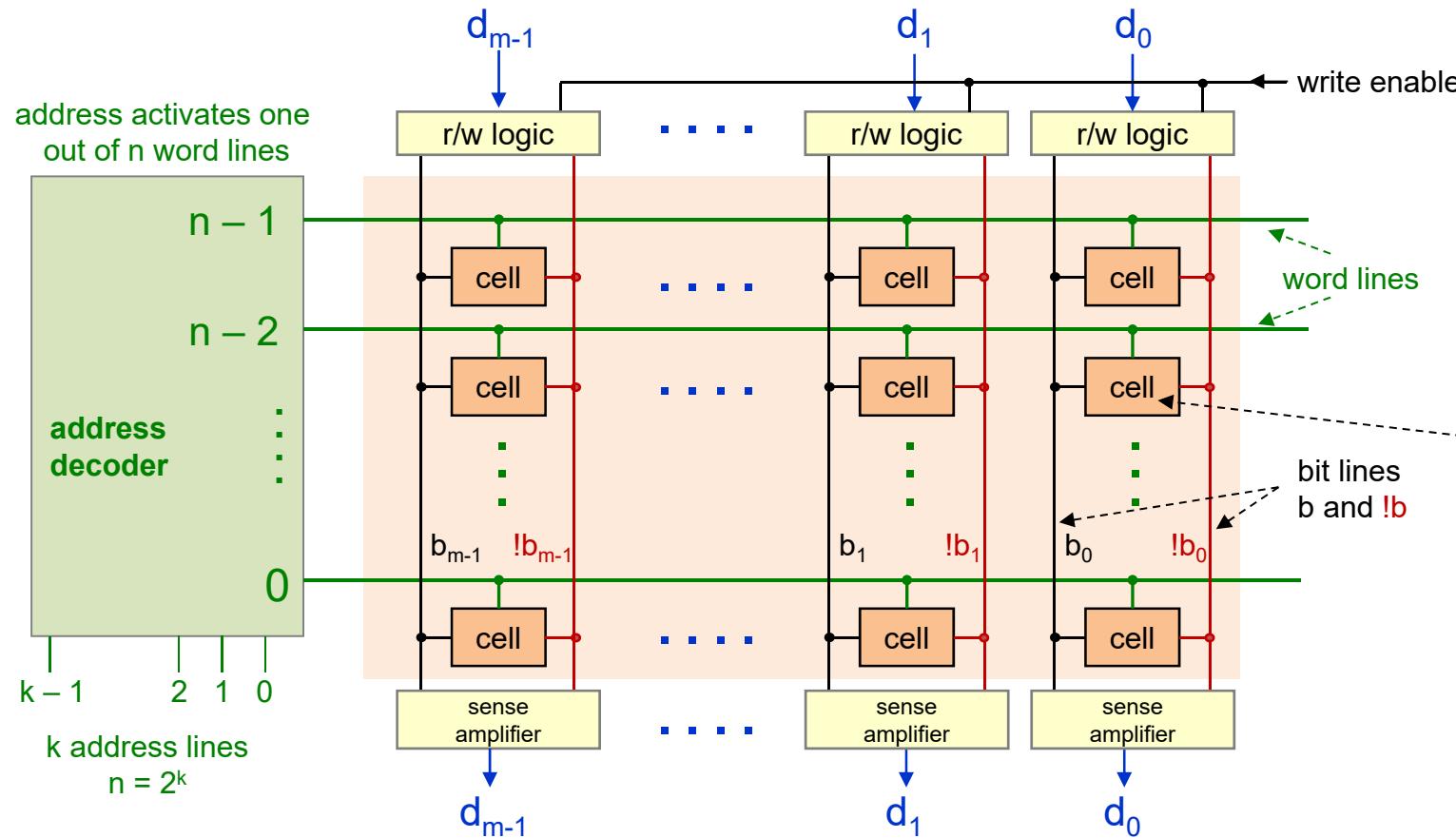
Flash – NOR vs NAND Topology

NOR Flash		NAND Flash
Topology		
Applications	<ul style="list-style-type: none"> Execute code directly from memory Persistent device configurations (replacement of EEPROM) 	<ul style="list-style-type: none"> File-based IO, disks Large amounts of sequential data (images, SD cards, SSD) Load programs into RAM before executing
Density	Medium Up to 2 Gbit = 256 MByte	High Up to 1 Tbit
Interface	<ul style="list-style-type: none"> Read same as asynchronous SRAM Types with serial interface available 	<ul style="list-style-type: none"> Special NAND flash interface Error correction for defective blocks
Access	<ul style="list-style-type: none"> Random access read ~0.12 µs Writing individual bytes possible Slow writes ~180 µs / 32 Byte 	<ul style="list-style-type: none"> Slow random access read: 1. Byte 25 µs, then 0.03 µs each Writing of individual bytes difficult Fast block write ~300 µs / 2'112 Bytes

SRAM – Static Random Access Memory

■ $n \times m$ SRAM Architecture

→ flip-flop (latch) based cells



Example for NMOS: N-type metal-oxide-semiconductor logic

SRAM – Static Random Access Memory

Writing a Row (Word)

Set bit lines b and !b to (1, 0) or (0, 1) respectively



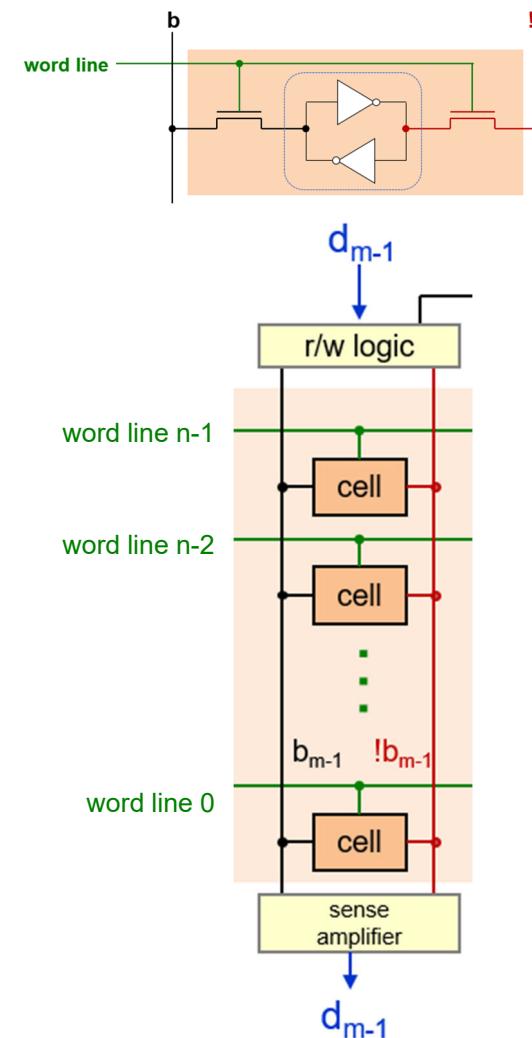
Set the addressed word line to 1



Data is stored in cells



Set word line to 0



Reading a Row (Word)

Pre-charge both bit lines b and !b to 1



Briefly set word line to 1



Inverters pull either b or !b towards (not to) ground

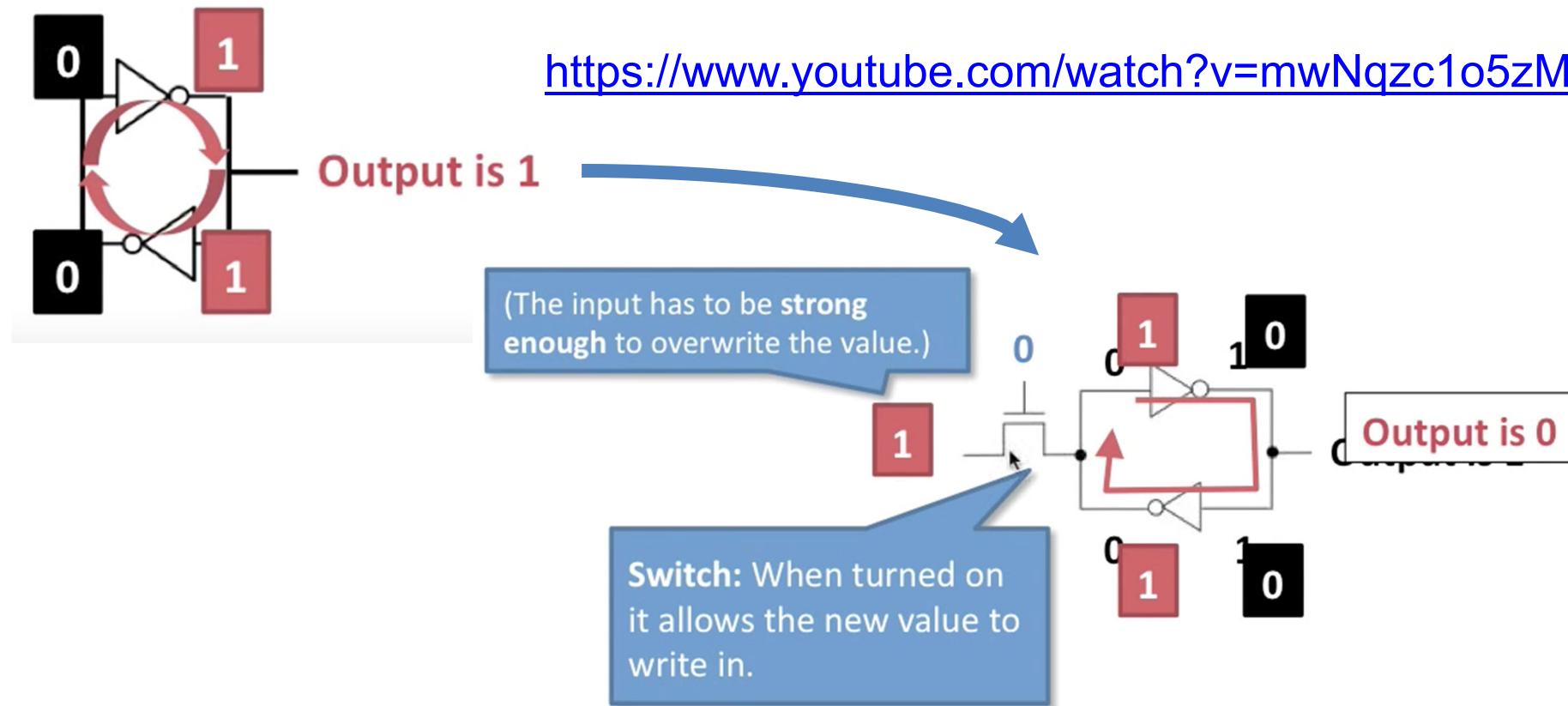


Sense amplifier amplifies small voltage difference between lines b and !b

SRAM – Static Random Access Memory

■ Structure of SRAM cell in NMOS¹⁾

- Flip-flop (latch) based structure, change from '1' to '0'



■ Read and write

- All accesses take roughly the same time
- Access time independent of location of data item in memory
- Access time independent of previous access¹⁾

■ Volatile

- Memory content retained only as long as device is powered

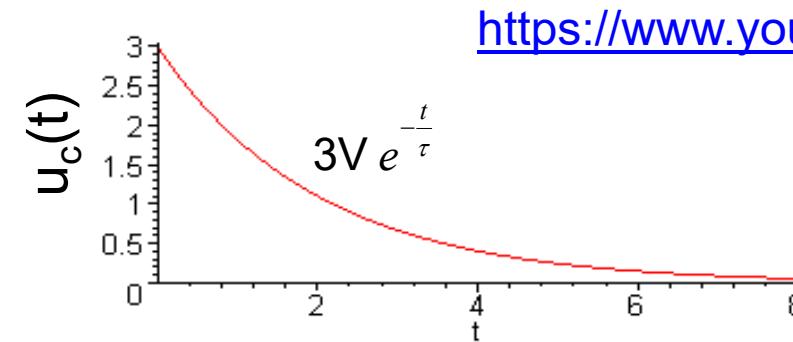
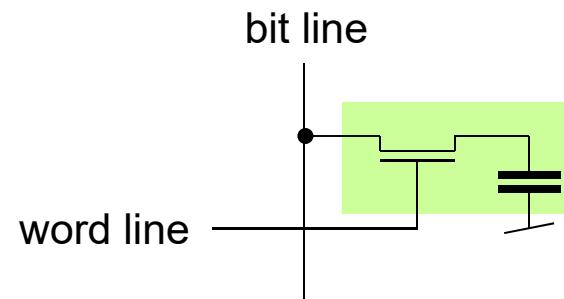
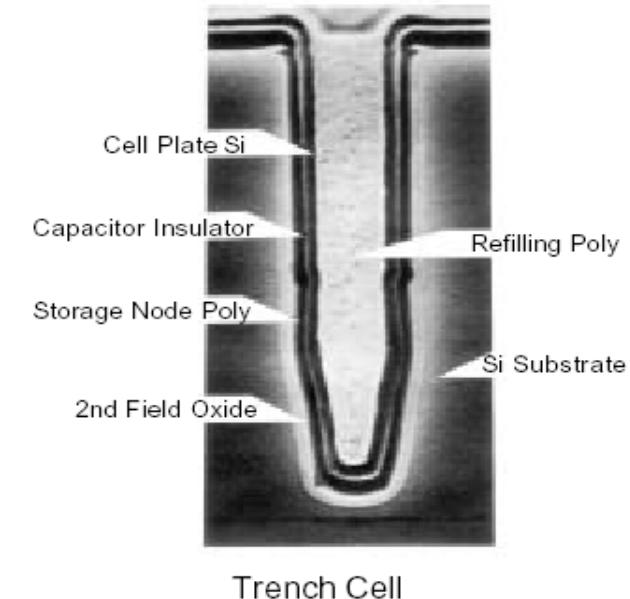
■ Static

- Storage elements similar to flip-flops / latches
- No refresh required
 - refresh: periodic reading and rewriting of memory cell to maintain the content

1) as opposed to a DRAM that favors burst accesses

■ Synchronous Dynamic Random Access Memory

- Information stored as charge in capacitor
- High integration
 - Large memories at low cost
 - Allows to store large amounts of data
- Leakage current → Loss of charge
 - Capacitor holds charge only for a few milliseconds
 - Charge has to be refreshed periodically → dynamic
 - Refresh logic usually located on SDRAM device

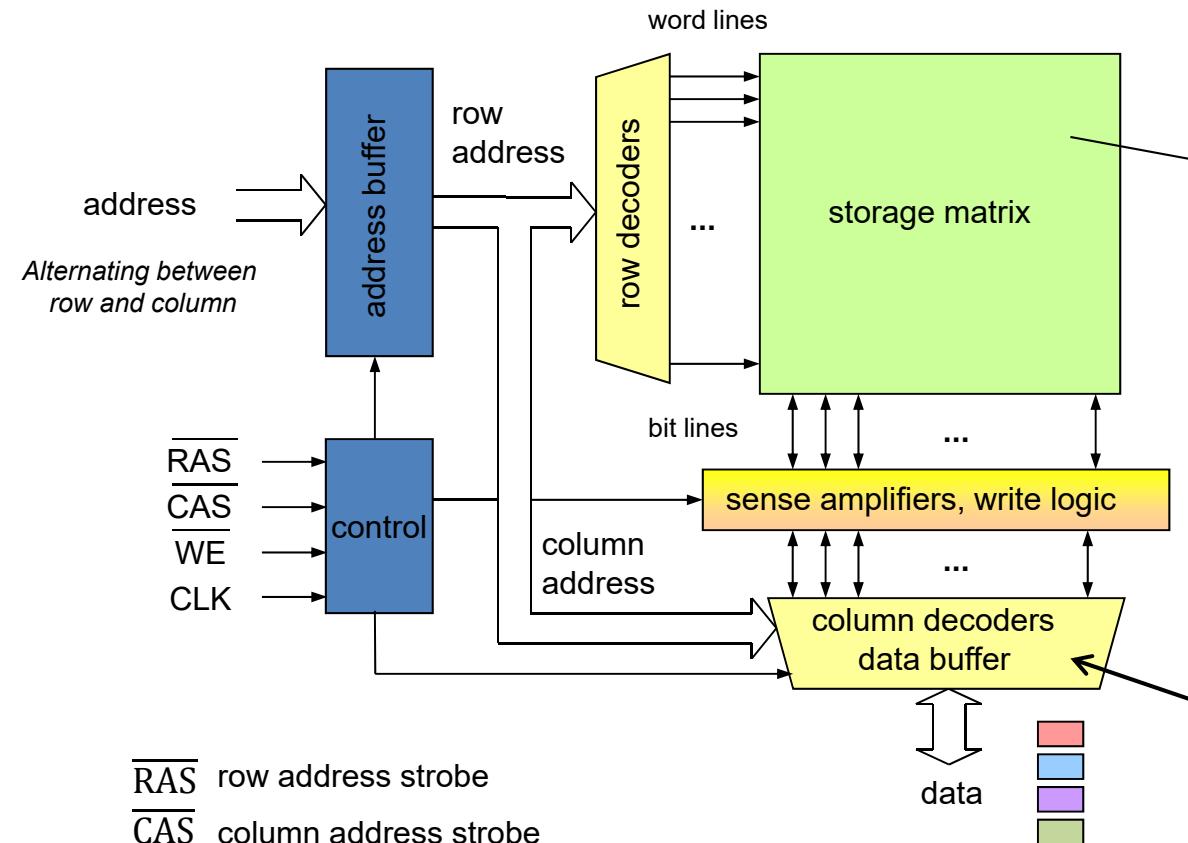


<https://www.youtube.com/watch?v=3s7zsLU83bY>

SDRAM – Synchronous Dynamic RAM

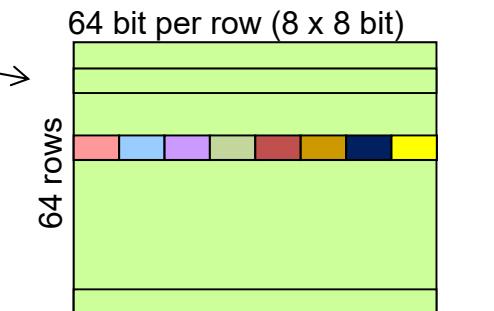
■ SDRAM Structure

- Row and column addresses multiplexed



Organization e.g.

512 x 8-bit
64 x 64 bit
row address
A ₅ .. A ₀
column address
A ₂ .. A ₀
data
D ₇ .. D ₀

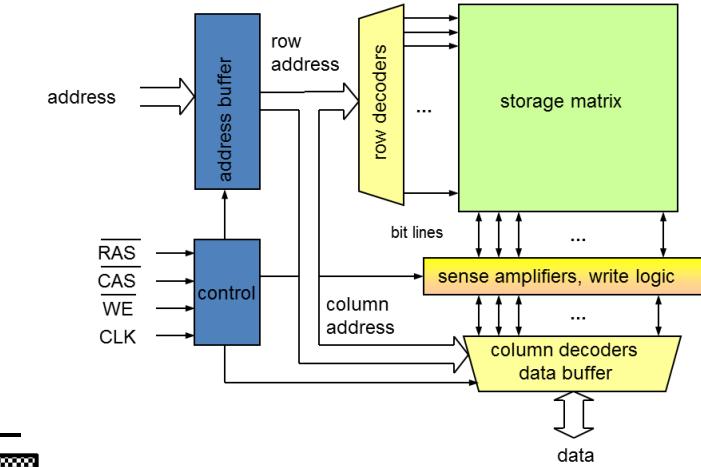
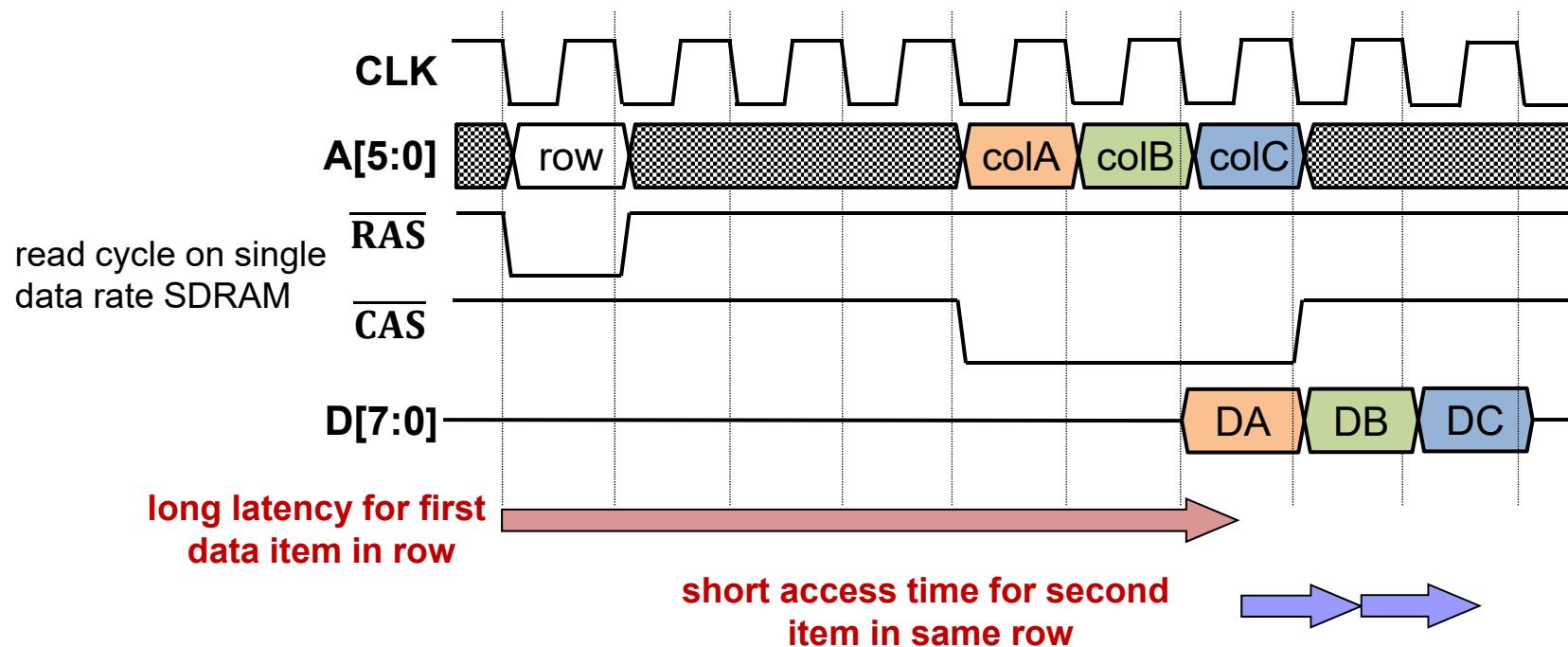


buffer: stores content of a complete row
acts as a cache

SDRAM – Synchronous Dynamic RAM

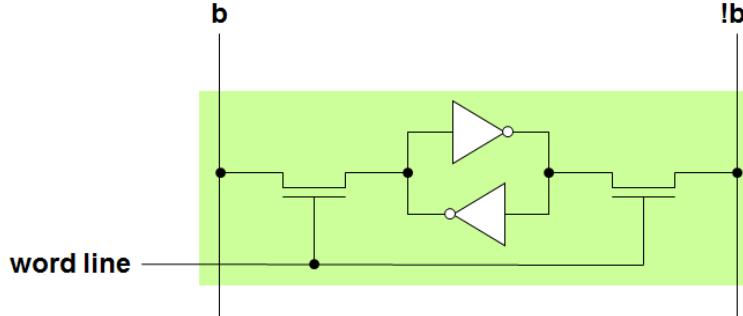
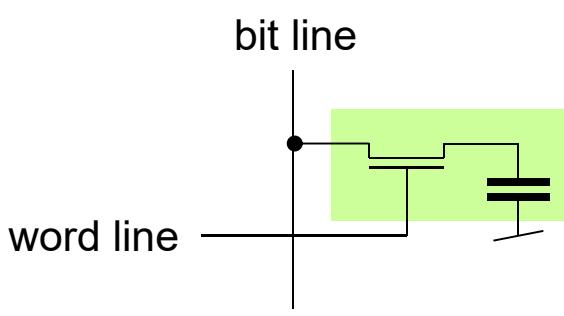
Synchronous Interface

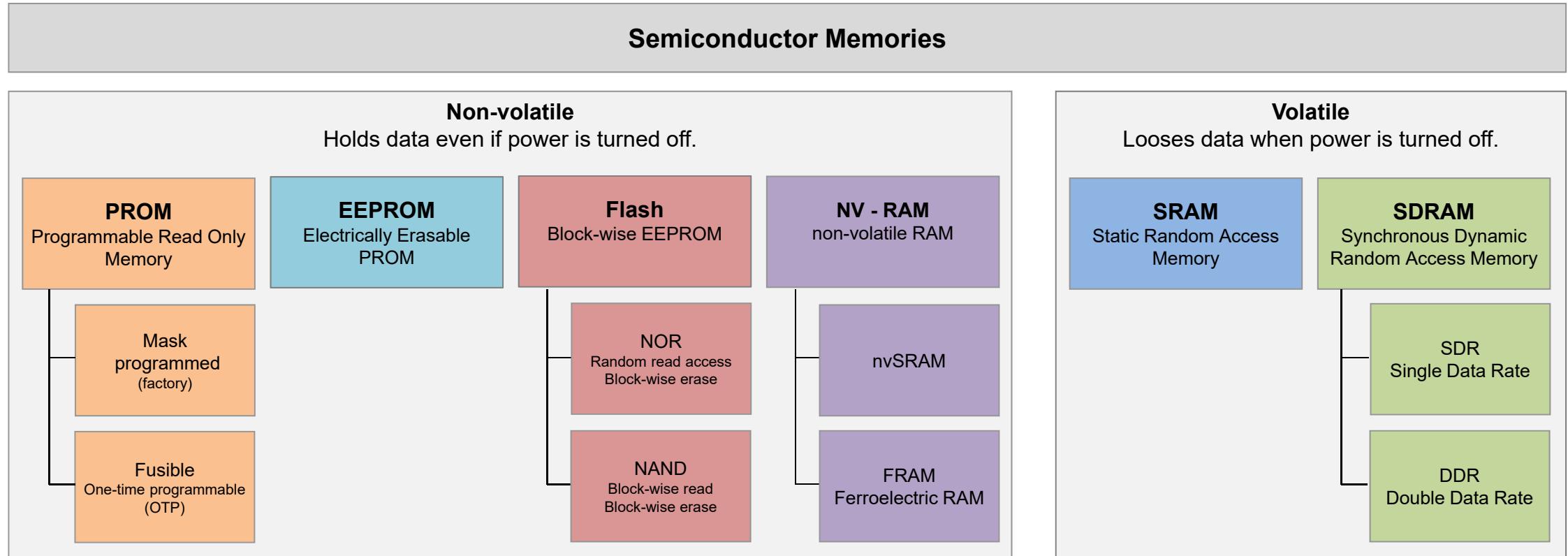
- Multiplexed row and column addresses
- Clocks up to 1200 MHz



RAS low → The master places the 6-bit row address on lines A[5:0].
CAS low → The master places the 3-bit column address on lines A[2:0]. Lines A[5:3] are unused.

SDRAM – Synchronous Dynamic RAM

Static RAM (SRAM)	Synchronous Dynamic RAM (SDRAM)
Flip-flop/latch → 4 Transistors / 2 resistors	Transistor and capacitor
	
Large cell <ul style="list-style-type: none">• Low density, high cost• Up to 64 Mb per device	Small cell <ul style="list-style-type: none">• High density, low cost• Up to 4 Gb per device
Almost no static power consumption <ul style="list-style-type: none">• Static i.e. no accesses taking place	Leakage currents <ul style="list-style-type: none">• Requires periodic refresh
Asynchronous interface (no clock) <ul style="list-style-type: none">• Simple connection to bus	Synchronous interface (clocked) <ul style="list-style-type: none">• Requires dedicated SDRAM Controller
All accesses take roughly the same time <ul style="list-style-type: none">• ~5ns per access → 200 MHz• Suitable for distributed accesses	Long latency for first access of a block <ul style="list-style-type: none">• Fast access for blocks of data (bursts)• Large overhead for single byte

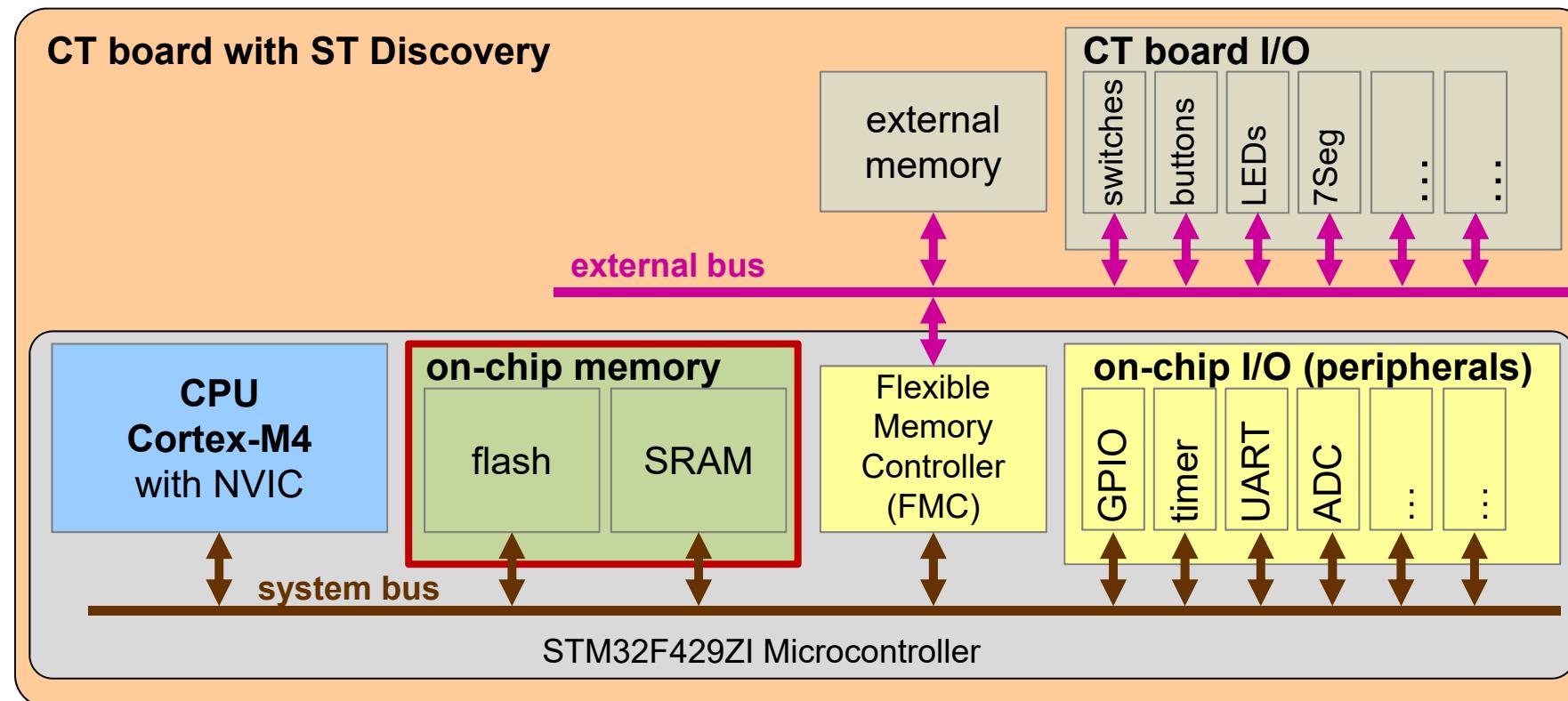


Our System

ON-CHIP MEMORIES STM32F429ZI

■ Simplified Model STM32F429ZI

- On-chip system bus 32 data lines, 32 address lines and control signals

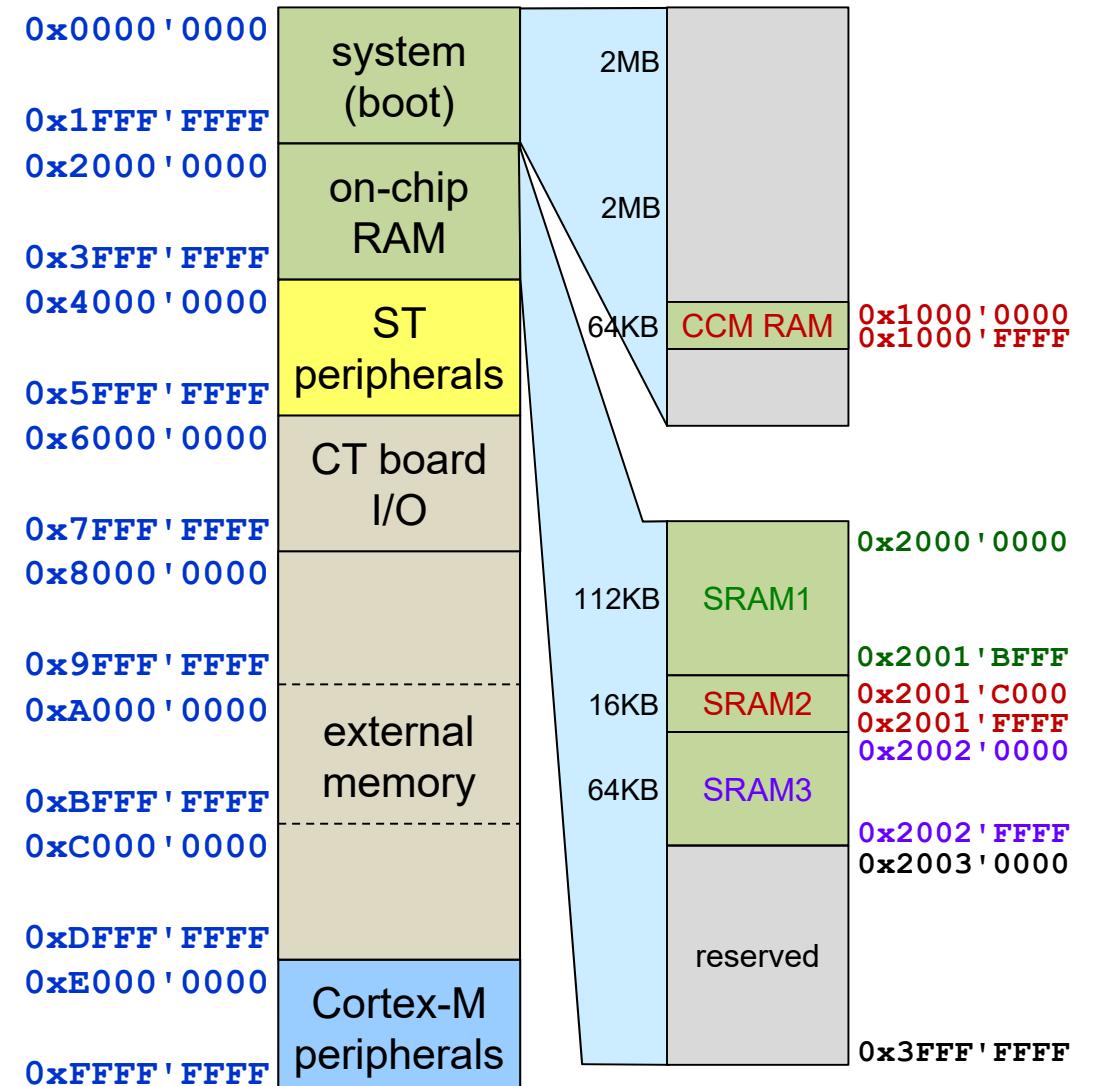


On-chip Memory: SRAM

■ Address Regions

- SRAM1 112K bytes
- SRAM2 16K bytes
- SRAM3 64K bytes
- CCM 64K bytes

CCM: Core Coupled Memory –
Fast memory exclusively addressable by the CPU.



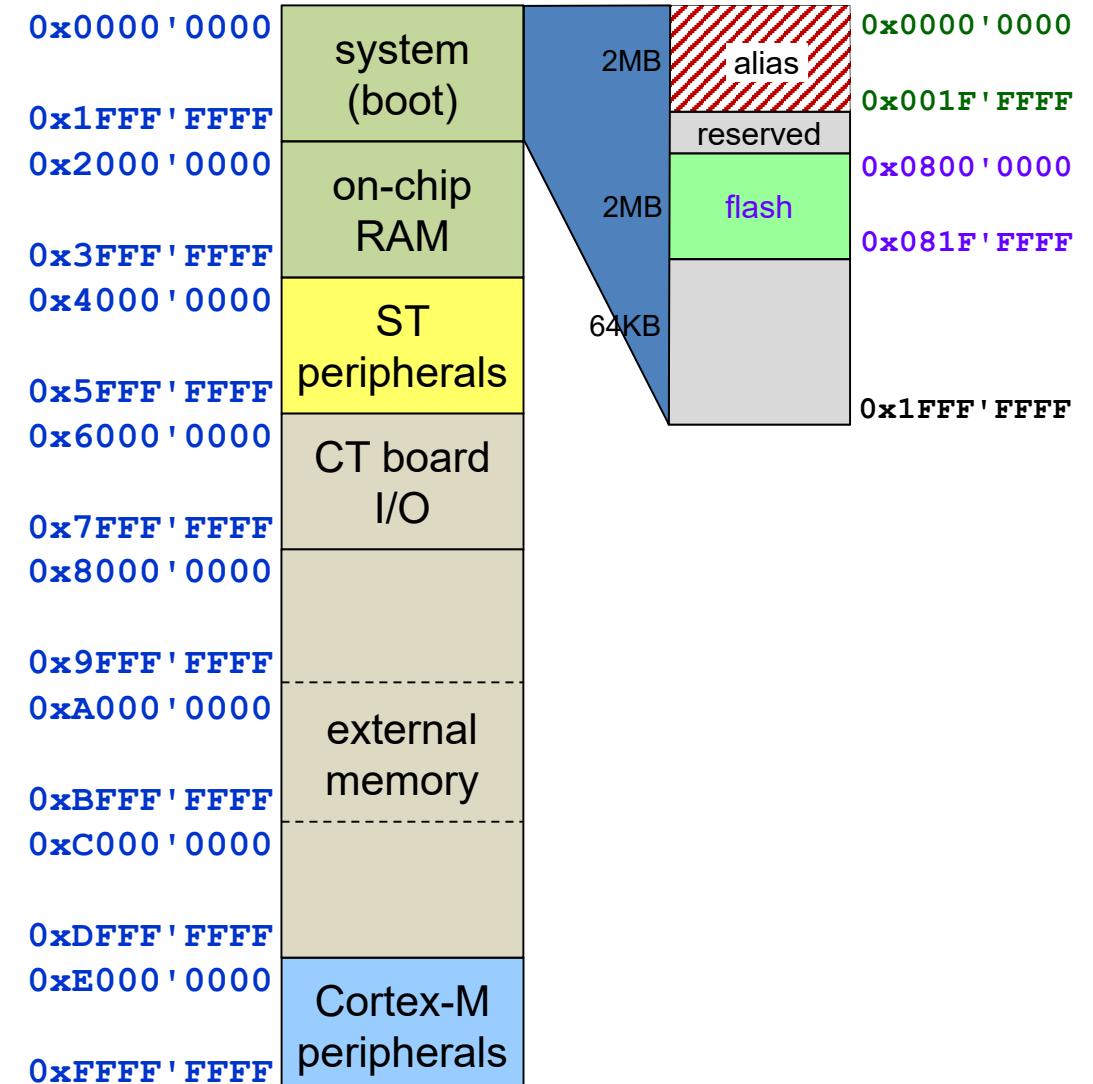
On-chip Memory: Flash

Flash

- Non-volatile memory
 - Memory content retained after power off
- Store code and persistent data
- NOR topology
 - Like most on-chip flash memories

Persistent Data denotes information that is infrequently accessed and not likely to be modified.

Source: Wikipedia



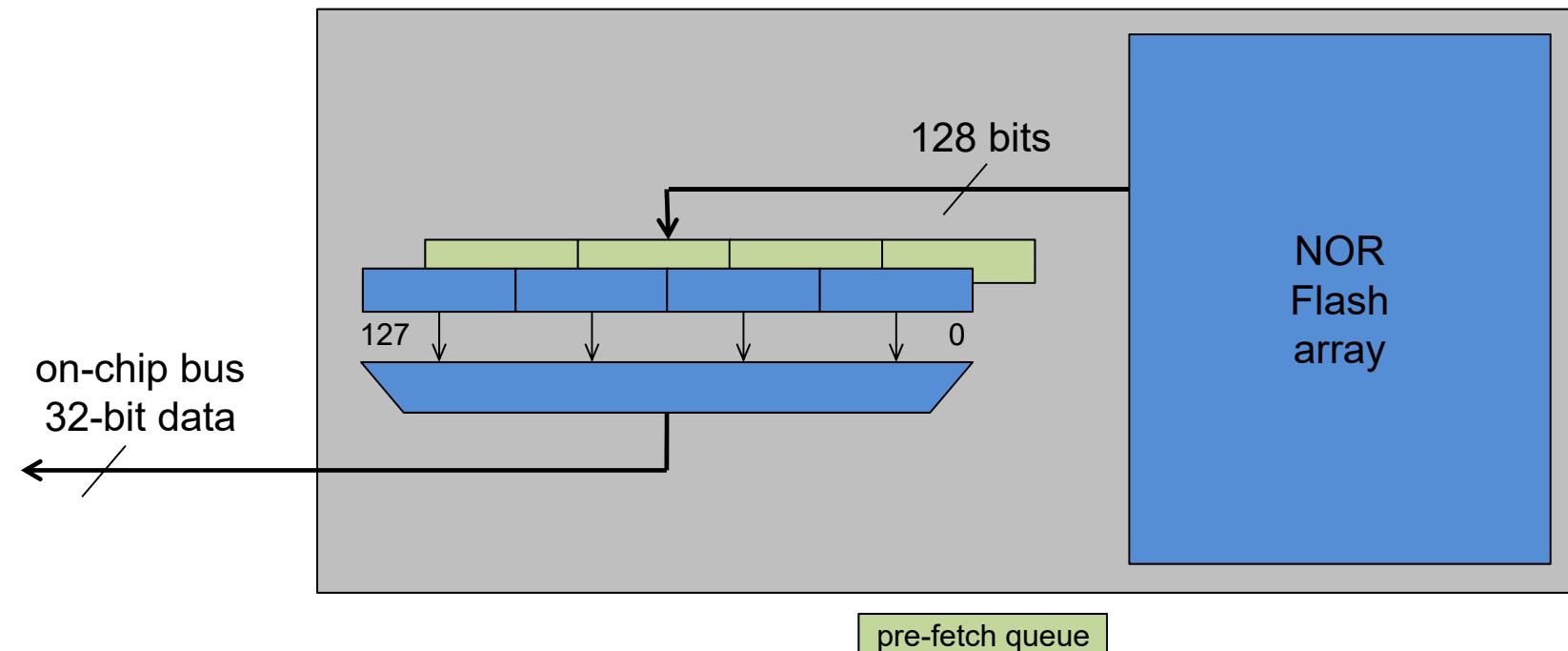
■ Flash Is Partitioned into Sectors

- Sectors can only be erased as a whole
- Writing through control registers – no direct memory write accesses

Bank 1	Sector 0	0x0800'0000 – 0x0800'3FFF	16 Kbytes	total 2 Mbytes
	Sector 1	0x0800'4000 – 0x0800'7FFF	16 Kbytes <th data-kind="ghost"></th>	
	Sector 2	0x0800'8000 – 0x0800'BFFF	16 Kbytes	
	Sector 3	0x0800'C000 – 0x0800'FFFF	16 Kbytes	
	Sector 4	0x0801'0000 – 0x0801'FFFF	64 Kbytes	
	Sector 5	0x0802'0000 – 0x0803'FFFF	128 Kbytes	
	
	Sector 11	0x080E'0000 – 0x080F'FFFF	128 Kbytes	
	Bank 2	Sector 12	0x0810'0000 – 0x0810'3FFF	16 Kbytes
		Sector 13	0x0810'4000 – 0x0810'7FFF	16 Kbytes
		Sector 14	0x0810'8000 – 0x0810'BFFF	16 Kbytes
		Sector 15	0x0810'C000 – 0x0810'FFFF	16 Kbytes
		Sector 16	0x0811'0000 – 0x0811'FFFF	64 Kbytes
		Sector 17	0x0812'0000 – 0x0813'FFFF	128 Kbytes
	
		Sector 23	0x081E'0000 – 0x081F'FFFF	128 Kbytes

■ Flash Has Higher Latency

- Read requires up to 8 Wait States¹⁾ on on-chip bus
- ST uses 128-bit buffer with pre-fetch queue
 - Reduces performance penalty when executing sequential instructions



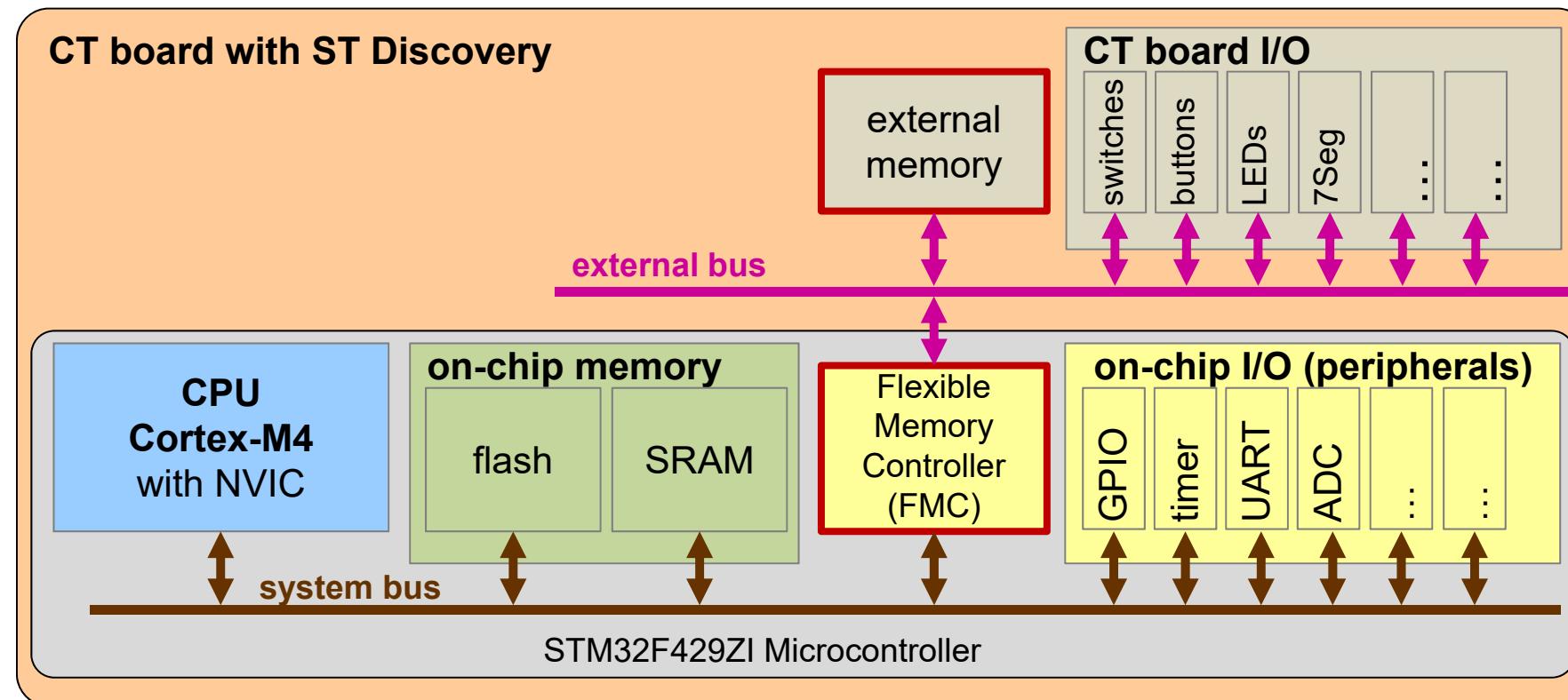
1) depending on clock frequency and supply voltage

Extending Our System

EXTERNAL MEMORY (OFF-CHIP)

■ Simplified Model STM32F429ZI

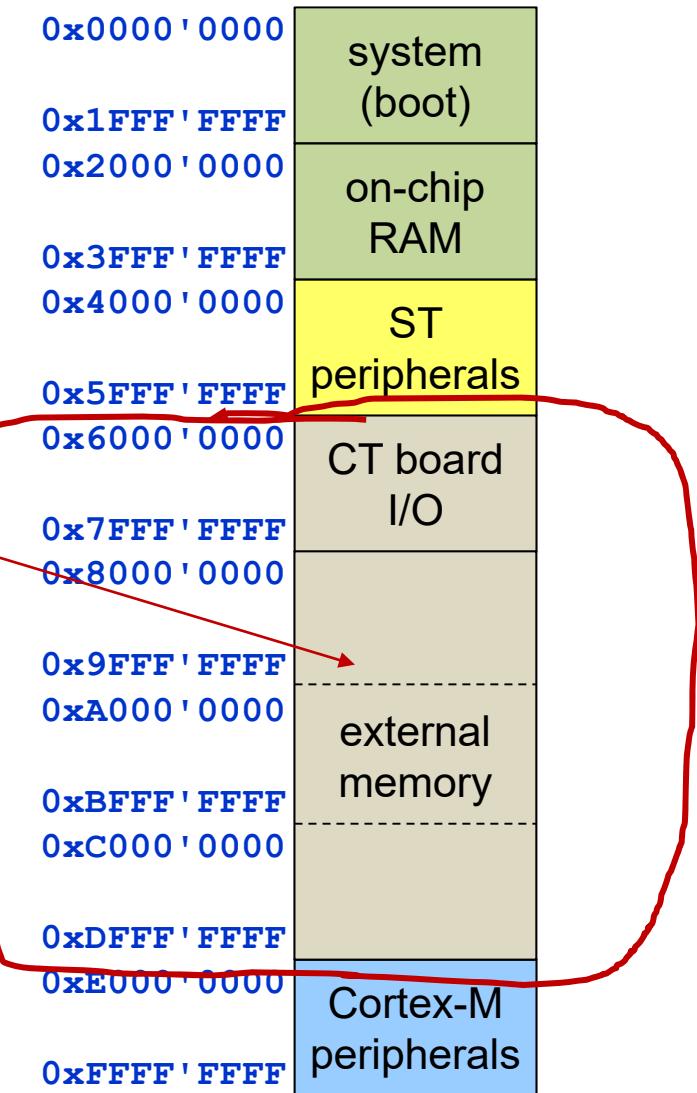
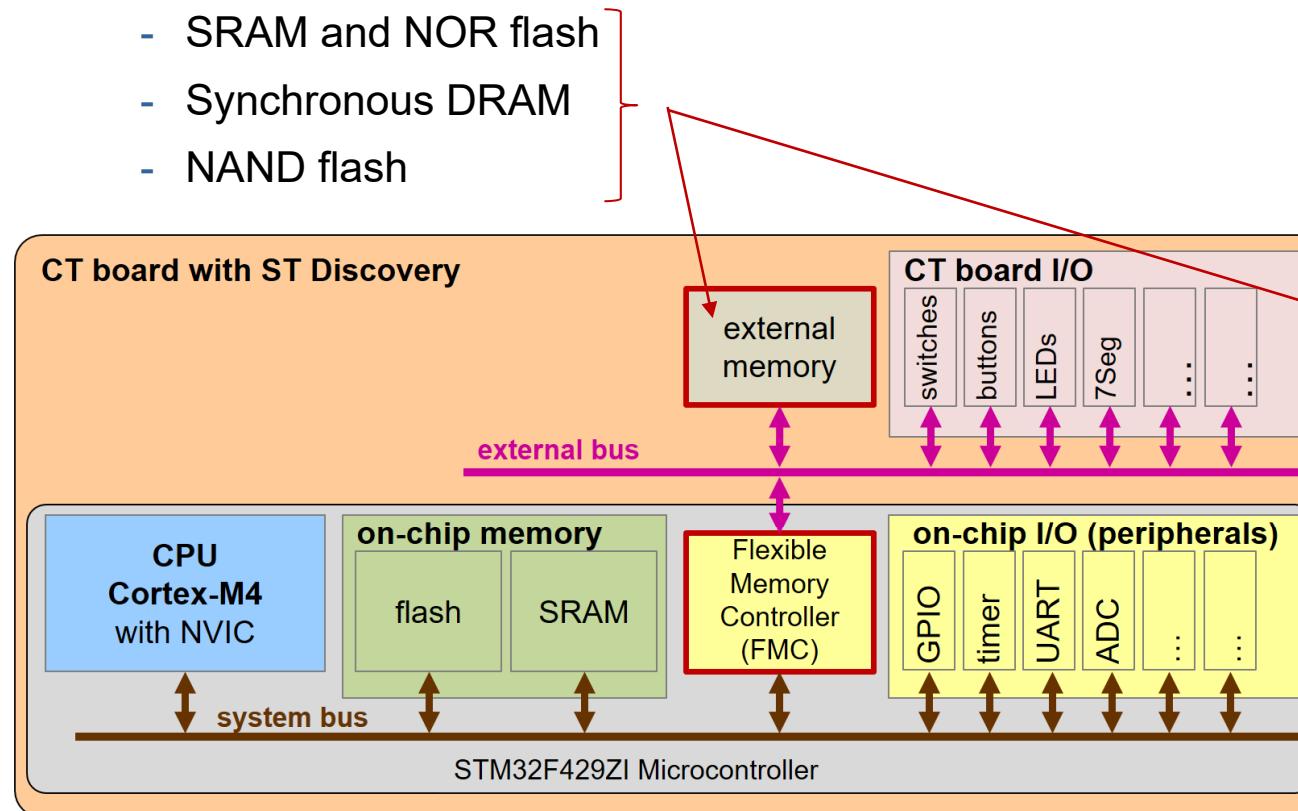
- On-chip system bus 32 data lines, 32 address lines and control signals
- External bus 16 data lines, 26 address lines and control signals



External Memory

■ Extend On-chip Memory

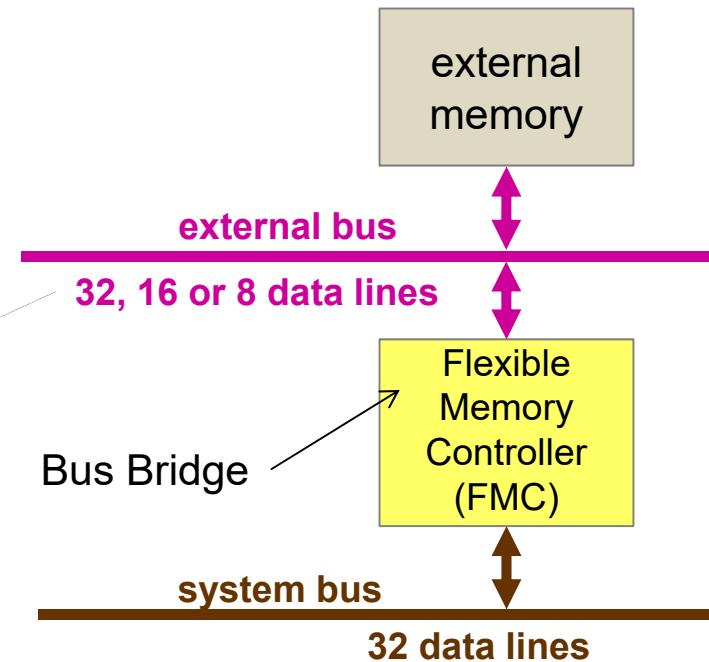
- Flexible Memory Controller (FMC)
 - SRAM and NOR flash
 - Synchronous DRAM
 - NAND flash



■ FMC – Configurable Bus Bridge

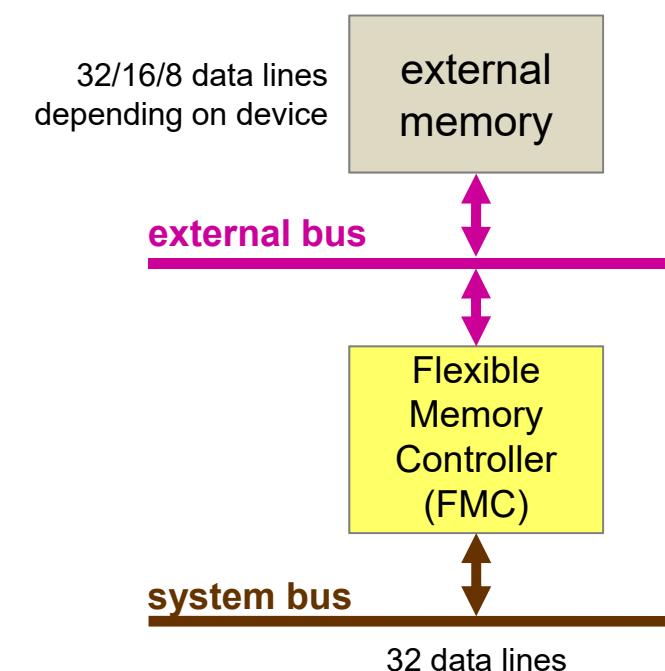
- Bridge between system bus and external bus
 - Slave on system bus
 - Master on external bus
- System bus accesses
 - In address range 0x6000'0000 to 0xDFFF'FFFF
 - Bridged to external bus
 - I.e. FMC initiates a bus cycle on external bus

Number of data lines is a design decision and depends on the external memory device



■ Different Number of Data Lines Causes Bottleneck

- E.g. 32-bit (word) access to 8-bit external memory ¹⁾
 - Single access on system bus
 - Results in 4 accesses on external bus
→ increases access time by factor 4

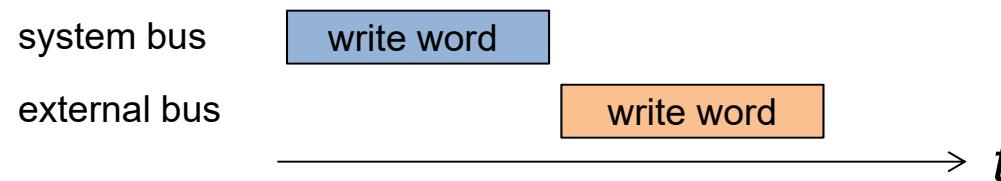


■ Implementation FMC

- CPU write to memory
 - Address and data stored in FMC FIFO buffer
 - Avoids wait for slow memory
 - Free system bus for other accesses
- CPU read from memory
 - System bus has to wait until external memory device provides data

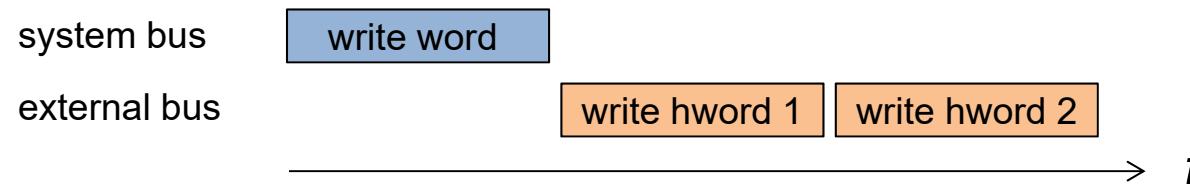
■ Writing a 32-bit Word from System Bus (32 Data Lines)

- to a 32-bit wide external memory (32 data lines)

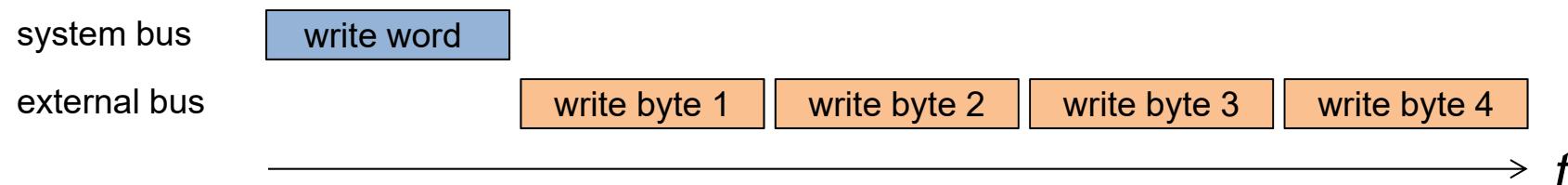


- Word stored in FMC-FIFO
- System bus is released for other accesses
- FMC-FIFO content is transferred to external memory using 1 to 4 bus cycles

- to a 16-bit wide external memory (16 data lines)

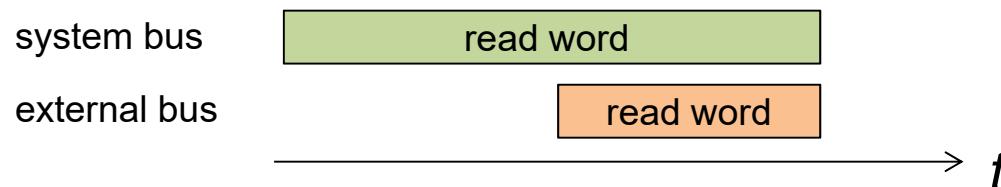


- to an 8-bit wide external memory (8 data lines)

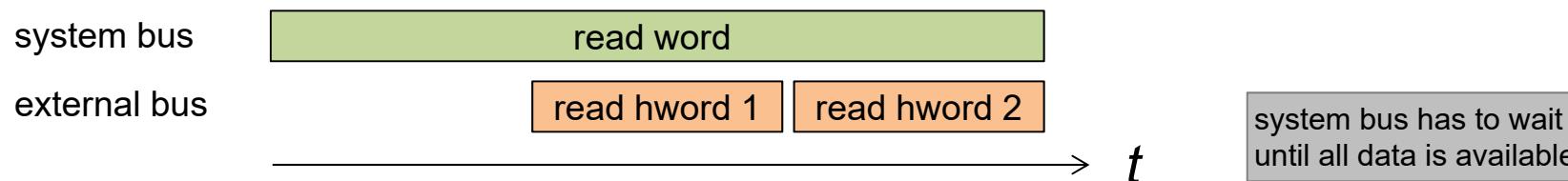


■ Reading a 32-bit Word from

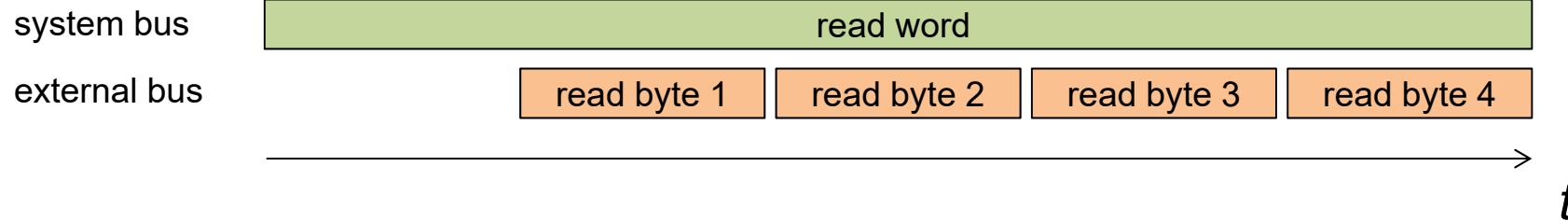
- a 32-bit wide external memory (32 data lines)



- a 16-bit wide external memory (16 data lines)



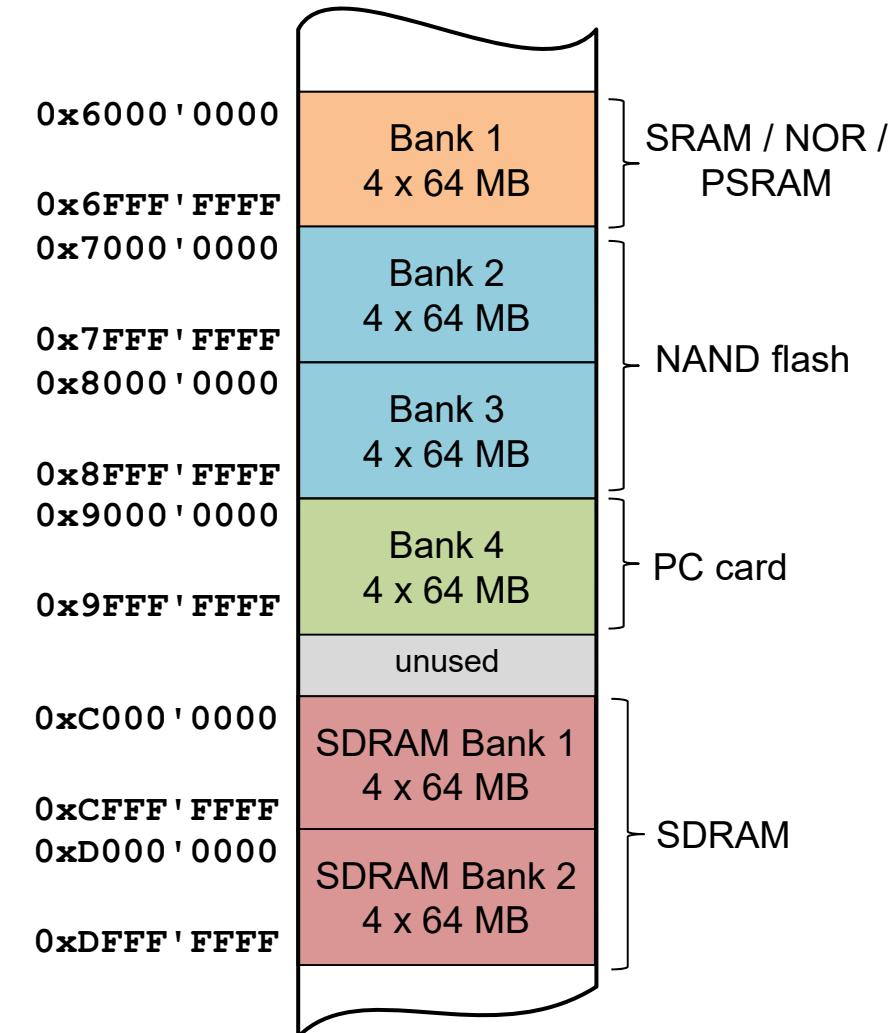
- an 8-bit wide external memory (8 data lines)



■ FMC – Memory Banks¹⁾

- ST defined address ranges for each type of memory
- Organized in 6 banks
- Each bank allows connection of 4 devices
- Pins are multiplexed
 - Not possible to fully use all the banks simultaneously

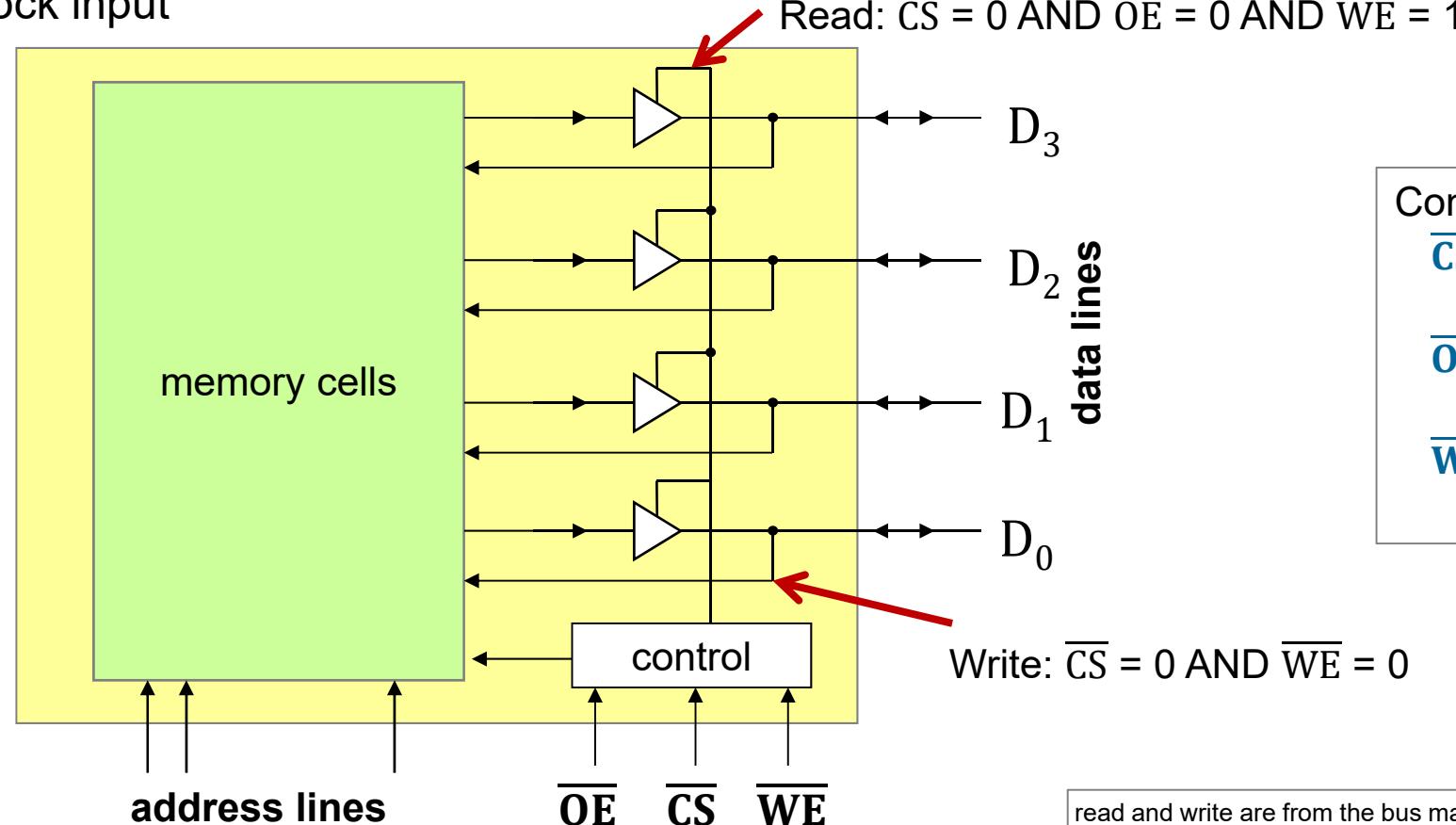
Memory banks and their location in memory



1) An organizational unit of memory. Bank size is architecture dependent

■ Asynchronous SRAM Device

- No clock input



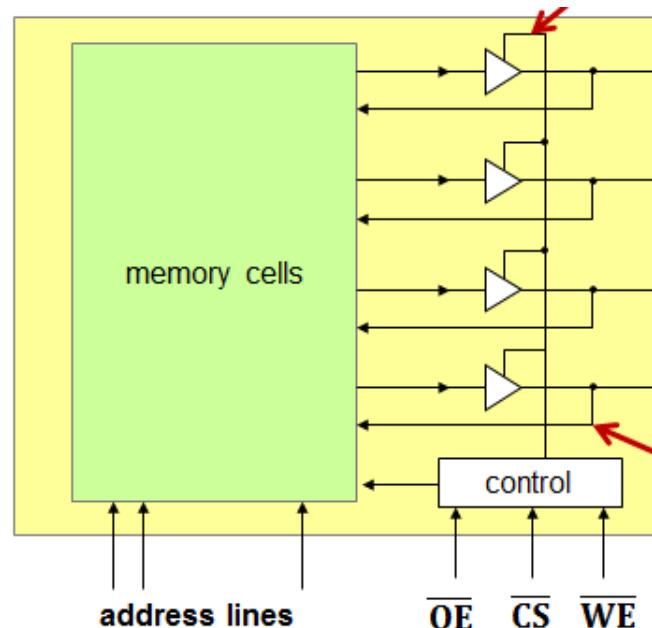
Control signals

\overline{CS}	Chip Select usually active-low
\overline{OE}	Output Enable usually active-low
\overline{WE}	Write enable usually active-low

read and write are from the bus master's perspective

■ Asynchronous SRAM Device

- I.e. the device does not have a clock signal.



Alternatively the control logic can be represented with a truth table

\overline{CS}	\overline{OE}	\overline{WE}	I/O	Function
L	L	H	DATA OUT	Read Data
L	X	L	DATA IN	Write Data
L	H	H	HIGH-Z	Outputs Disabled
H	X	X	HIGH-Z	Deselected

Some memory vendors call the signal \overline{CE} (chip enable) instead of \overline{CS}

■ Exercise: Example Asynchronous SRAM Device

- Use the Data Sheet IDT 71V124SA15TYG
 - Type of memory ?
 - Memory size in bit ?
 - Organization ___K x ___bit ?
 - Number of pins ?
 - Operating voltage ?
 - Which inputs and outputs does the device have ?
 - Truth Table of the control logic ?
 - Access time (read und write) in ns ?
 - Number of accesses per second (in MHz) ?
 - Maximum power [mW] in operating and in full standby ?

■ FMC Signals for SRAMs

- Prefix 'N' → active-low signal

FMC signal name	I/O	Function
A[25:0]	OUT	Address bus
D[31:0]	INOUT	Data bidirectional bus
NE[4:1]	OUT	Four enable lines ¹⁾
NOE	OUT	Output enable
NWE	OUT	Write enable
NBL[3:0]	OUT	Byte enable

see "Synchronous Bus" in slide set "Microcontroller Basics"

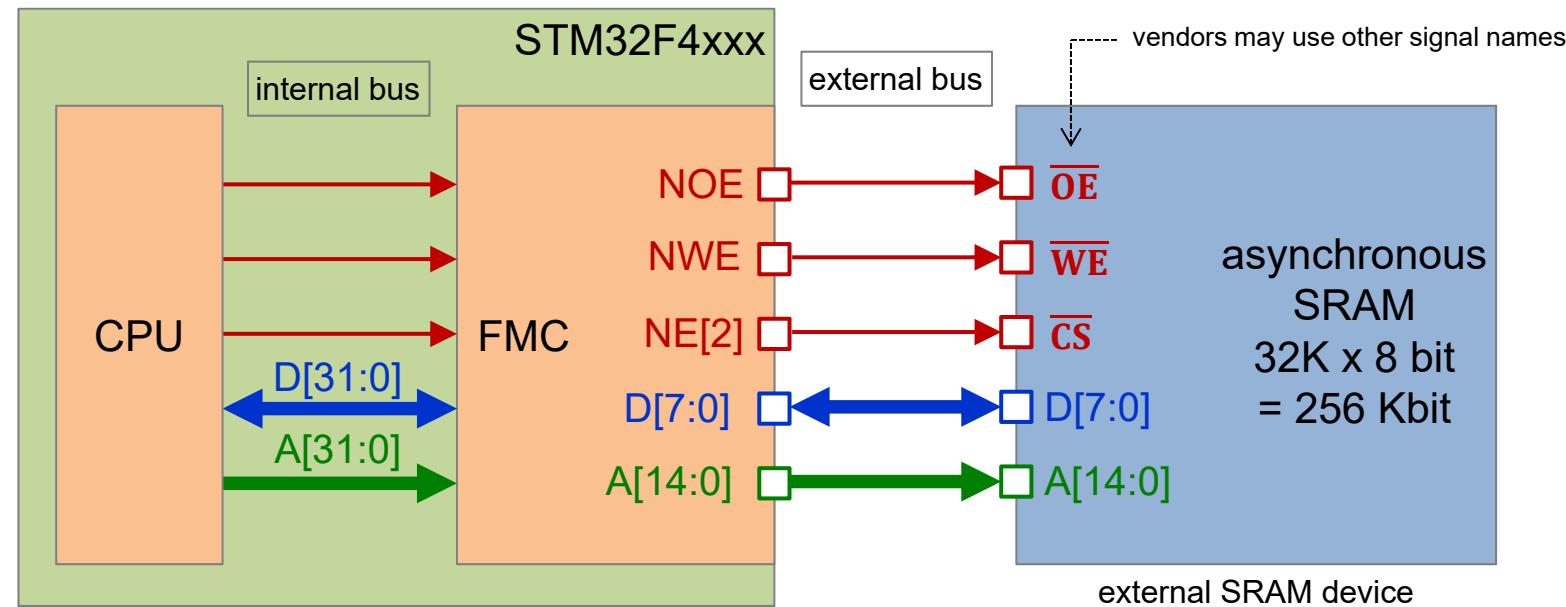
- Write accesses: NBL[3:0] indicate which bytes shall be updated (see lab)

- Example 32-bit data bus D[31:0]

- | | | |
|--------------------|-------------------------|-----------------------|
| ▶ Word access | → all four bytes | NBL[3:0] = 0000b |
| ▶ Half-word access | → two out of four bytes | e.g. NBL[3:0] = 0011b |
| ▶ Byte access | → one out of four bytes | e.g. NBL[3:0] = 1011b |

■ Example 32K x 8 bit SRAM

- Connecting an external 8-bit asynchronous SRAM device

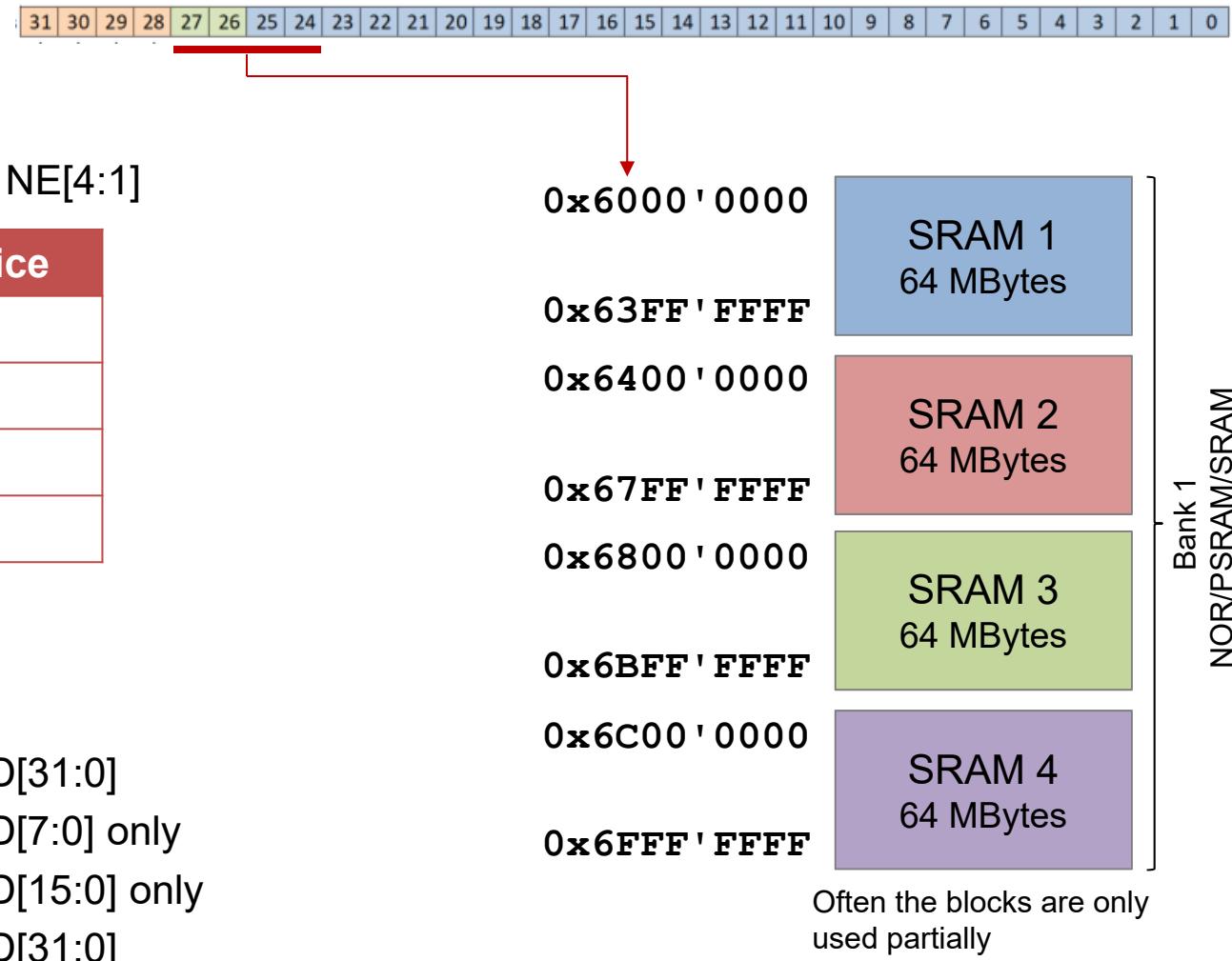


■ FMC – SRAM (Bank 1)

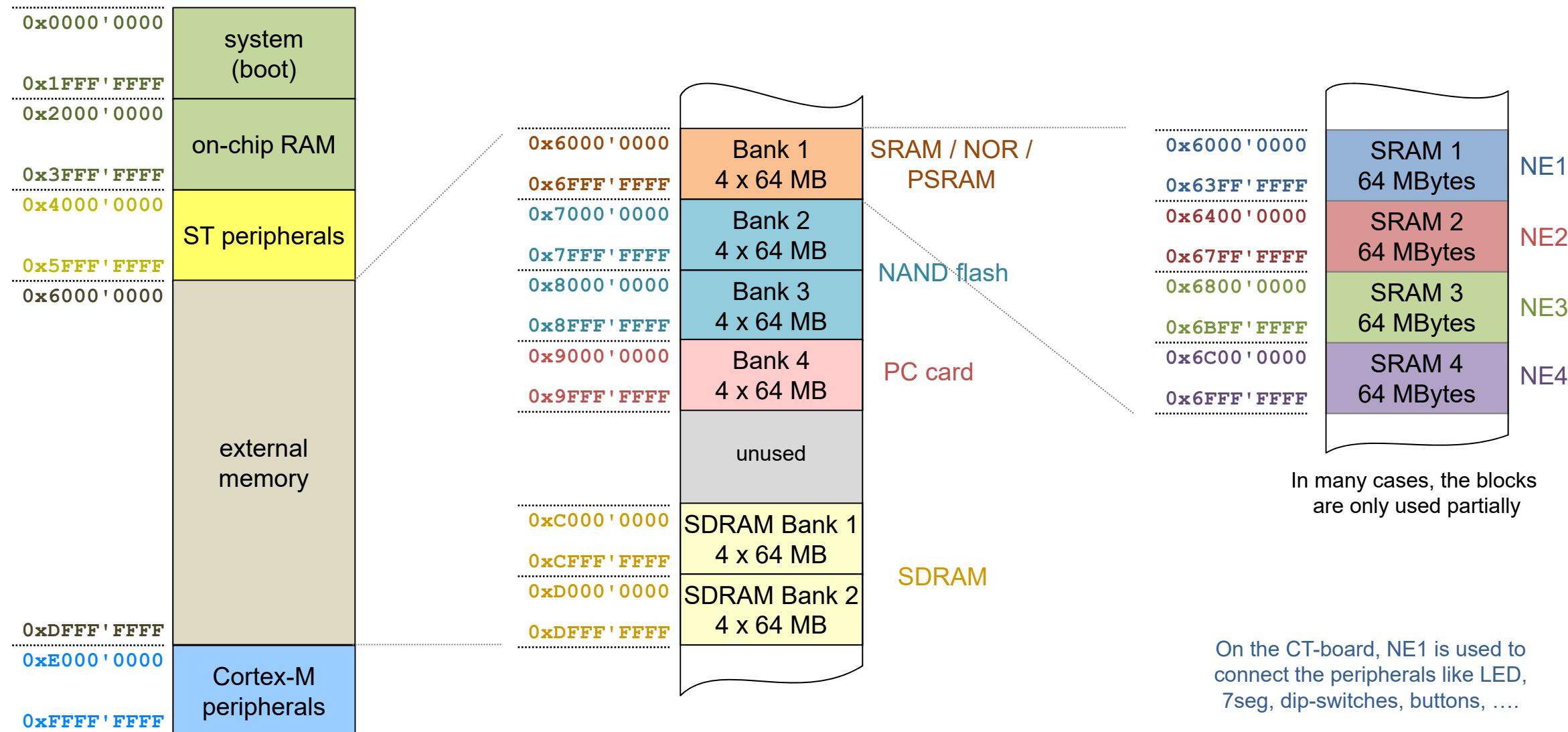
- Select one out of four SRAM devices
 - Address bits 27:26 → Encoded in signals NE[4:1]

A[27:26]	Enable	Memory Device
00	NE[1]	SRAM 1
01	NE[2]	SRAM 2
10	NE[3]	SRAM 3
11	NE[4]	SRAM 4

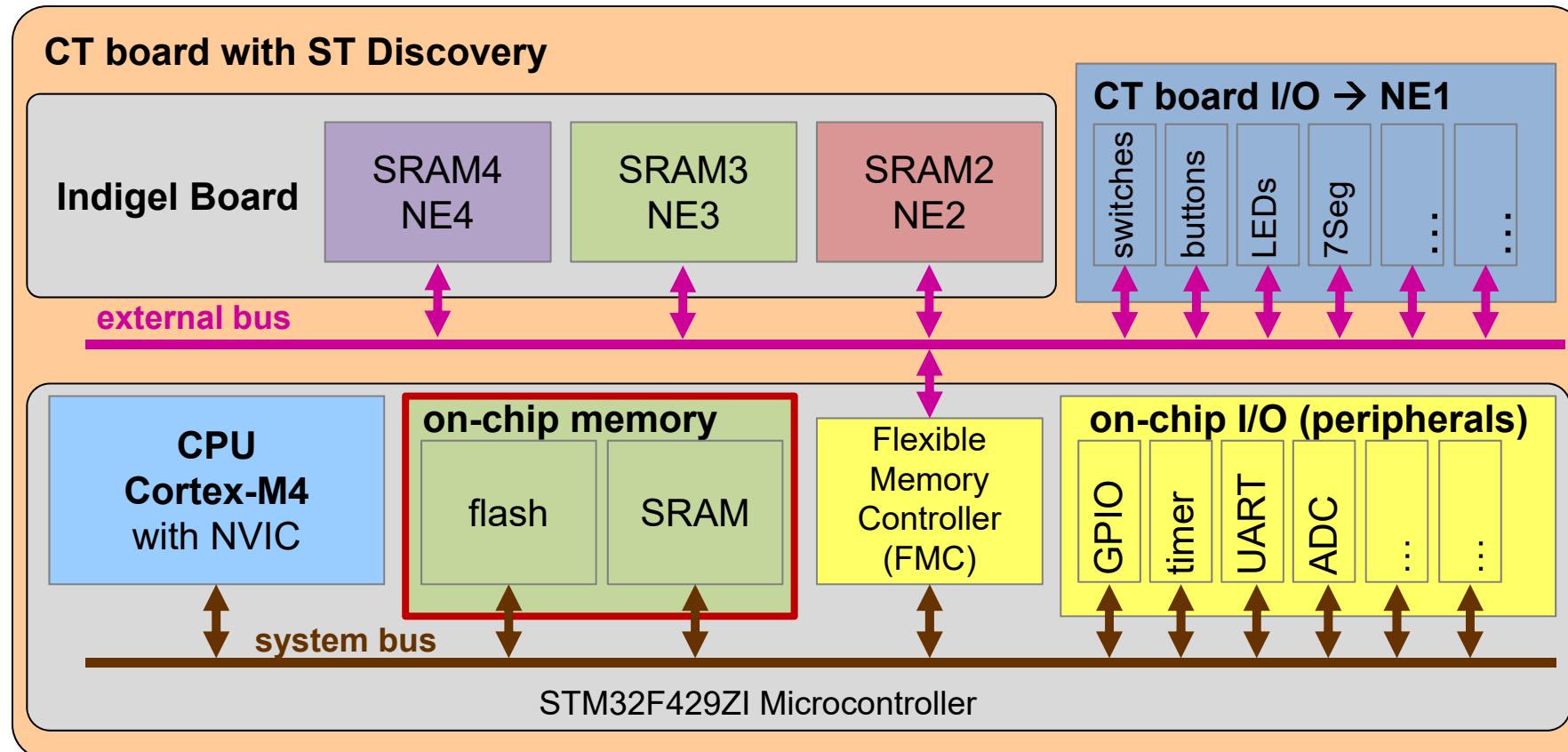
- Data bus configured in control registers
 - Example
 - ▶ SRAM1 as 32-bit → D[31:0]
 - ▶ SRAM2 as 8-bit → D[7:0] only
 - ▶ SRAM3 as 16-bit → D[15:0] only
 - ▶ SRAM4 as 32-bit → D[31:0]



Memory Banks and their Locations in Memory



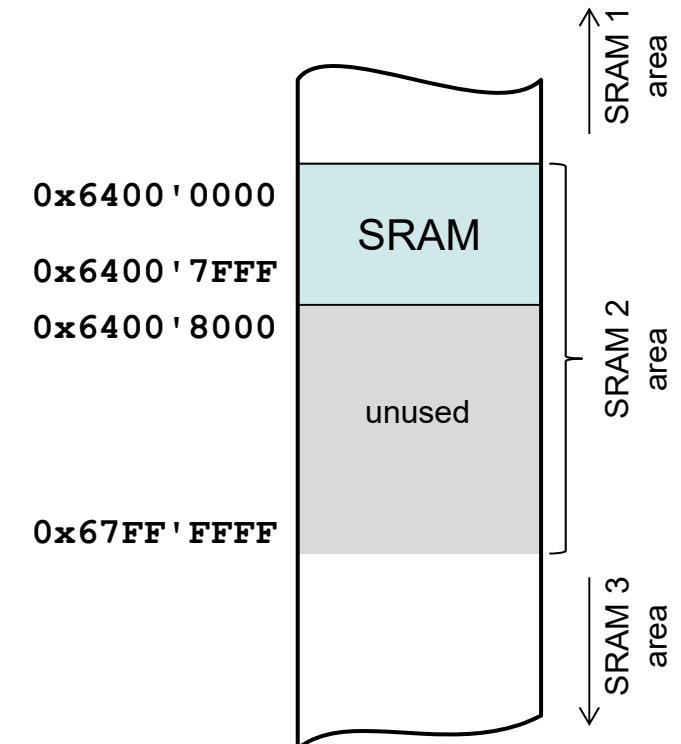
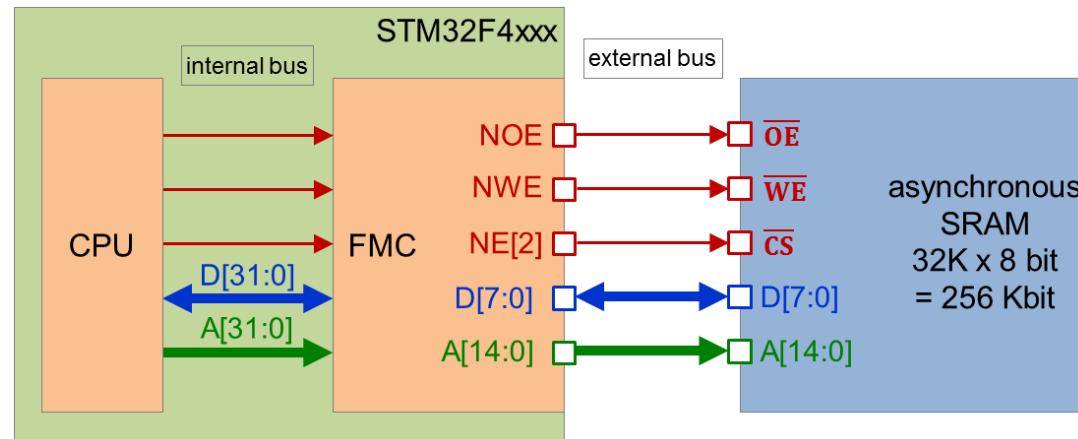
CT System Overview



Asynchronous SRAM

■ Example (revisited)

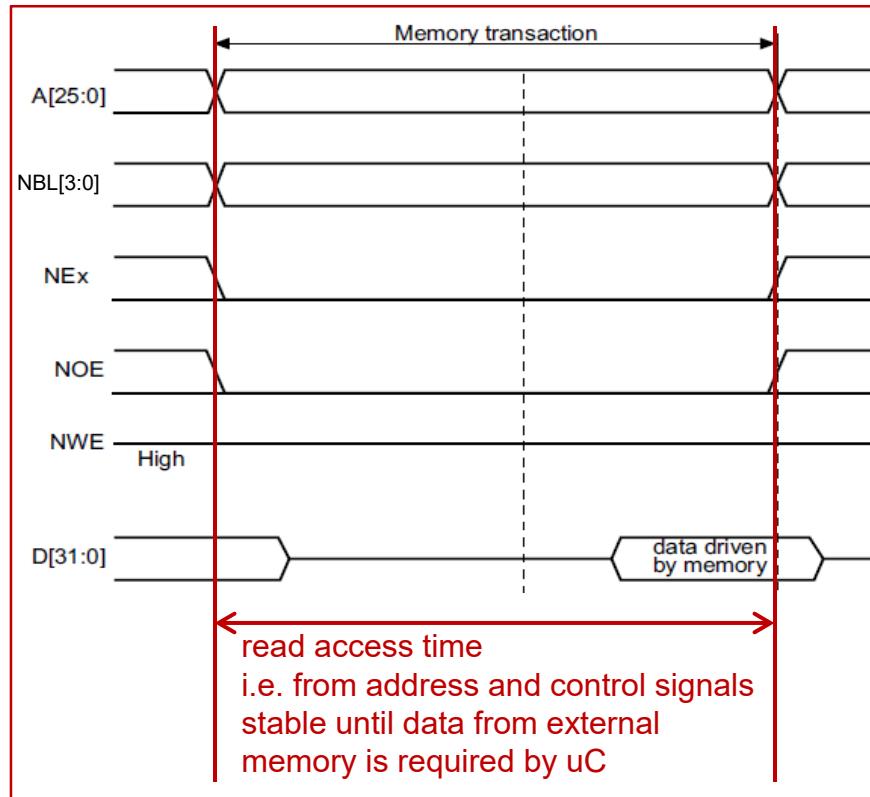
- Memory map of our previous 32K x 8 bit SRAM



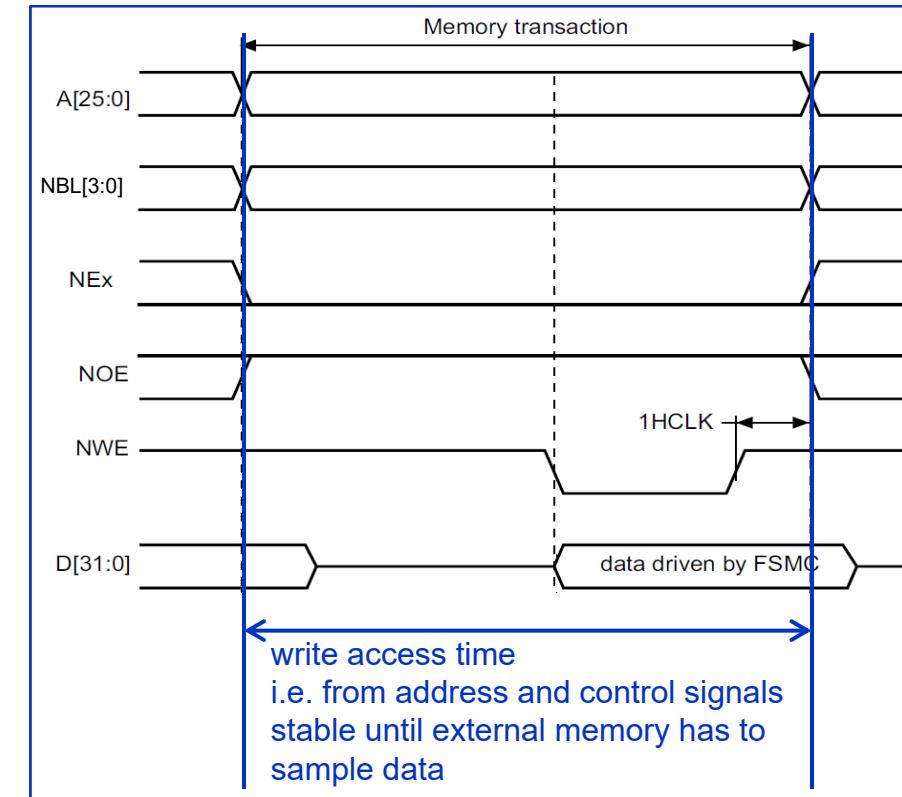
■ Timing on External Bus

- As seen from the microcontroller

Read Access



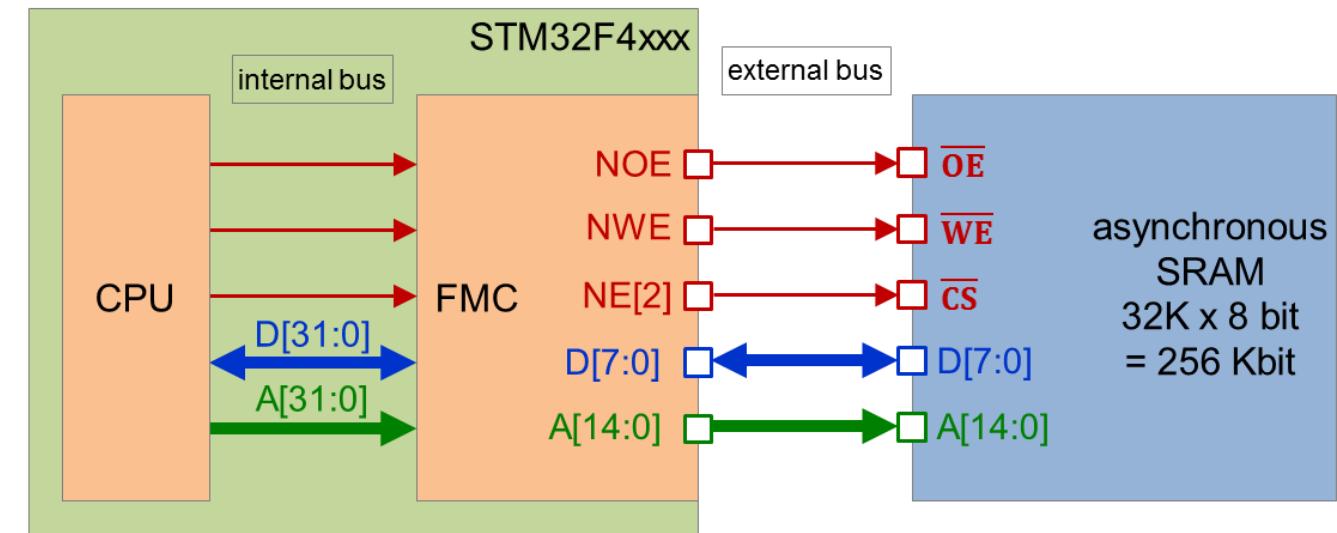
Write Access



Figures from STM32F4xxx reference manual
p. 1591, chapter 37, Flexible Memory Controller

■ Configuration of FMC

- Location of FMC control registers
 - 0xA000'0000 – 0xA000'0FFF
- Configure FMC according to SRAM datasheet
 - Data bus size → 8-bit, 16-bit, 32-bit
 - Access times
 - and others



The FMC Registers allow configuration for many different memory types. However we only cover a few selected parameters for asynchronous SRAM.

Asynchronous SRAM

■ Configuring the FMC for SRAM

Table 291. FMC register map

Control register
for SRAM1 →
@ 0xA000'0000

Control register
for SRAM2 →
@ 0xA000'0008

Control register
for SRAM3 →
@ 0xA000'0010

Timing register
for SRAM1 →
@ 0xA000'0004

00	8 bits
01	16 bits
10	32 bits
11	reserved

Use between 1 and 255 HCLK cycles during data phase.

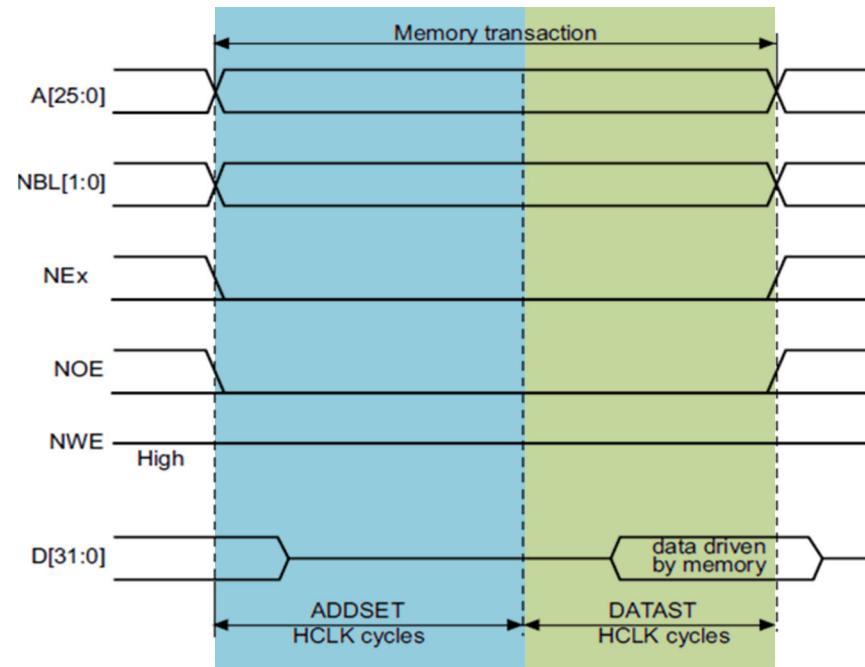
Use between 1 and
15 HCLK cycles during
address phase

■ ADDSET and DATAST

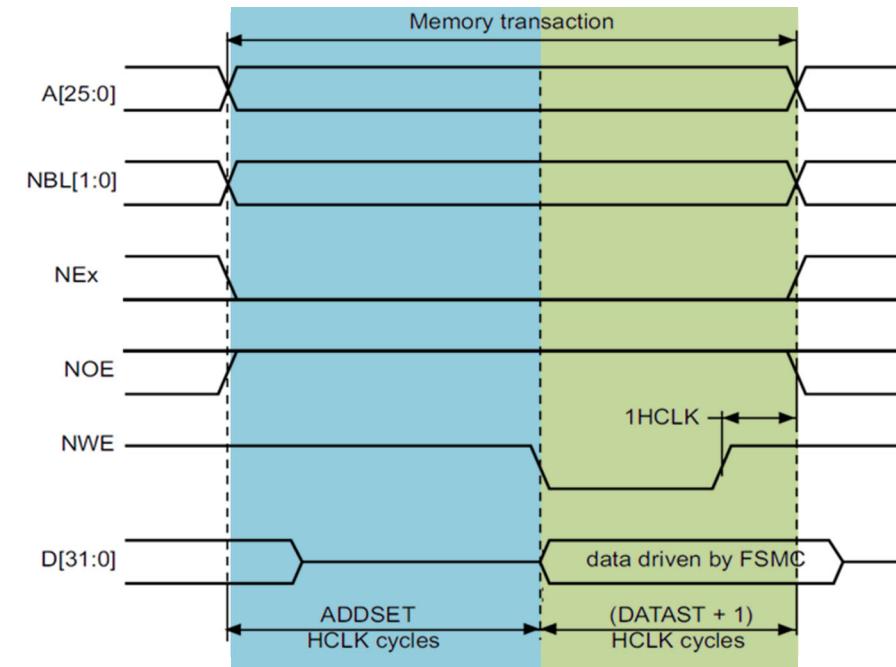
→ Adapt STM32F4 to the speed of the memory

- Configuring length of access cycles
- HCLK programmed to 84 MHz during start-up of CT-Board
 - HCLK = Frequency of CPU and internal bus

Read Access



Write Access



Conclusions

Semiconductor Memories

Non-volatile

Holds data even if power is turned off.

PROM

Programmable Read Only Memory

- Programmed through fuses/masks
- Factory or one time user programmed
- Irreversible programming

EEPROM

Electrically Erasable PROM

- Floating gate technology
- Random read and write
- Low density → expensive

Flash

Block-wise EEPROM

- High density
- Medium read latency
- Sectors for erasing
- NOR: random read access → allows direct code execution
- NAND: High density, block-wise access SD-cards, SSD

NV - RAM

non-volatile RAM

Volatile

Looses data when power is turned off.

SRAM

Static Random Access Memory

- Flip-flop based structure
- Static: No refresh required
- Each access requires the same amount of time

SDRAM

Synchronous Dynamic Random Access Memory

- Capacitor-based
- Refresh
- High density
- Synchronous interface
- Latency
- Block-wise transfers

■ Flexible Memory Controller STM32

- Configurable bridge to connect external memories → e.g. asynchronous SRAM, NOR flashes, etc.

For Information Only

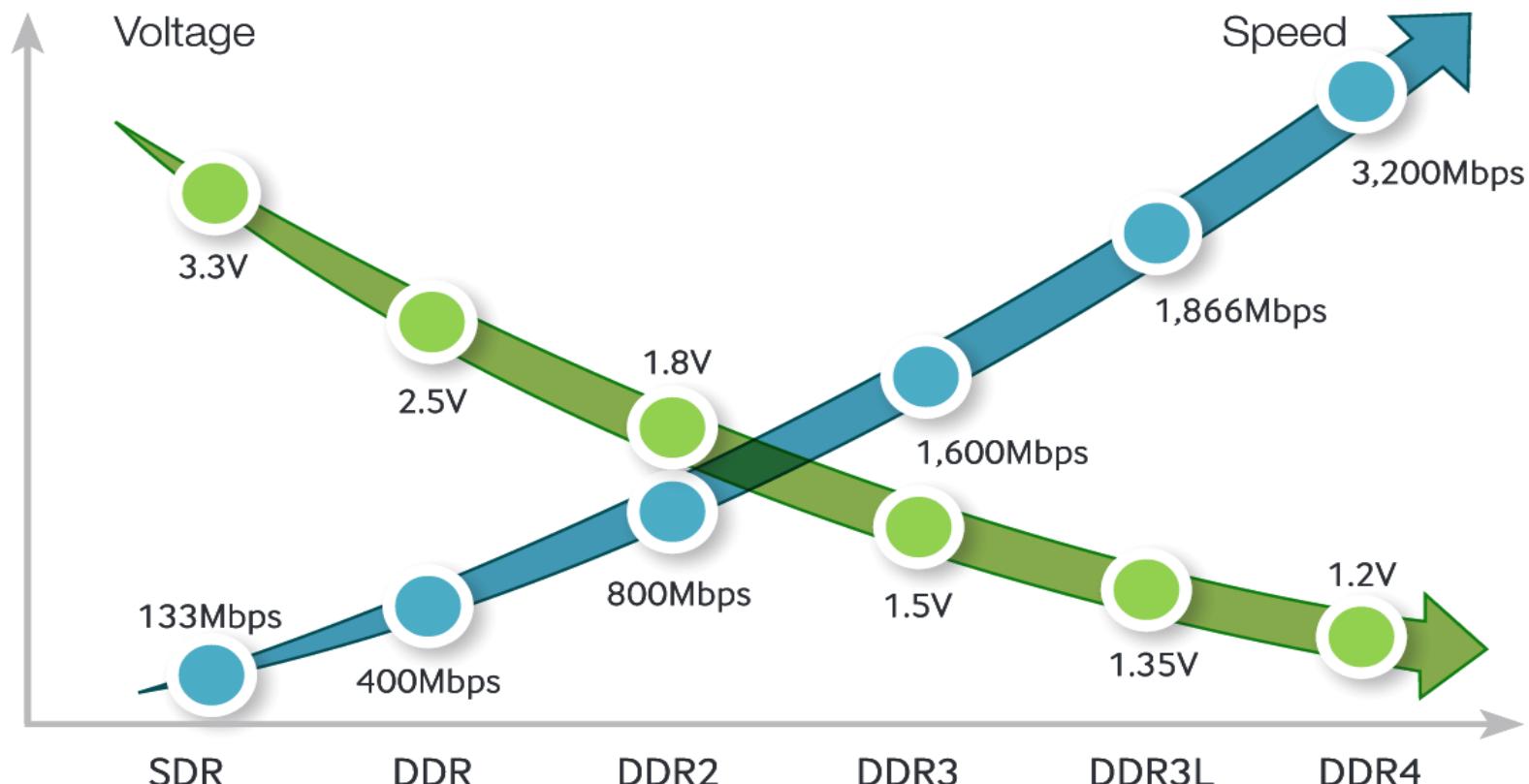
TRENDS AND FIGURES

SDRAM – Synchronous Dynamic RAM

■ Voltage and Speed

- Different generations of SDRAM

SDR	Single Data Rate
DDR	Dual Data Rate uses rising and falling clock edge

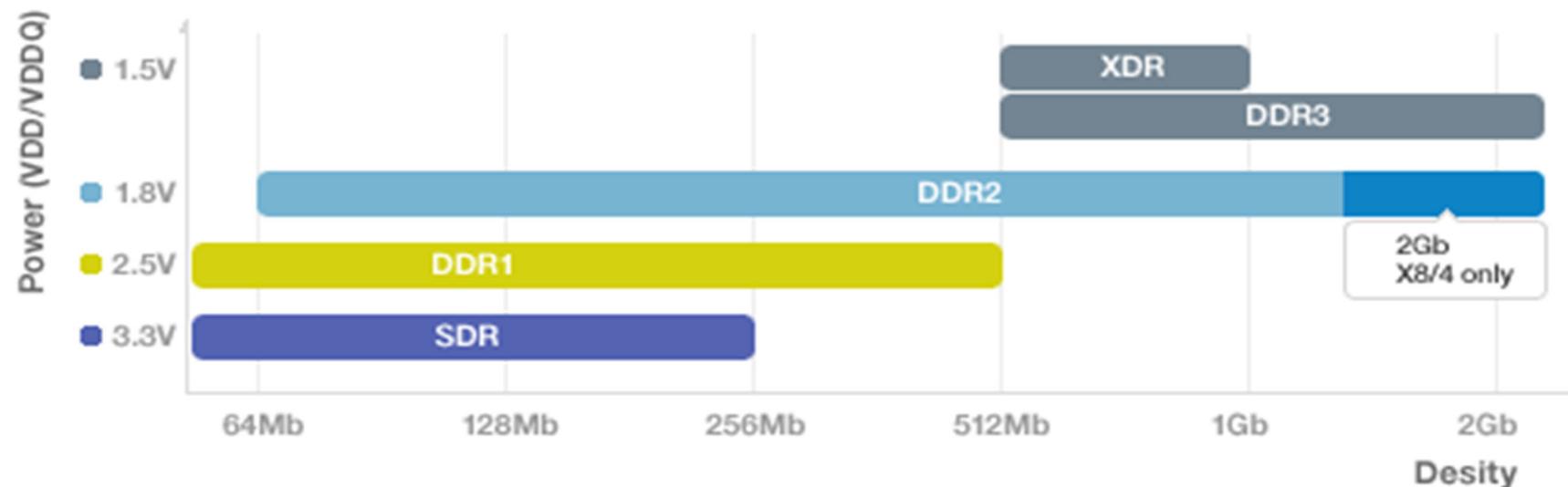


source: Samsung

■ Densities

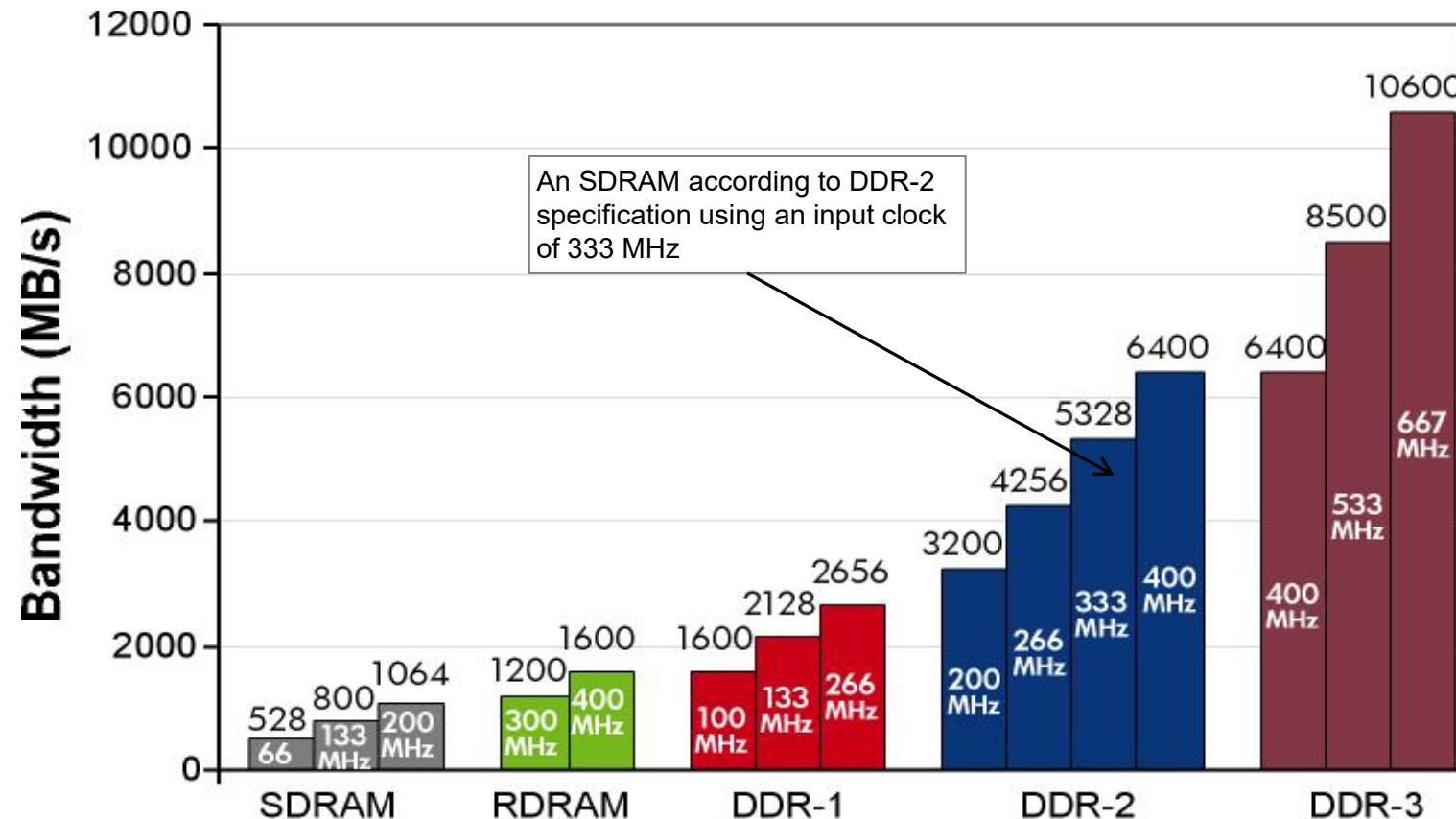
DRAM Density & Power Supply for Consumer Applications in 2009~2010

SAMSUNG supports 1.5V, thus meeting customer demand in a wide range of applications.



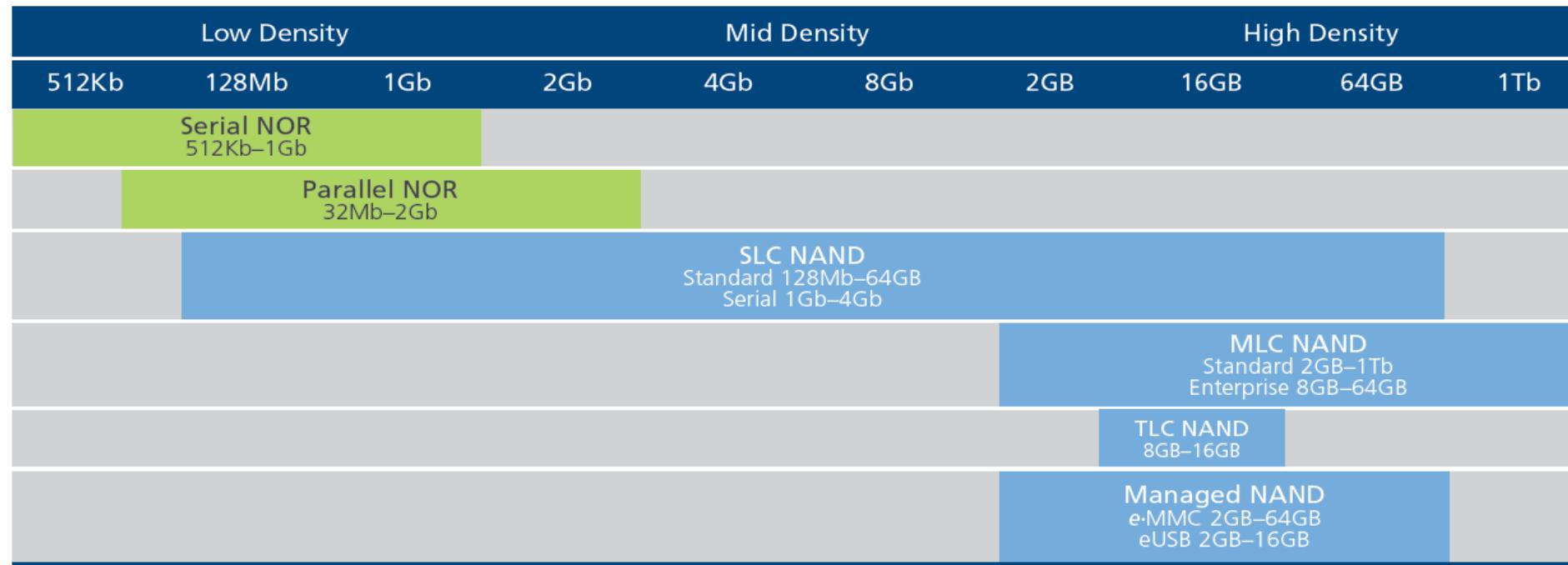
SDRAM – Synchronous Dynamic RAM

■ Peak Bandwidth



Source: HP: Memory technology evolution: an overview of system memory technologies

■ Flash Densities



Source: Micron Technology, Inc., 2014

Managed NAND includes a controller for tasks like error correction.

Cache

Computer Engineering 2

- **Principle of locality – The memory hierarchy**
- **Cache mechanics**
- **Cache organization**
 - Fully associative
 - Direct mapped
 - N-way set associative
- **Performance**
- **Replacement and write strategies**
- **The programmer's perspective**

■ Situation

- Processor
 - Fast cycle time
- Fast DRAM¹
 - Slow cycle time (up to 100x)
 - Efficiently reads only in bursts
- → Bridging the gap such that pipelining is effective!

■ Goal

- Access “slower” memory in bursts and maintain a fast cache for fast single accesses
- But: Data consistency must be carefully managed, such that both, cache and main memory have the same data

¹ Dynamic RAM

- **At the end of this lesson you will be able**

- to understand the principles of cache memory
- to explain the principle of locality
- to enumerate advantages and disadvantages of different cache models
 - Fully associative
 - Direct mapped
 - N-way set associative
- to enumerate types of cache misses
- to understand how cache size and cache hit rate are related
- to name different replacement strategies

■ Principle of locality

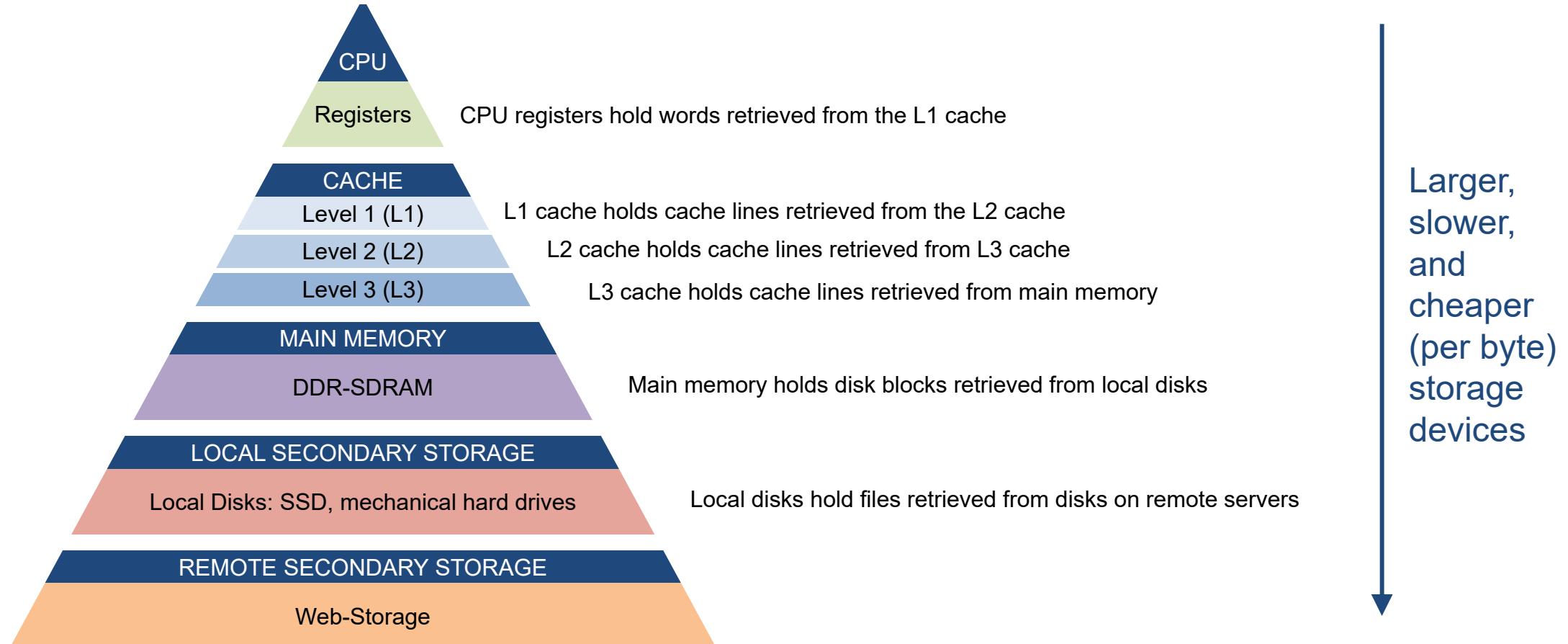
Programs usually access small regions of memory in a given interval of time

- **Spatial locality** → Current data location is likely being close to next accessed location
- **Temporal locality** → Current data location is likely being accessed again in near future

```
for(i = 0; i < 100000; i++) {      // incremental access
    a[i] = b[i];
}

if (a[1234] == a[4321]) {          // → temporal locality
    a[1234] = 0;
}
```

The Memory Hierarchy

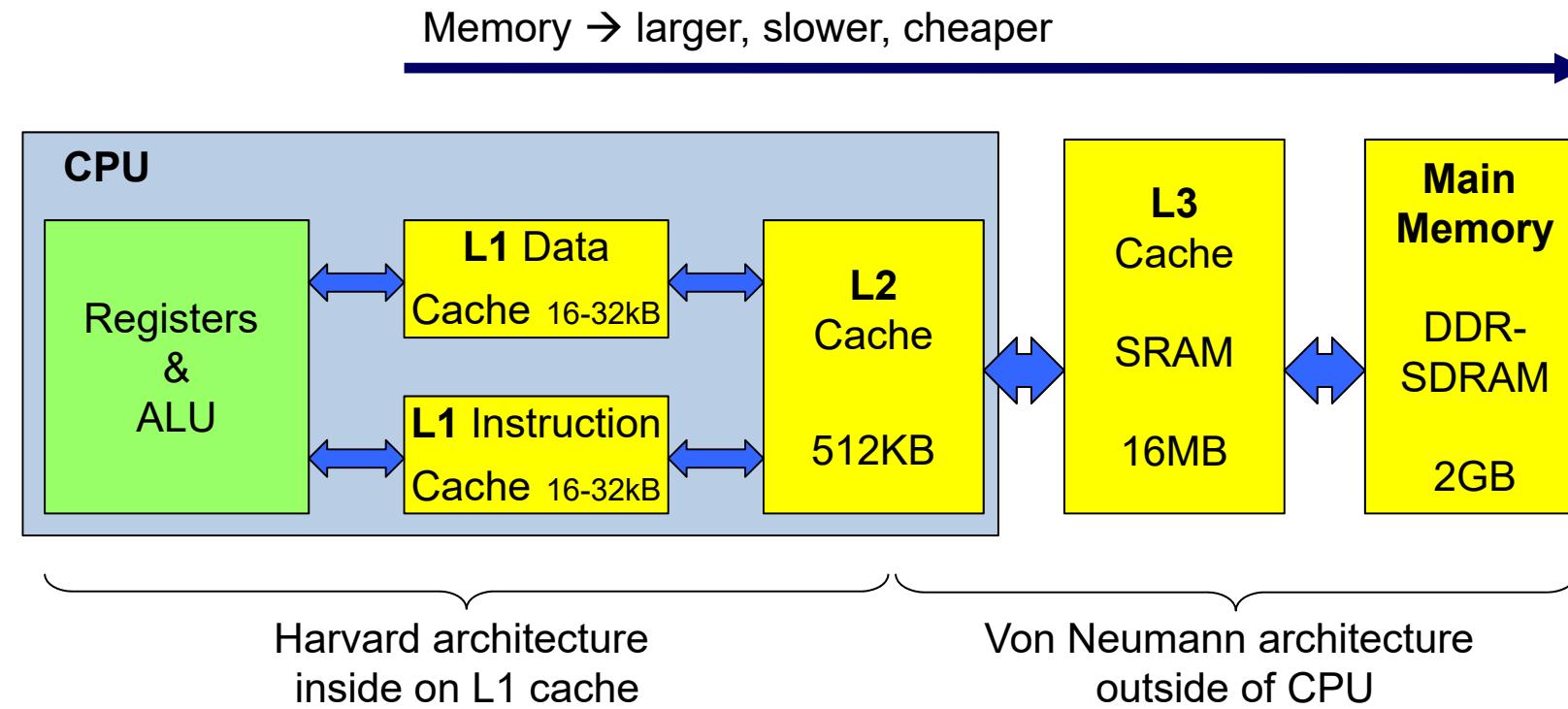


Adapted from Bryant and O'Hallaron

The Memory Hierarchy

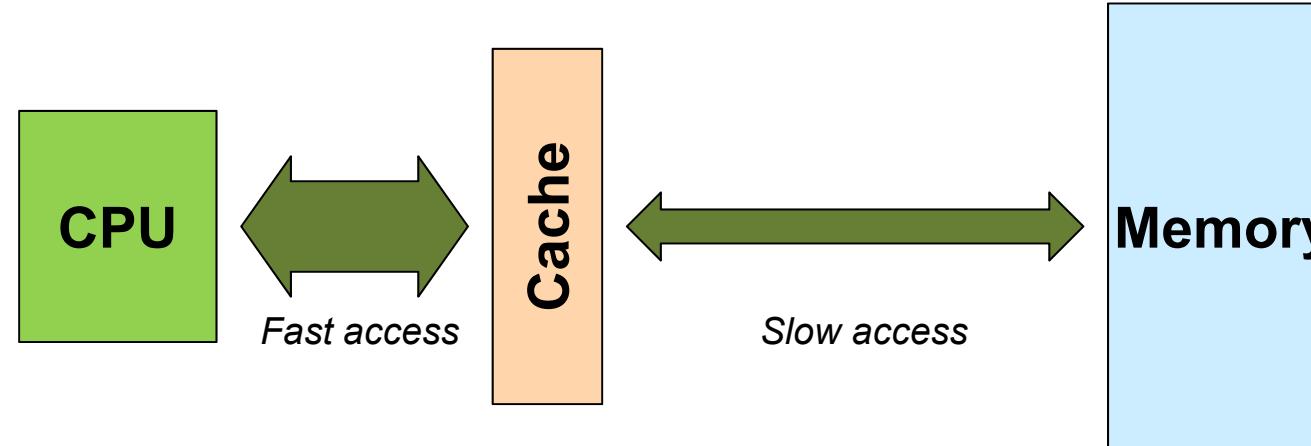
■ Cache Levels

- Typical Cache Architecture



■ Definition cache

- Computer memory with short access time
- Storage of frequently or recently used instructions or data



CPU

Very small
Registers (Word)

Cache

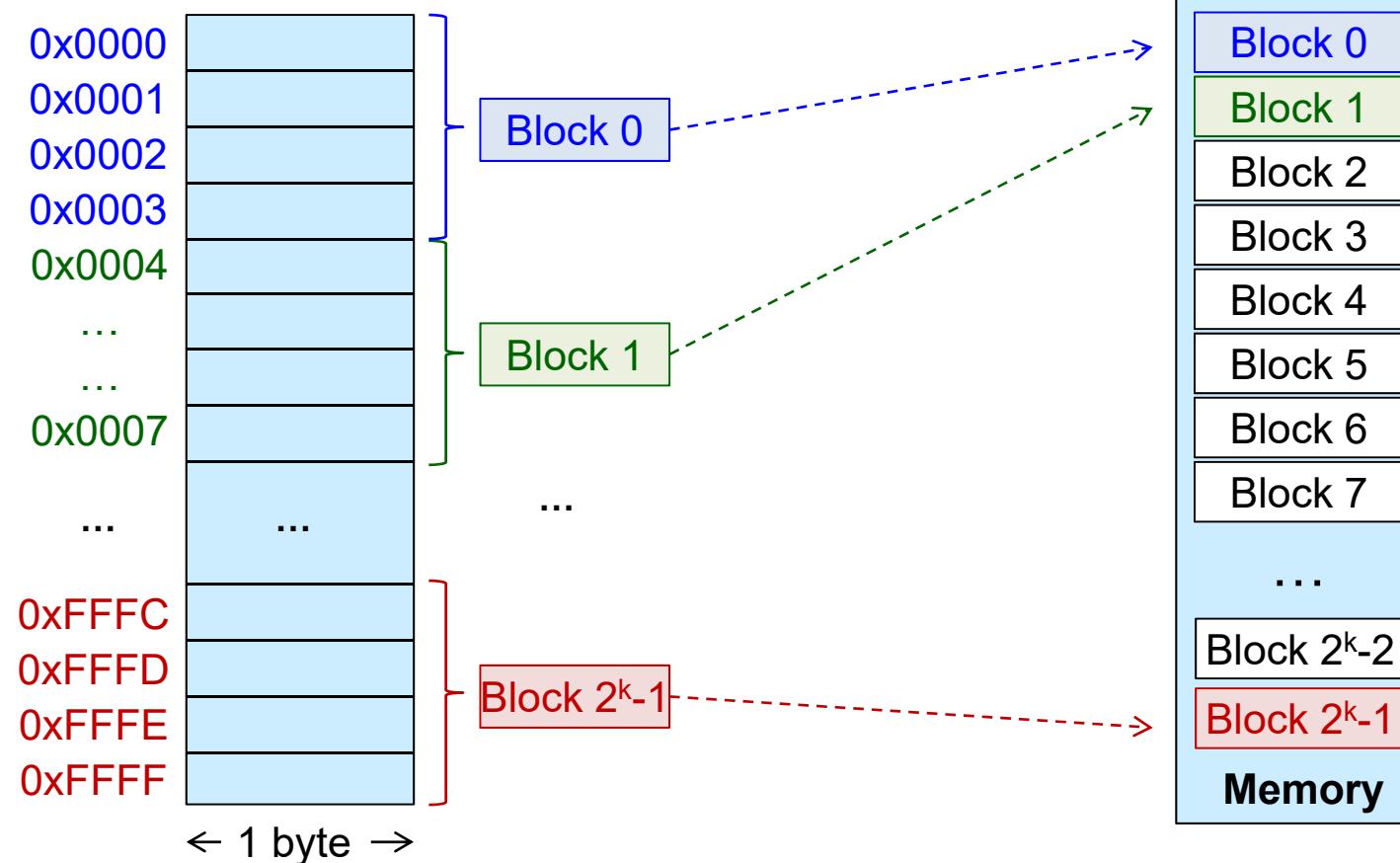
Small, fast, expensive
Caches a subset of
memory blocks

Memory

Large, slow, cheap
Partitioned into memory blocks

■ Memory blocks

- Address range is partitioned into memory blocks



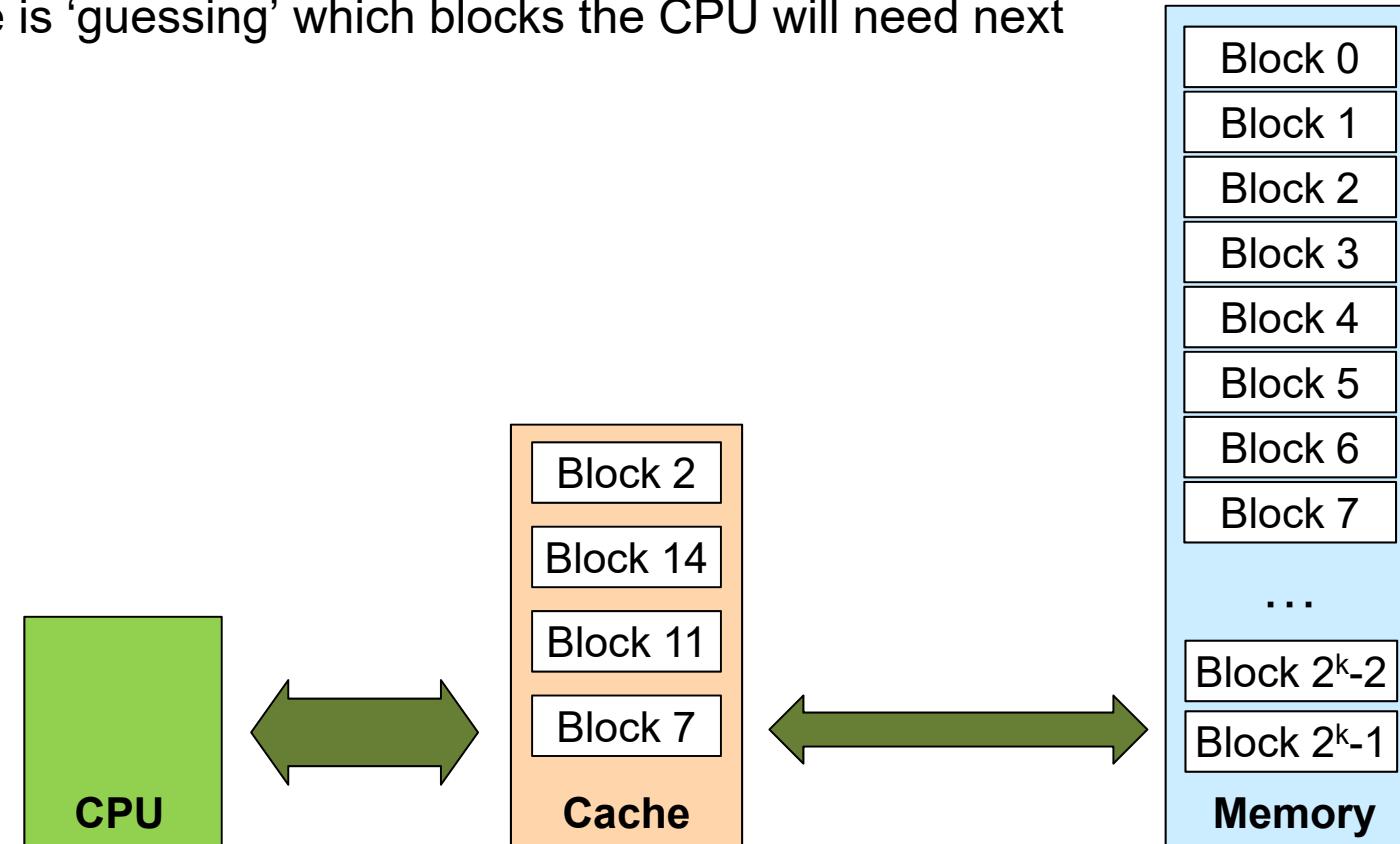
All examples given in this lecture are using

- *hypothetical 16 bit wide addresses*
- *blocks of 4 Bytes*

The number of Bytes per block is a design decision

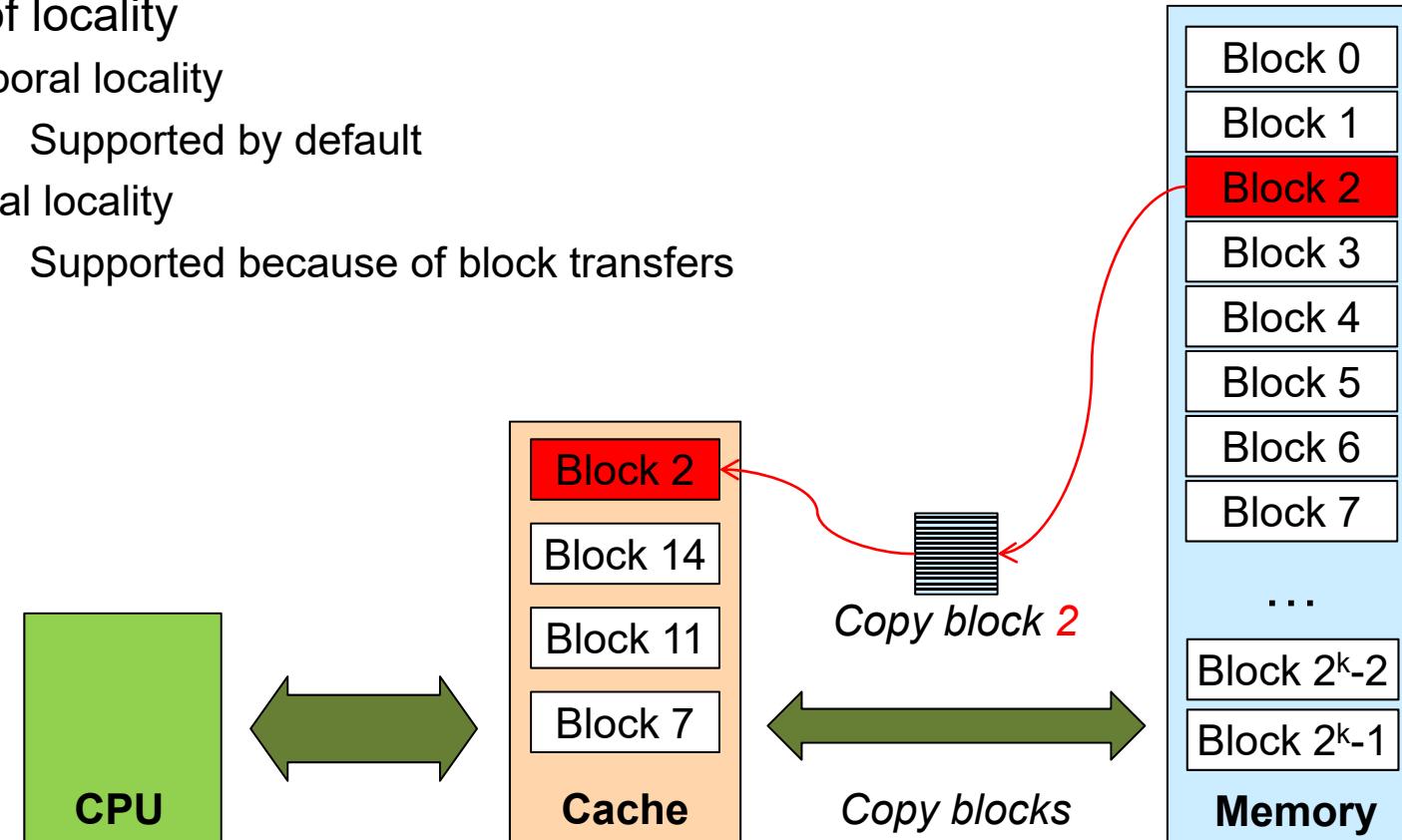
■ Memory blocks

- Selected blocks of main memory copied to faster cache memory
- The cache is ‘guessing’ which blocks the CPU will need next



■ Memory blocks

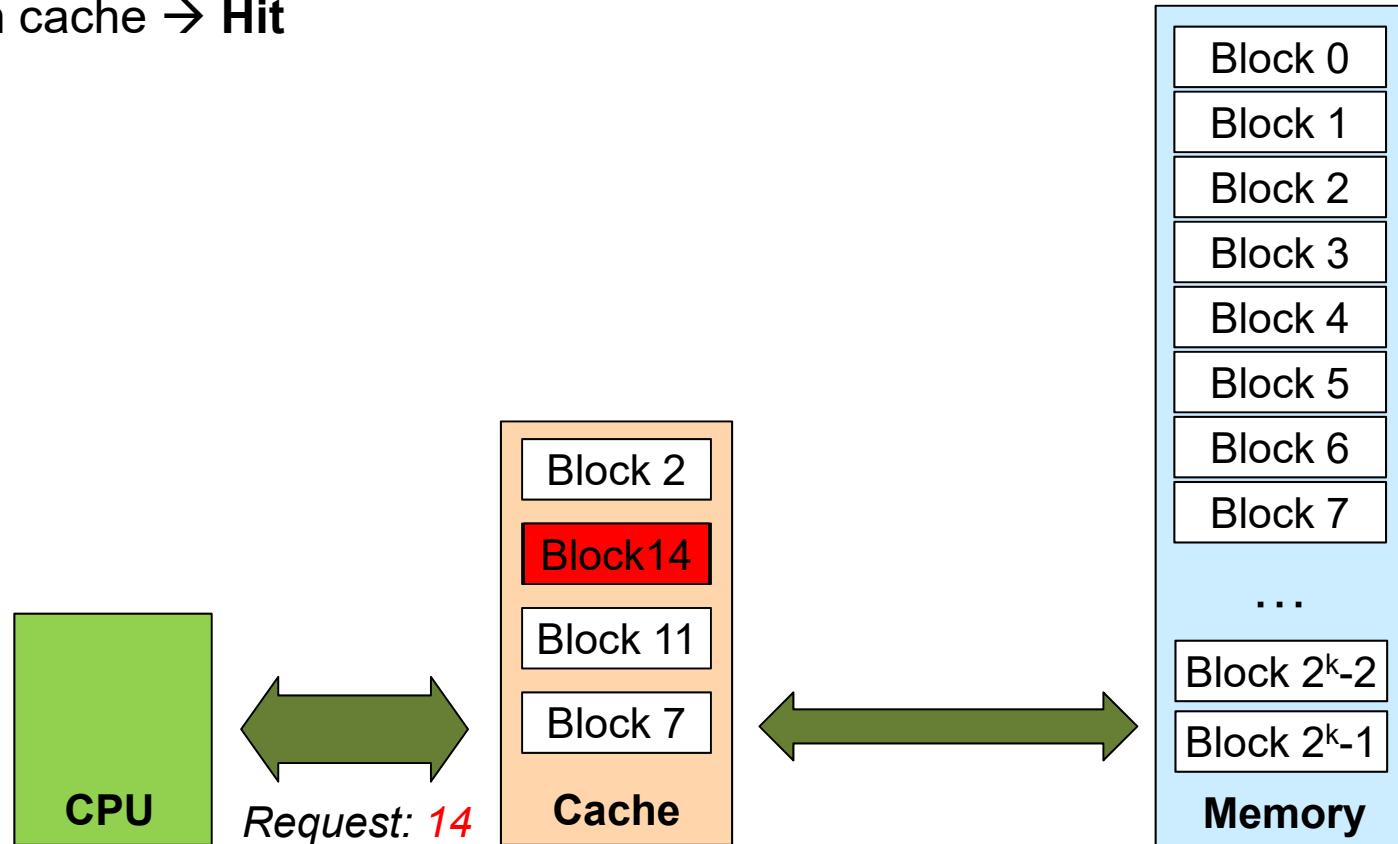
- Blocks of main memory copied to faster cache memory
- Principle of locality
 - Temporal locality
 - ▶ Supported by default
 - Spatial locality
 - ▶ Supported because of block transfers



Cache Mechanics

■ Cache hit

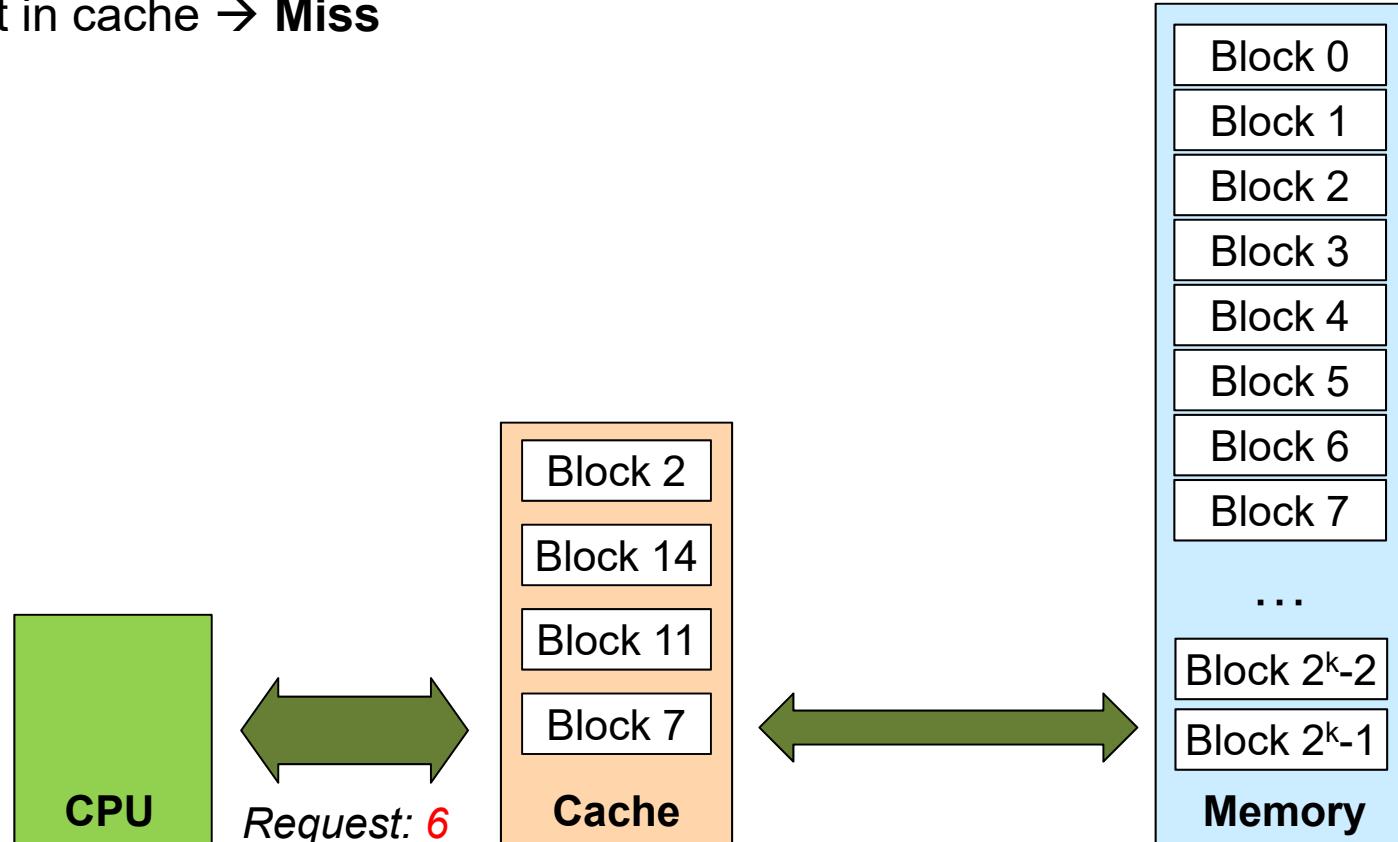
- Data in block 14 is needed
- Block 14 in cache → Hit



Cache Mechanics

■ Cache miss (I)

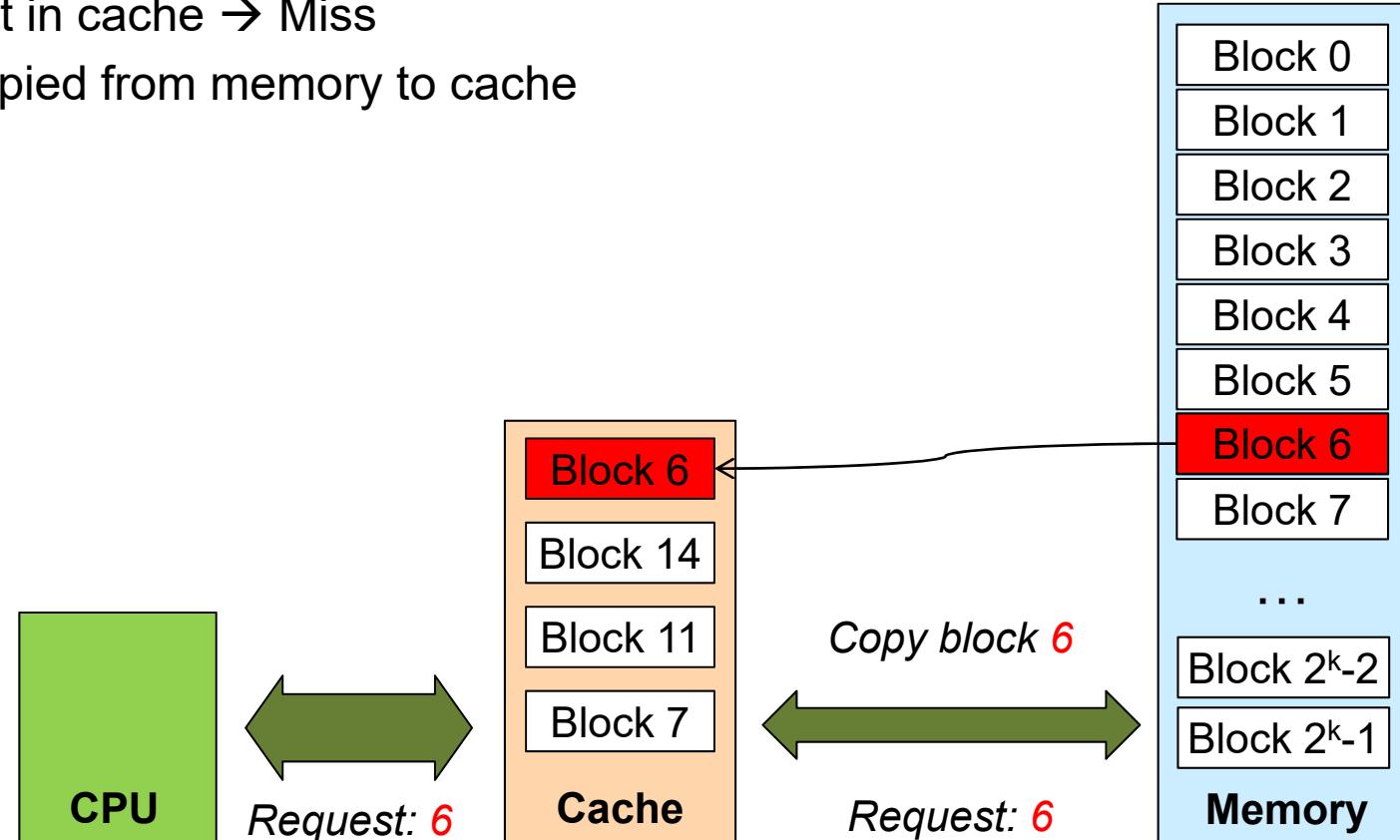
- Data in block 6 is needed
- Block 6 not in cache → **Miss**



Cache Mechanics

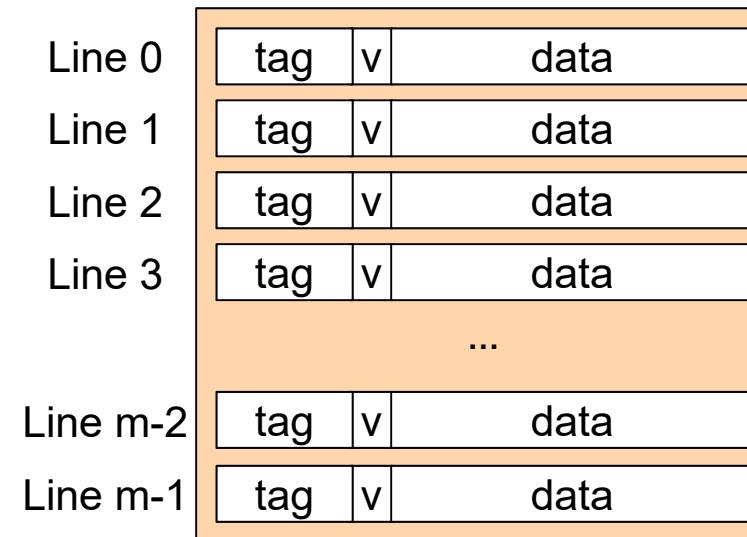
■ Cache miss (II)

- Data in block 6 is needed
- Block 6 not in cache → Miss
- Block 6 copied from memory to cache



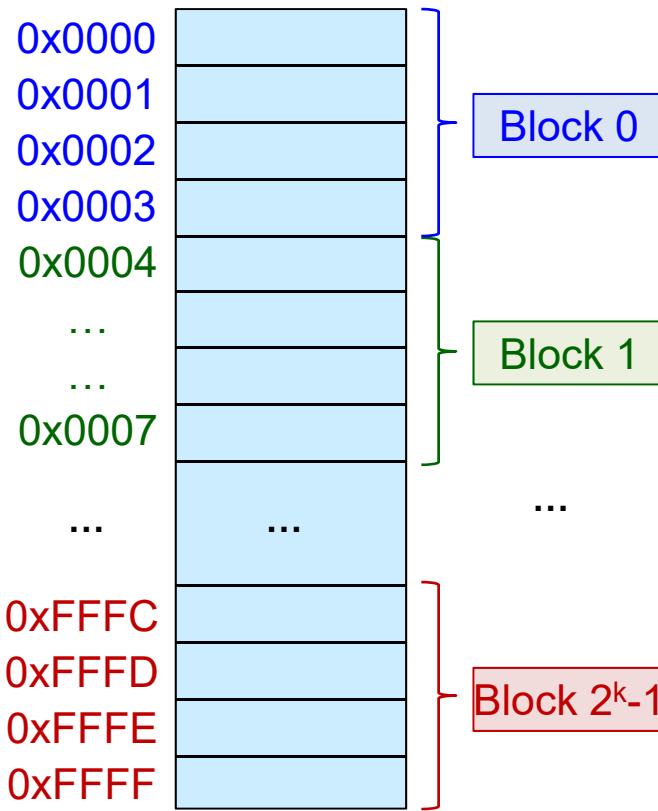
■ Organized in lines

- Valid bit v → indicates that line contains valid data
- Tag → unique identifier for memory location
- Data → data of exactly one memory block
- m = overall number of cache lines



Cache Organization

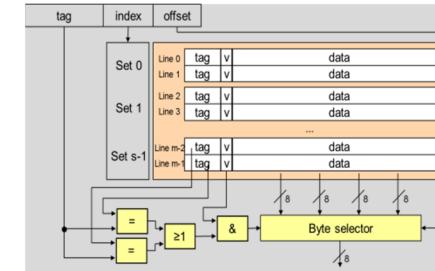
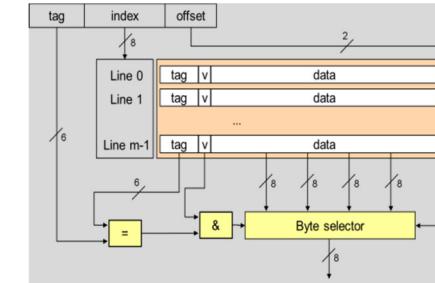
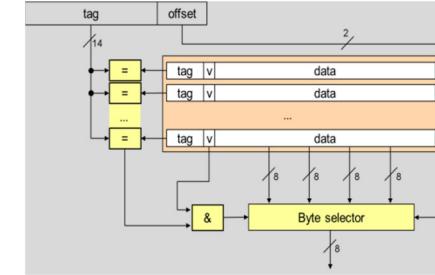
■ Addressing



Block	Address	Block identification bits	offset
Block 0	0x0000	0000 0000 0000 00	00
	0x0001	0000 0000 0000 00	01
	0x0002	0000 0000 0000 00	10
	0x0003	0000 0000 0000 00	11
Block 1	0x0004	0000 0000 0000 01	00
	0x0005	0000 0000 0000 01	01
	0x0006	0000 0000 0000 01	10
	0x0007	0000 0000 0000 01	11
...			
Block 2 ^k -1	0xFFFFC	1111 1111 1111 11	00
	0xFFFFD	1111 1111 1111 11	01
	0xFFFFE	1111 1111 1111 11	10
	0xFFFFF	1111 1111 1111 11	11

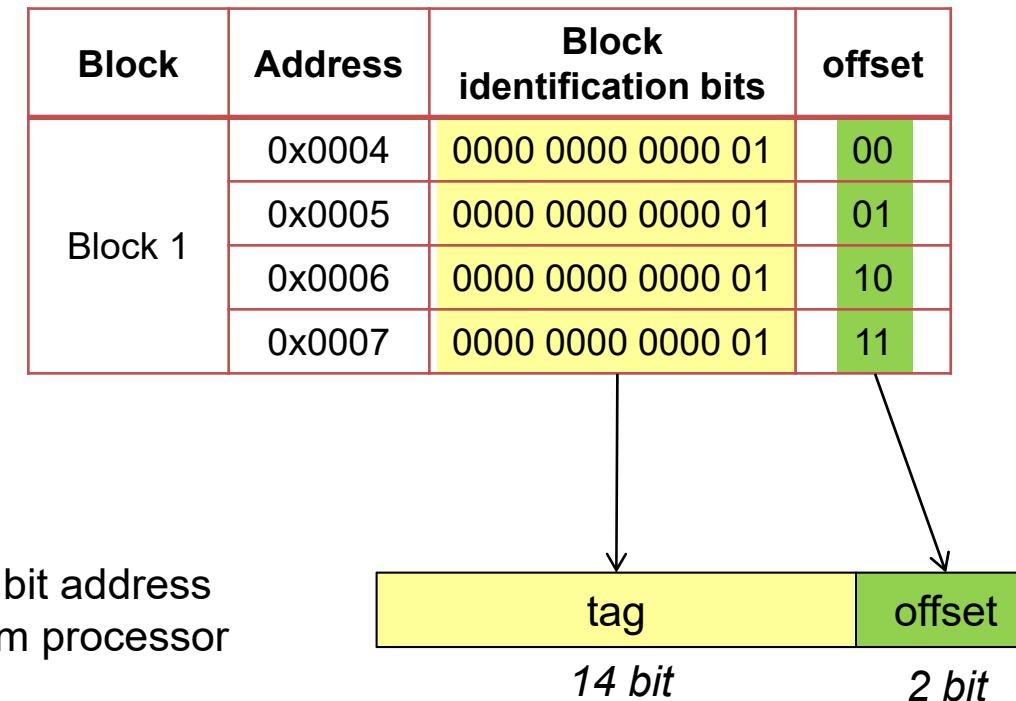
■ Three different cache models

- Fully associative
- Direct mapped
- N-way set associative



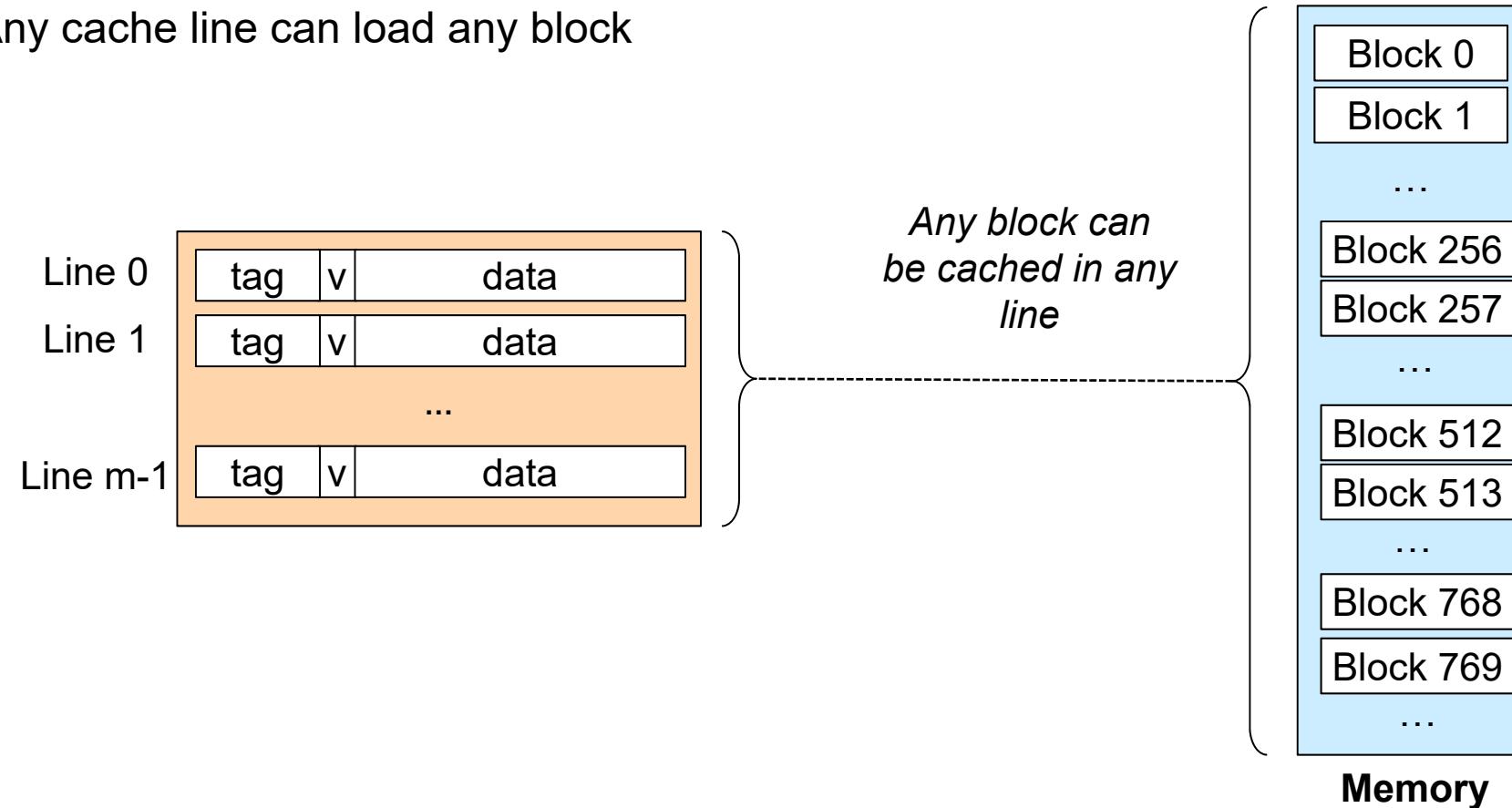
■ Addressing

- Tag contains complete block identification

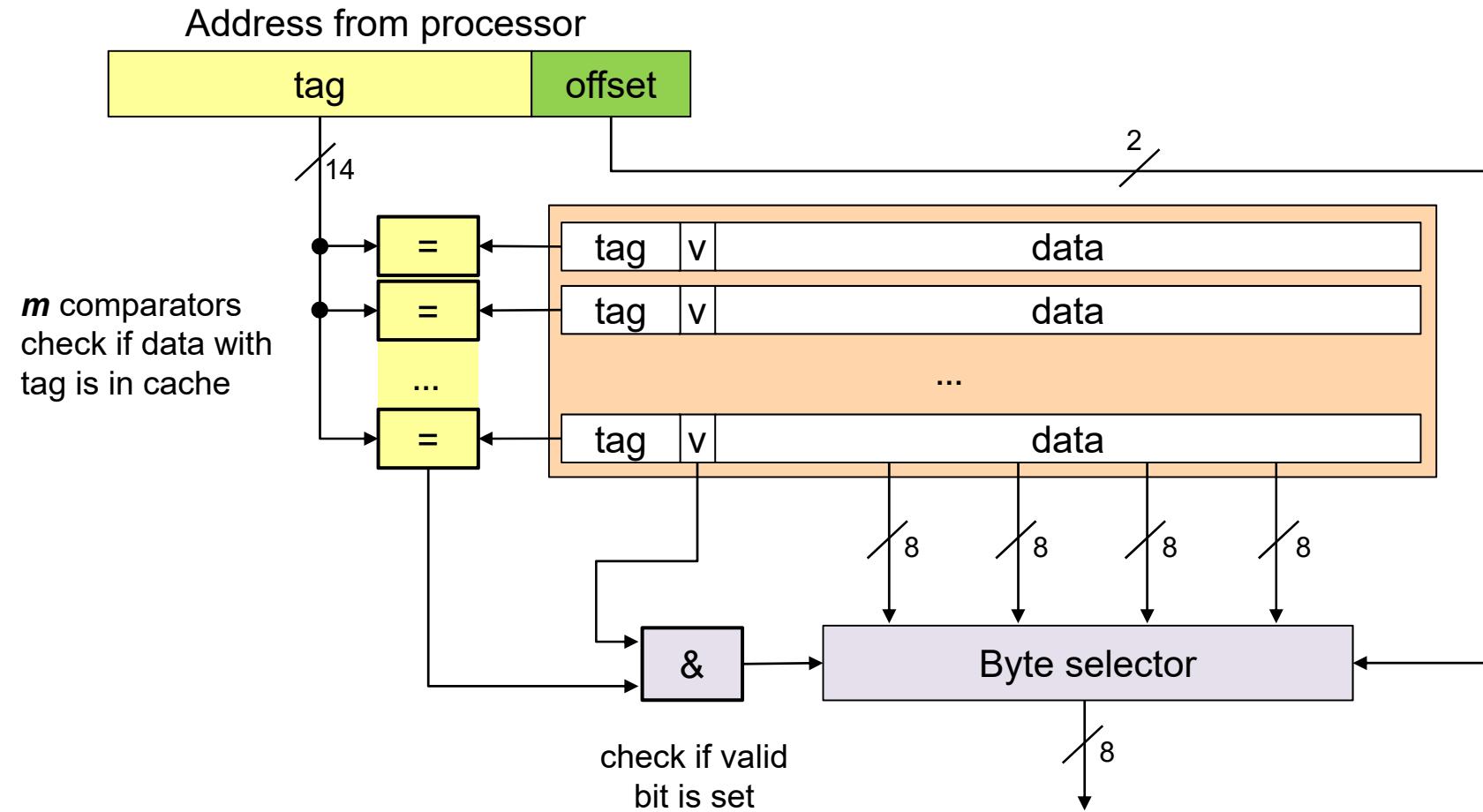


■ Organization

- Tag contains complete block identification
- Any cache line can load any block

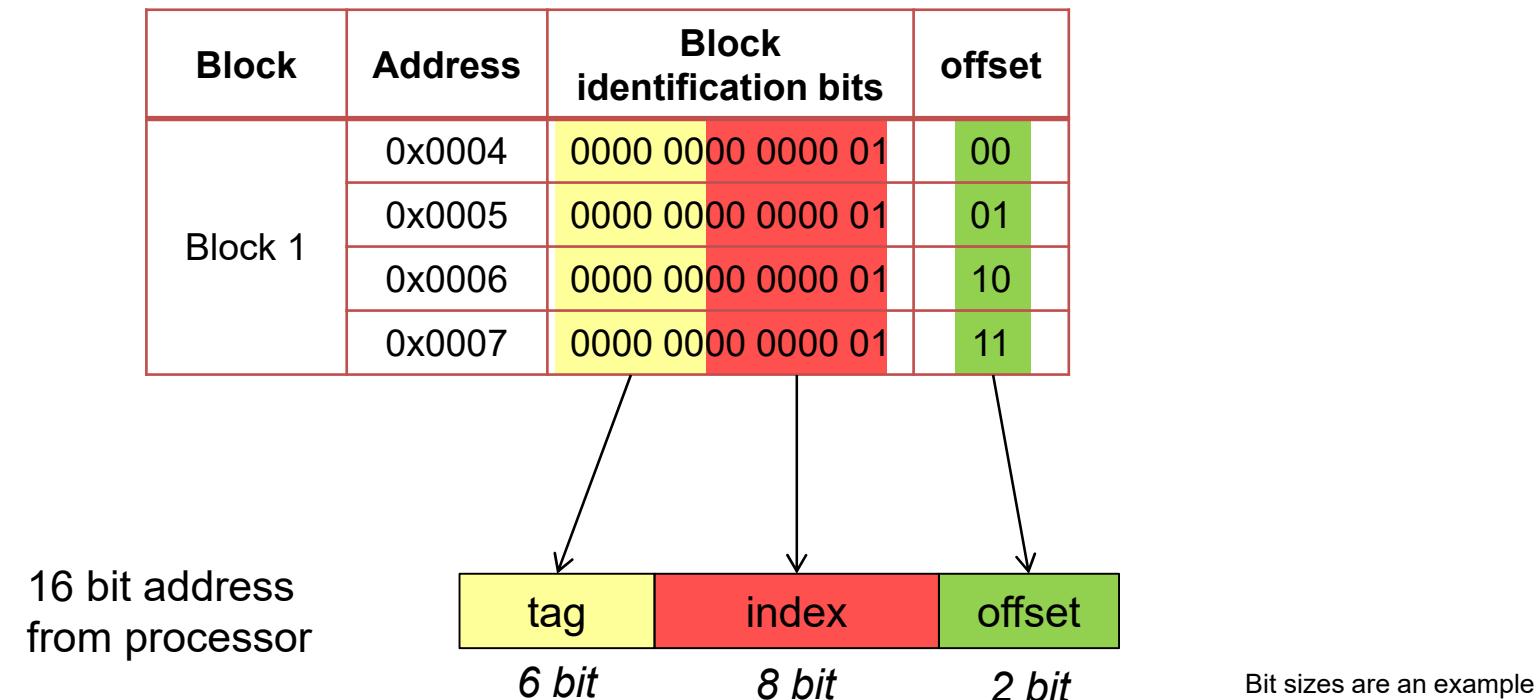


■ Architecture



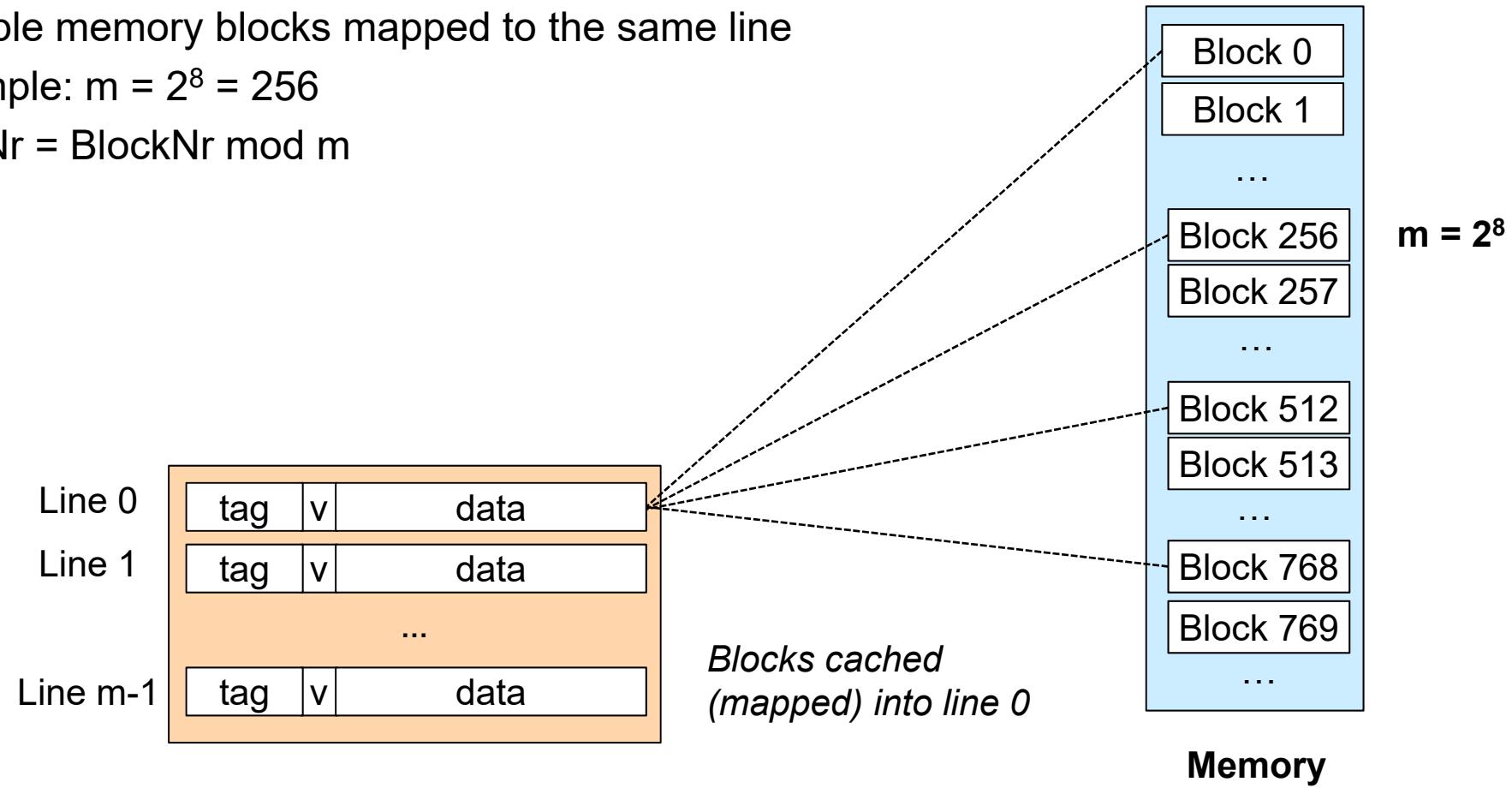
■ Addressing

- Block identification split into tag and index

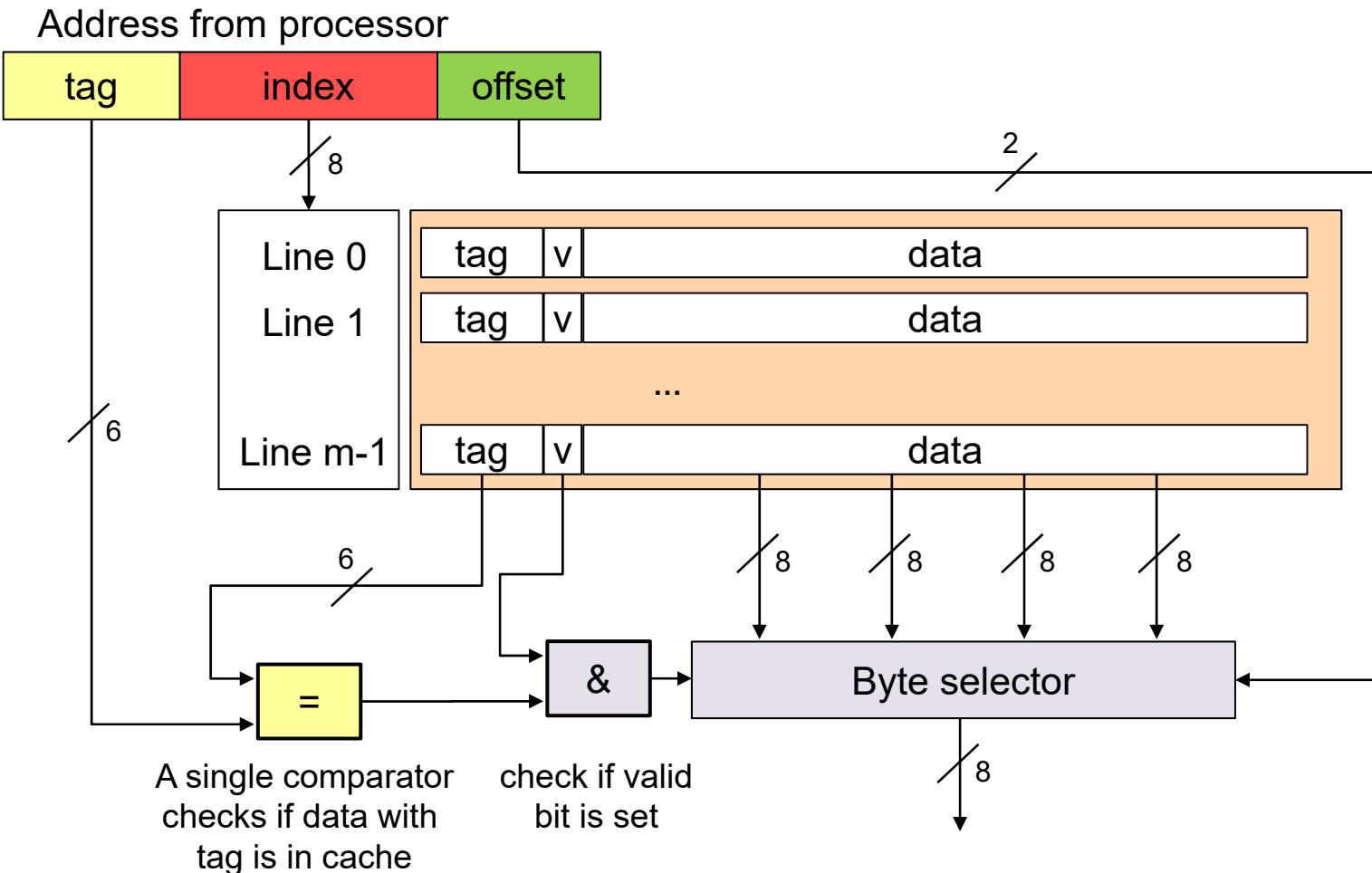


■ Organization

- Each memory block is mapped to exactly one cache line
- Multiple memory blocks mapped to the same line
- Example: $m = 2^8 = 256$
- $\text{LineNr} = \text{BlockNr} \bmod m$



■ Architecture



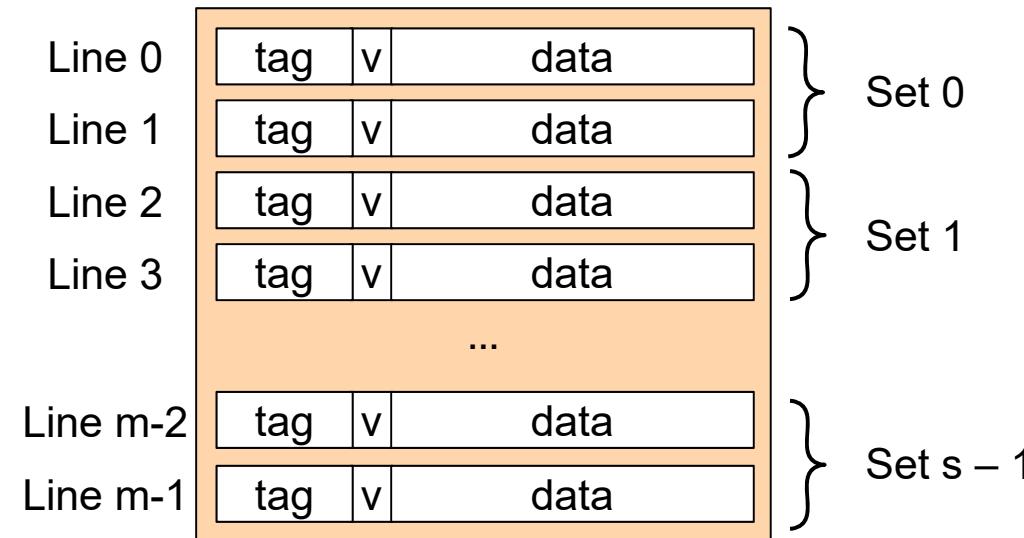
■ Organization

- Partition into sets
 - $s = m/n$ number of sets
 - n lines per set („N-way“)
 - b Bytes per line
- $s \times n \times b$ data Bytes

N-Way Set Associative

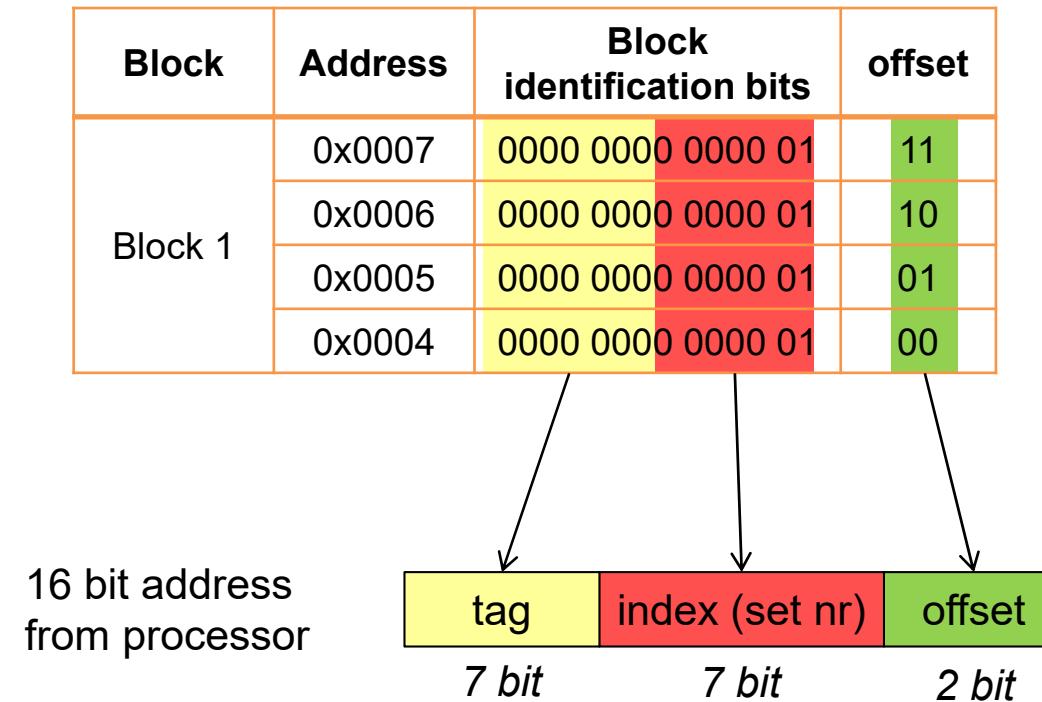
Finding a compromise between simple logic (Direct Mapped) and high hit rates (Fully Associative)

Example: $k = 2$



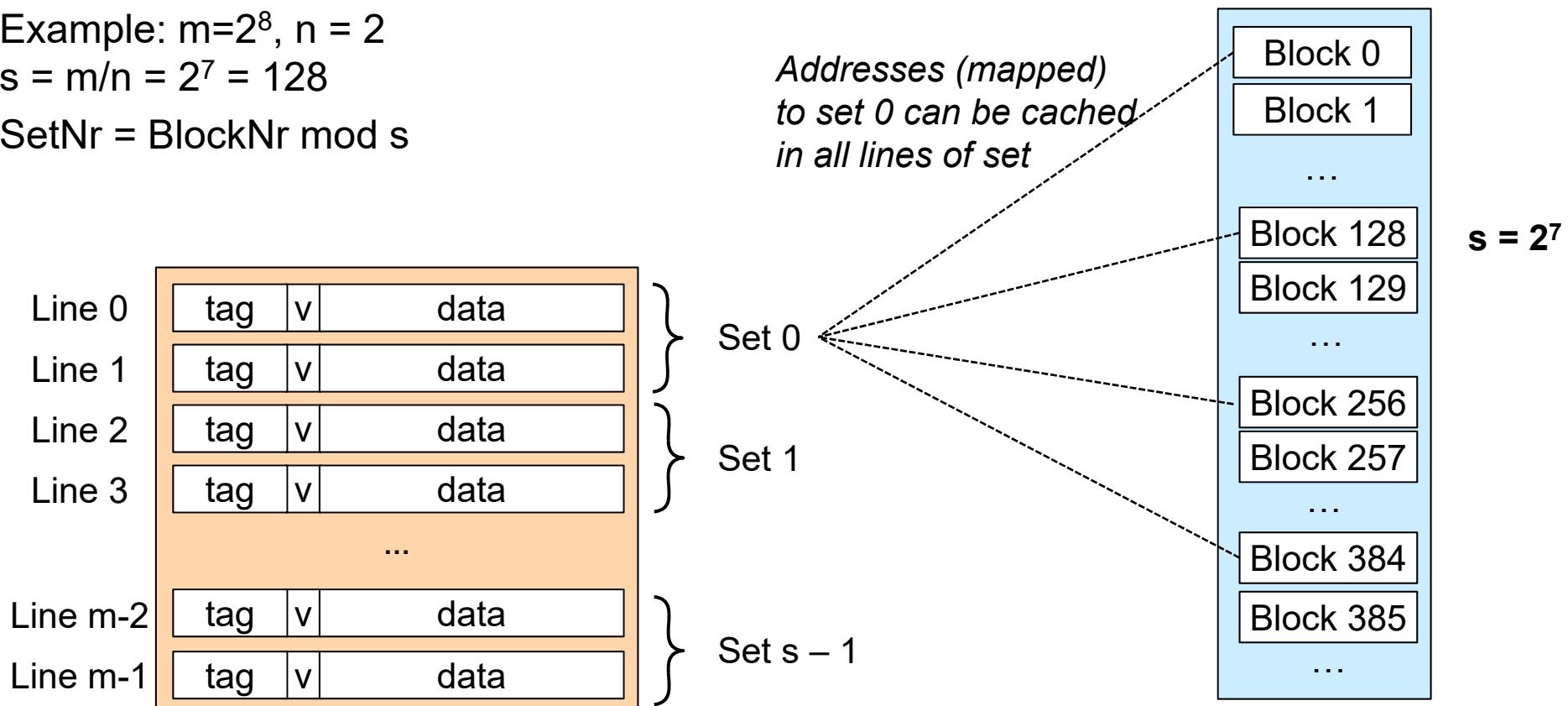
■ Addressing

- Example: $n = 2$
 - Maximum index corresponds to number of sets ($s = m/n$)



■ Organization

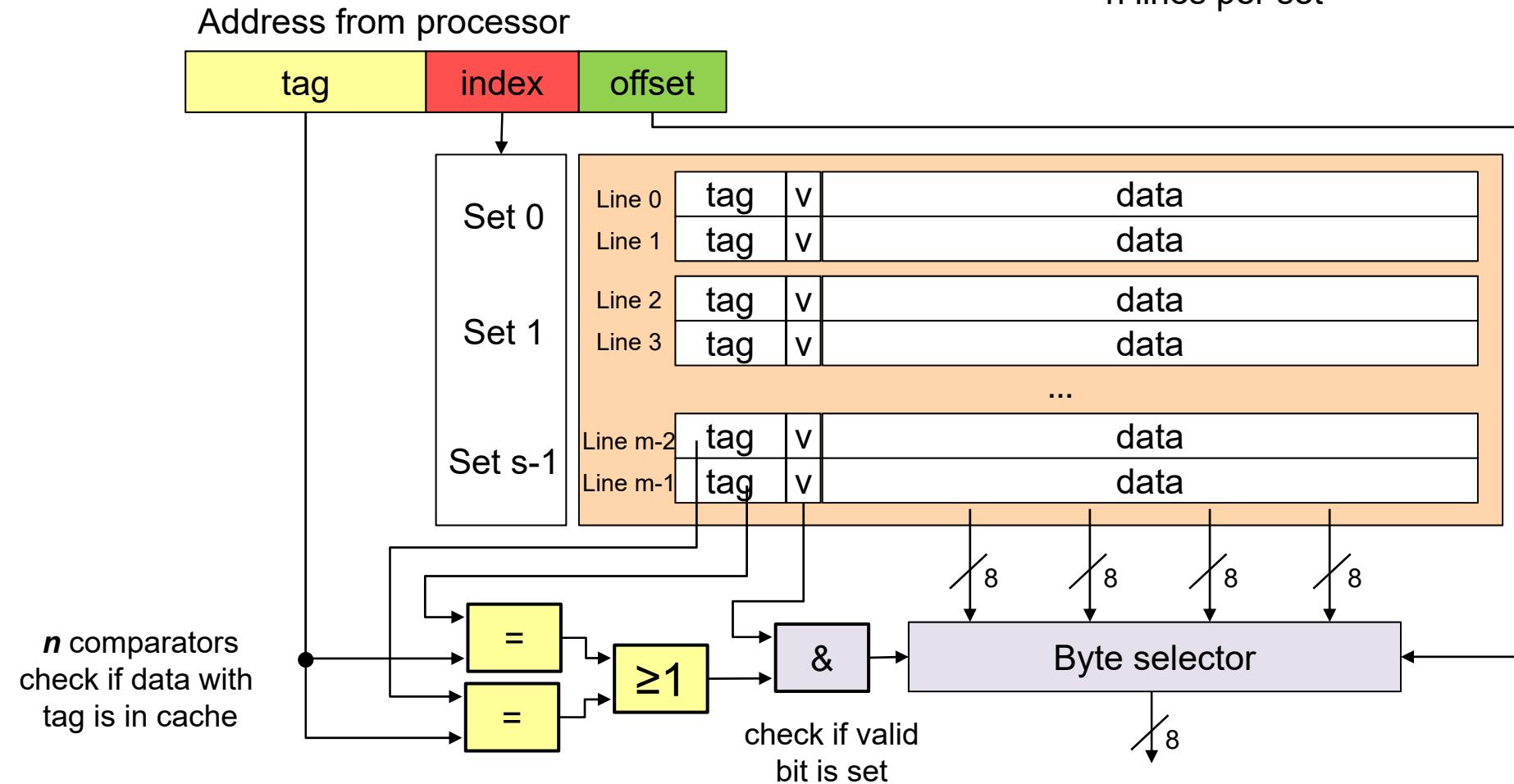
- Partition into sets
- Example: $m=2^8$, $n = 2$
 $s = m/n = 2^7 = 128$
- $\text{SetNr} = \text{BlockNr} \bmod s$



N-Way Set Associative

■ Architecture

Example: $n = 2$
 $s = m/n$ number of sets
 n lines per set



■ Comparison

Organization	Fully associative	Direct mapped	N-way set associative
Number of sets	1	m	m/n
Associativity	$m (=n)$	1	n
Advantages	<ul style="list-style-type: none">• Fast, flexible• Highest hit rates• Advanced replacement strategies	<ul style="list-style-type: none">• Simple logic• Replacement strategy defined by organization	Combination of both other concepts to combine advantages and to compensate disadvantages
Disadvantages	<ul style="list-style-type: none">• Complex logic: one comparator per line• Requires large area on silicon• Replacement can be complex	<ul style="list-style-type: none">• Lower hit rates	

■ **Cold miss**

- First access to a block

■ **Capacity miss**

- Working set larger than cache

■ **Conflict miss**

- Multiple data objects map to same slot

■ Hit rate / miss rate

- Fraction of memory references found / not found in cache
 - hit rate = $\text{nr_of_hits} / \text{nr_of_accesses}$
 - miss rate = $\text{nr_of_misses} / \text{nr_of_accesses} = 1 - \text{hit rate}$

■ Hit time

- Time to deliver a block in the cache to the processor

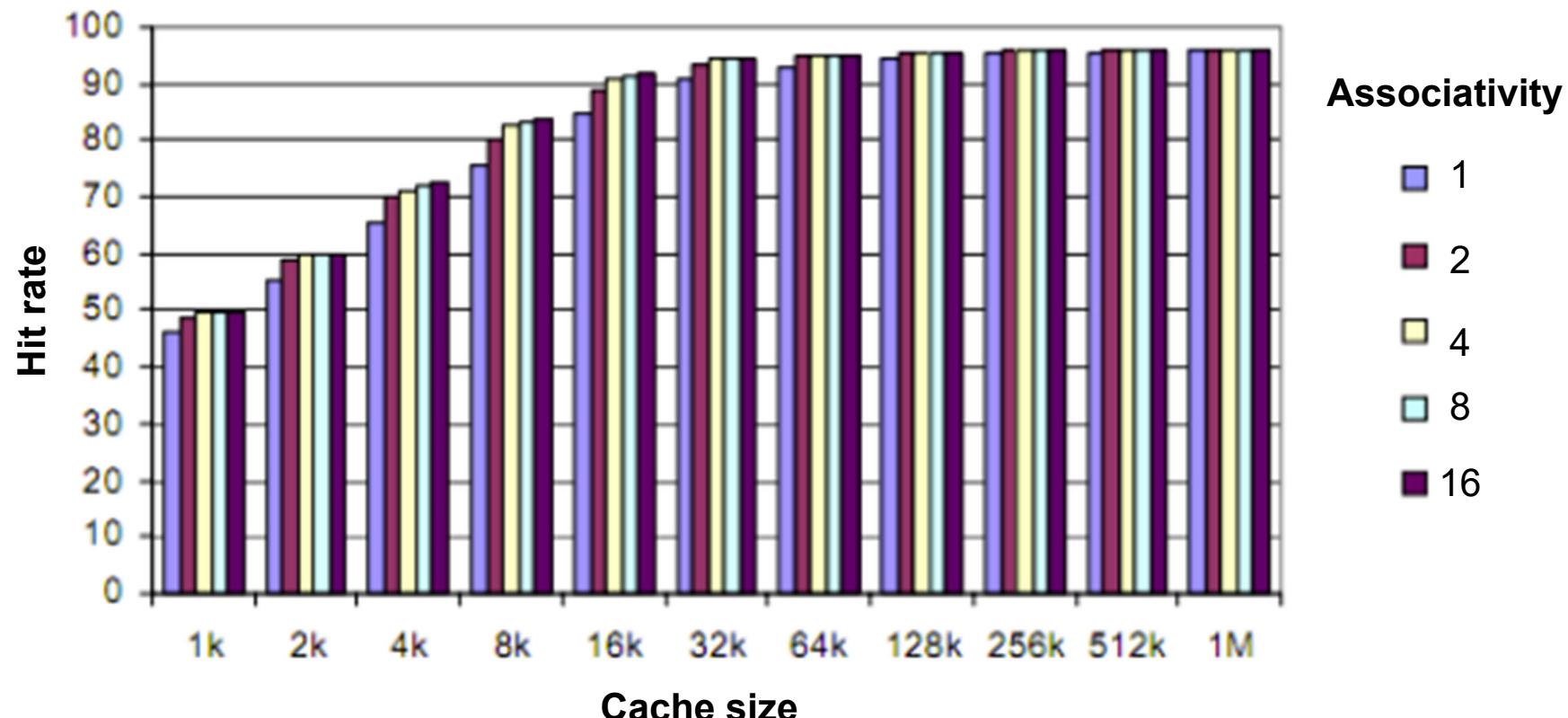
■ Miss penalty

- Additional time required to fetch data from memory because of a cache miss

- **High cache hit rate is important!**
 - 99% hit rate can be twice as fast as 97%
- **Example**
 - Cache hit time: 1 processor cycle
 - Miss penalty: 100 processor cycles
 - Average access time is:
 - 97% hits: $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles average}$
 - 99% hits: $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles average}$

■ Cache size versus hit rate

- Typical hit rate for Associativity 1, 2, 4, 8, 16 and cache size



■ Selecting cache line to replace by

- LRU: Least recently used
- LFU: Least frequently used
- FIFO: First In–First Out → oldest
- Random Replace: randomly chosen

*additional information
needed in cache*

*simple, but still
good performance*

- Only relevant for Fully- / N-way associative caches
→ Hard coded in cache implementation (=hardware)

■ What to do on a write hit*?

- Write-through
 - Write immediately to memory
- Write-back
 - Delay write to memory until replacement of line
(needs a valid bit)

■ What to do on a write miss**?

- Write-allocate
 - Load line into cache (from memory) and update line in cache
- No-write-allocate
 - Writes immediately to memory

* Write hit: data already in cache

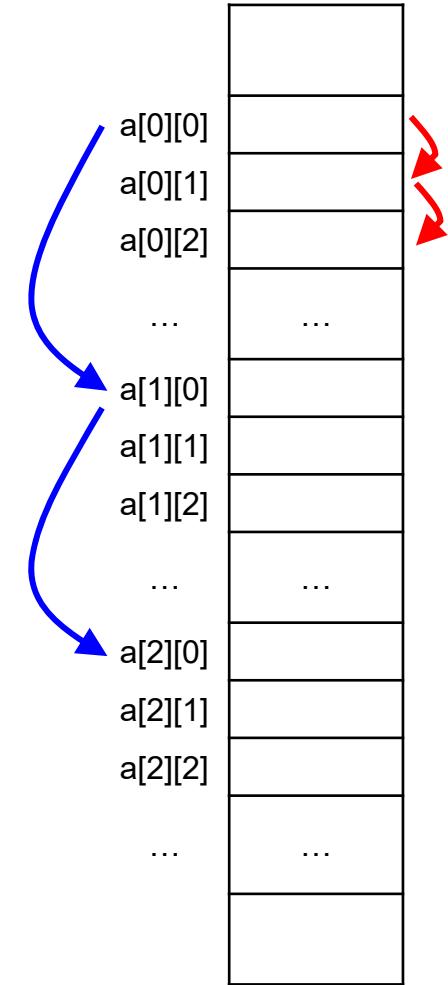
** Write miss: data not in cache

The Programmer's Perspective

- **For loops over multi-dimensional arrays**
 - Example: matrices (2-dim arrays)
- **Change order of iteration to match layout**
 - Gets better spatial locality
 - Layout in C: last index changes first!

```
for(j = 0; j < 10000; j++) {  
    for(i = 0; i < 40000; i++) {  
        c[i][j] = a[i][j] + b[i][j];  
    }  
}  
  
// a[i][j] and a[i+1][j]  
// are 10'000 elements apart
```

```
for(i = 0; i < 40000; i++) {  
    for(j = 0; j < 10000; j++) {  
        c[i][j] = a[i][j] + b[i][j];  
    }  
}  
  
// a[i][j] and a[i][j+1]  
// are next to each other
```



The Programmer's Perspective

■ Change order of iteration to match layout

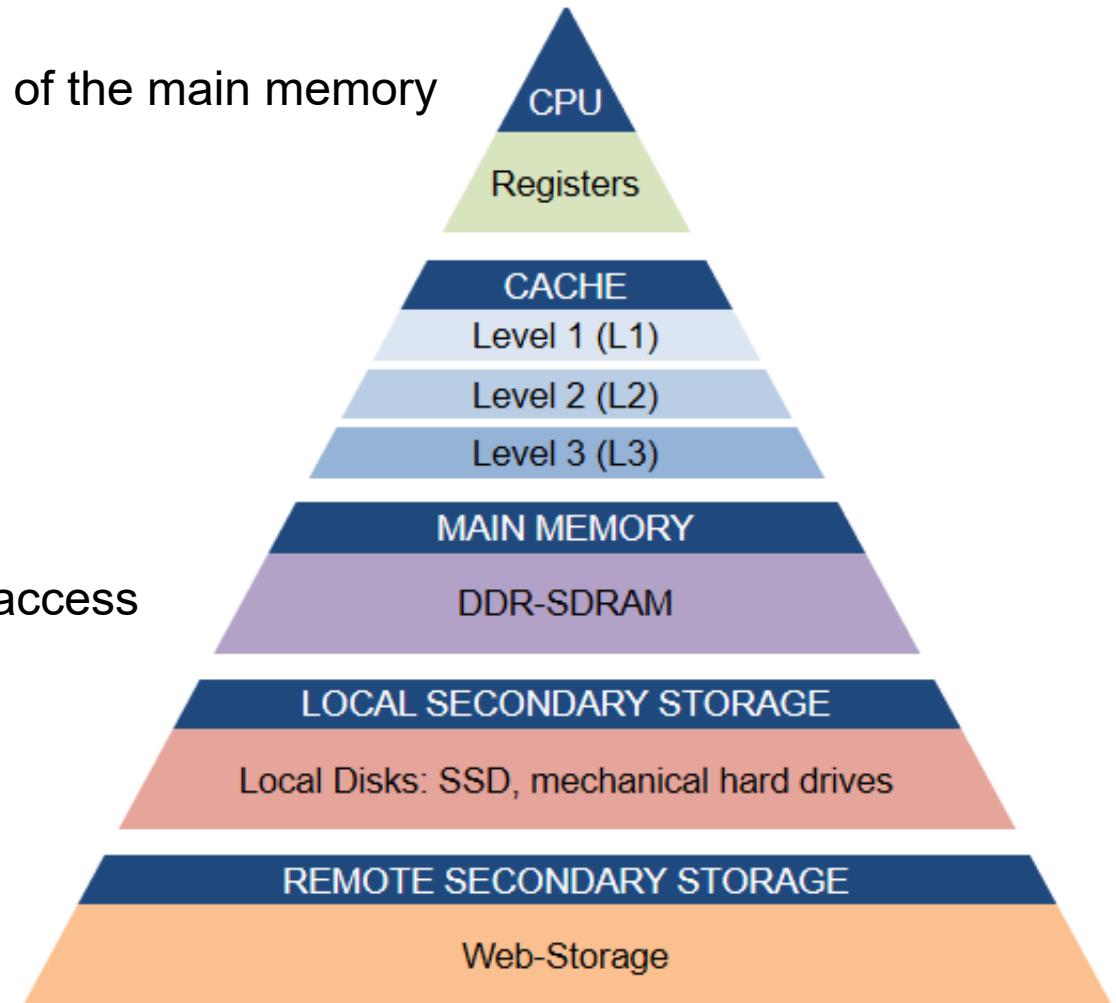
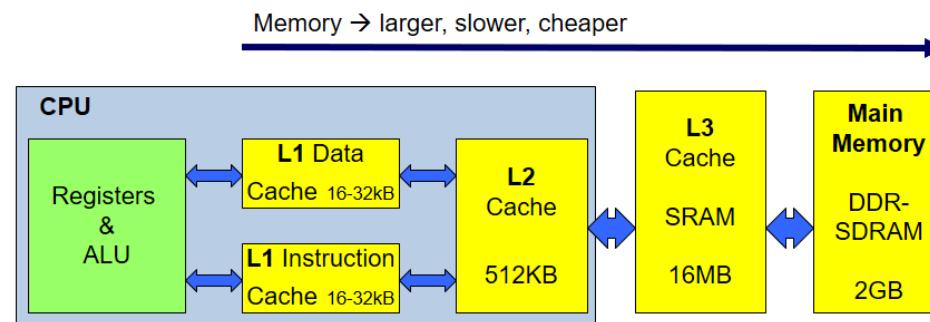
```
j is inner  
1047527424-1048575999 done  
  
real    0m3.460s  
user    0m3.452s  
sys     0m0.000s  
  
j is outer  
1047527424-1048575999 done  
  
real    0m18.509s  
user    0m18.472s  
sys     0m0.000s
```

```
#include<stdio.h>  
#include<time.h>  
  
int main( int argc, char** argv )  
{  
    int iterations = 1000;  
    int size = 1024;  
  
    int array[size][size];  
    int v = 0;  
  
    if (argc==2) {  
        printf("j is inner\n");  
        for (int n=0; n<iterations; n++) {  
            for (int i=0; i<size; i++) {  
                for (int j=0; j<size; j++) {  
                    array[i][j] = v;  
                    v++;  
                }  
            }  
        }  
    } else {  
        printf("j is outer\n");  
        for (int n=0; n<iterations; n++) {  
            for (int j=0; j<size; j++) {  
                for (int i=0; i<size; i++) {  
                    array[i][j] = v;  
                    v++;  
                }  
            }  
        }  
    }  
    printf("%i-%i done\n", array[0][0], array[size-1][size-1]);  
    return 0;  
}
```

Conclusion

■ Cache

- Cache allows fast data access to selected parts of the main memory which are mirrored in the cache
 - Spatial and temporal locality
- Different cache models
 - Fully associative
 - Direct mapped
 - N-way associative
- Replacement / Write strategies
- Reducing cache misses by optimizing memory access



Driver for Cache and Pipelining

ACAT2013

Journal of Physics: Conference Series **523** (2014) 012002

IOP Publishing

doi:10.1088/1742-6596/523/1/012002

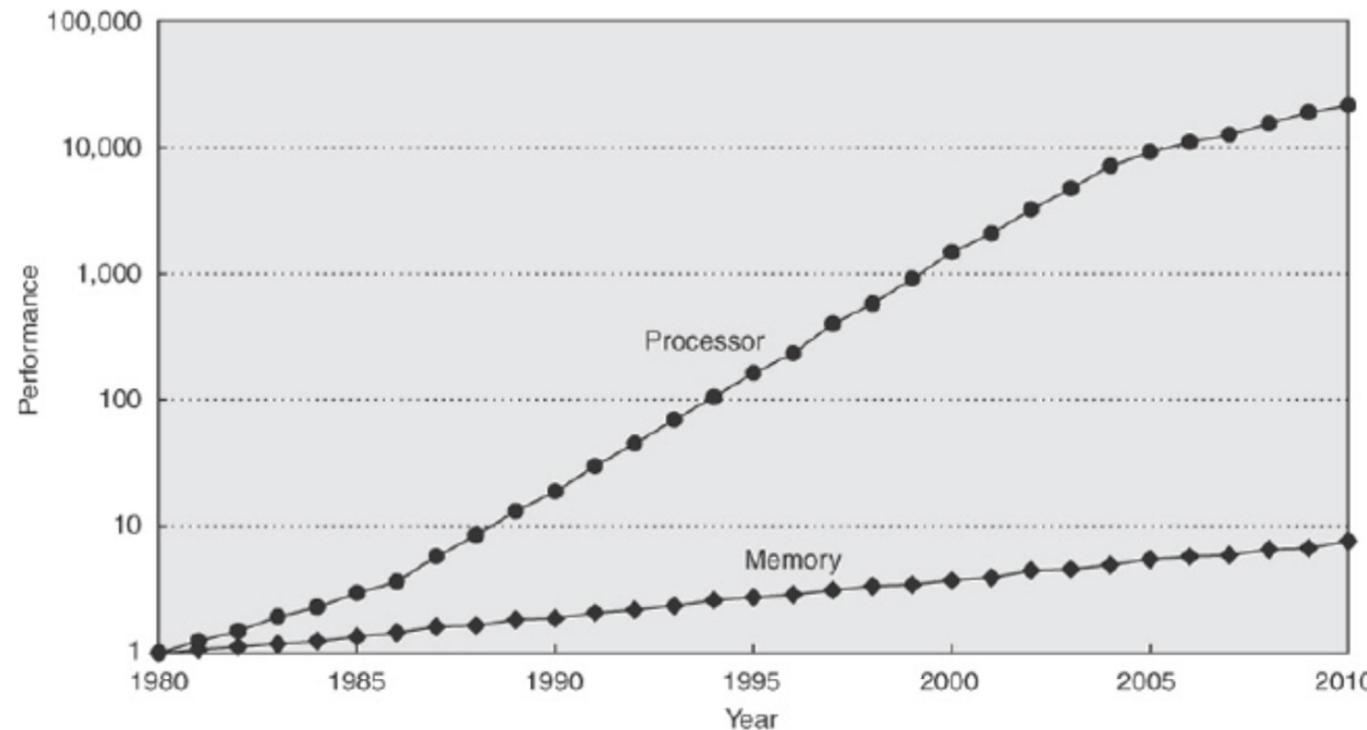


Figure 1: CPU-memory performance gap. Modelled after "Computer Architecture": Hennessy, John L.; Patterson, David A.

https://www.researchgate.net/publication/273029990_Opportunities_and_choice_in_a_new_vector_era

Cache Architecture Comparison

L1

L2

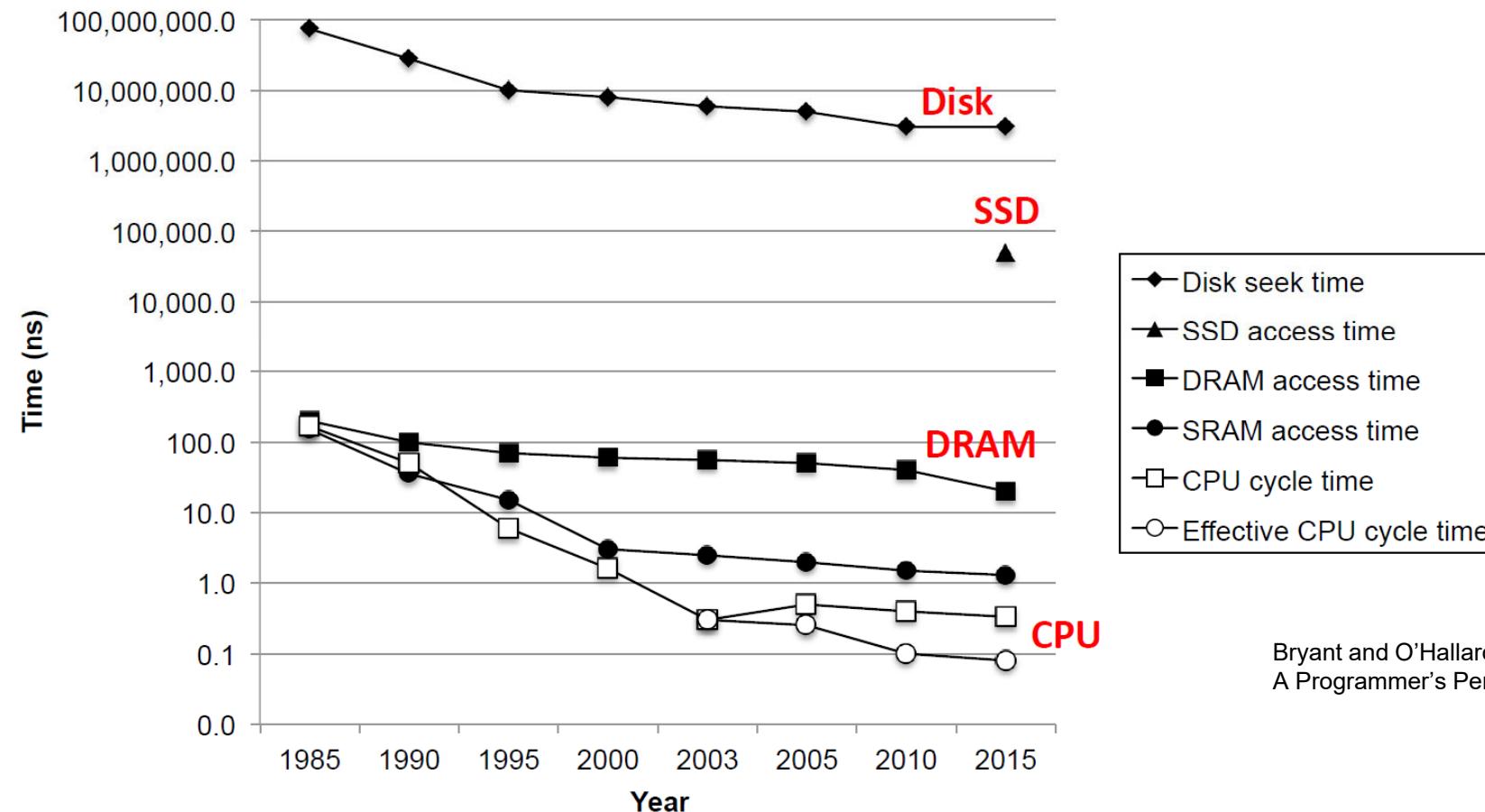
L3

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization		Unified (instruction and data)
L3 cache size		8 MiB, shared
L3 cache associativity		16-way set associative
L3 replacement		Approximated LRU
L3 block size		64 bytes
L3 write policy		Write-back, Write-allocate
L3 hit time		35 Clock Cycles

Source: Patterson / Hennessy: Computer Organization and Design, 5th edition, Elsevier

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.

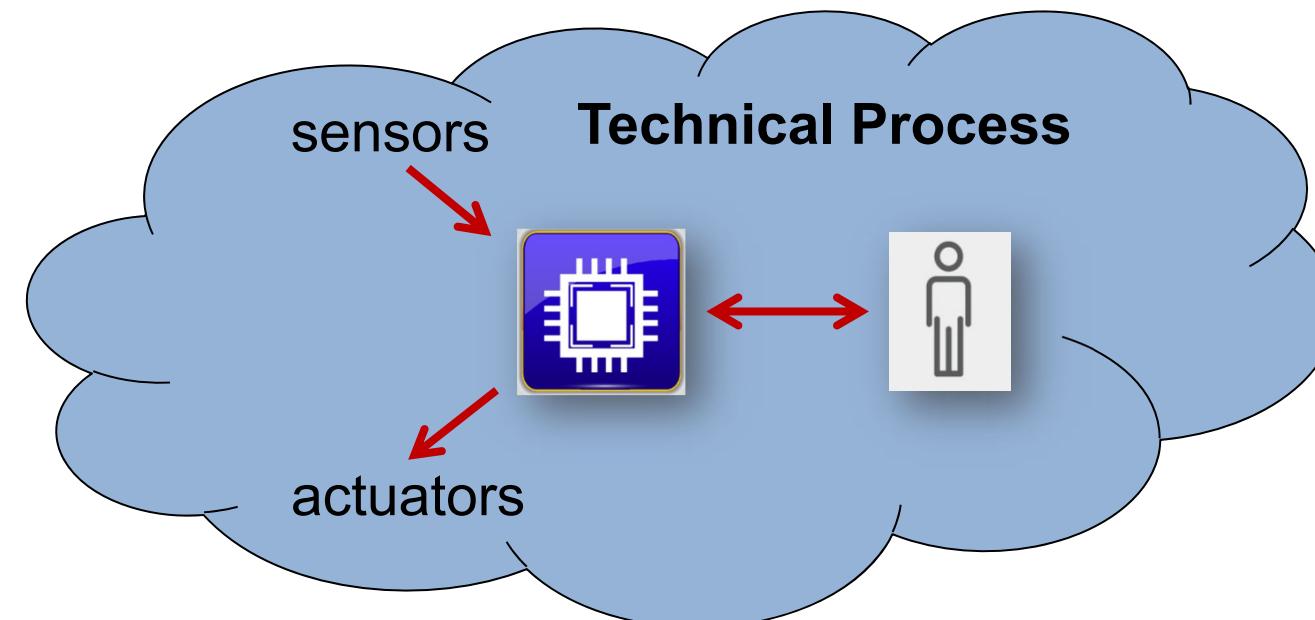
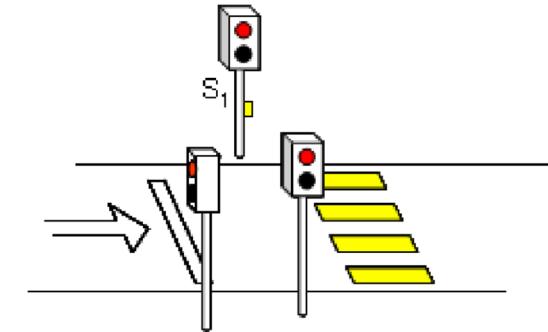


Bryant and O'Hallaron, Computer Systems:
A Programmer's Perspective, Third Edition

Software State Machines

Computer Engineering 2

- **Embedded Systems**
 - Embedded in technical context / process
 - Application specific
- **How is a Finite State Machine (FSM) modeled in software?**



- **Repetition: FSM in Hardware**
- **FSM in Software**
- **Modeling State Machines in UML**
- **Implementation in C**
- **Interaction of FSMs**

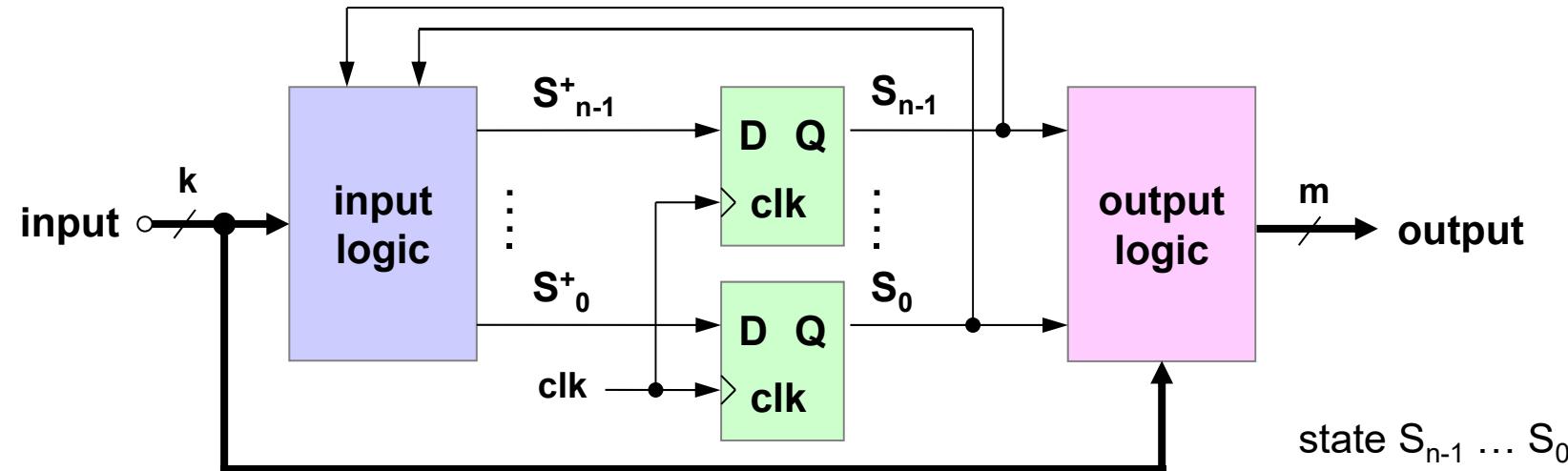
At the end of this lesson you will be able

- to explain the term «Finite State Machine» (FSM)
- to outline why FSMs in software are often modeled in a different way than in hardware
- to explain the concept of a reactive system (state-event model)
- to correctly use the related terms and to interpret a UML state diagram with the basic elements
- to correctly model/describe an FSM using the basic elements of a UML state diagram
- to name and describe the semantics of an UML FSM
- to translate a simple FSM modeled in UML into C-Code

Repetition: FSM in Hardware

■ Finite State Machine (FSM) in hardware

- Flip-flops store internal state
- Clock-driven
 - Inputs are evaluated¹⁾ at each clock edge
 - State can only change on a clock edge

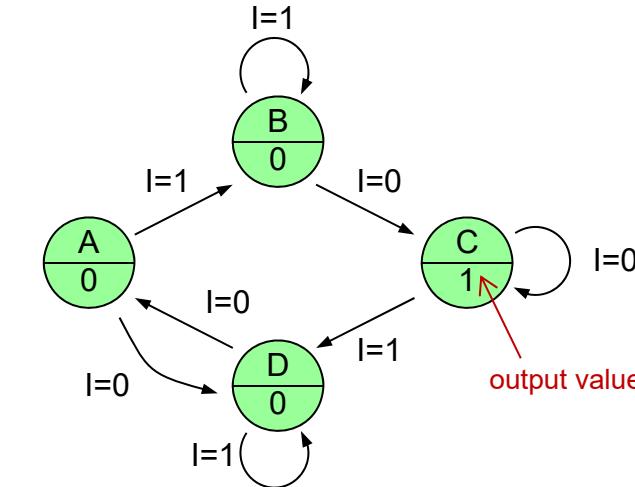
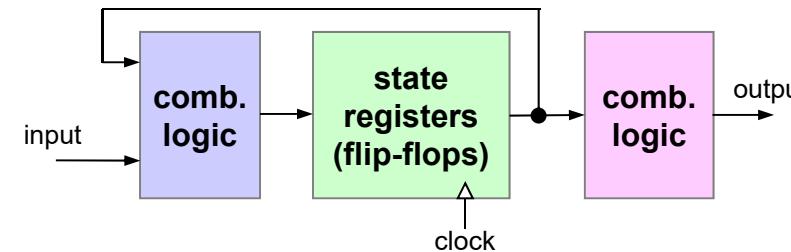


¹⁾ Evaluation of inputs → interpretation of signal levels: '0' vs. '1'

Repetition: FSM in Hardware

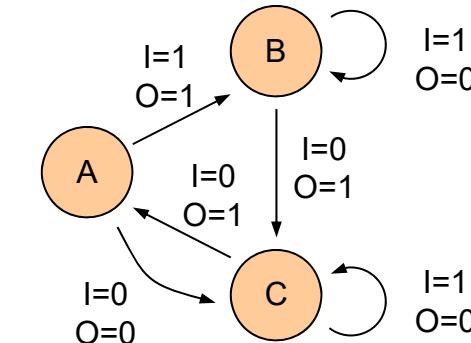
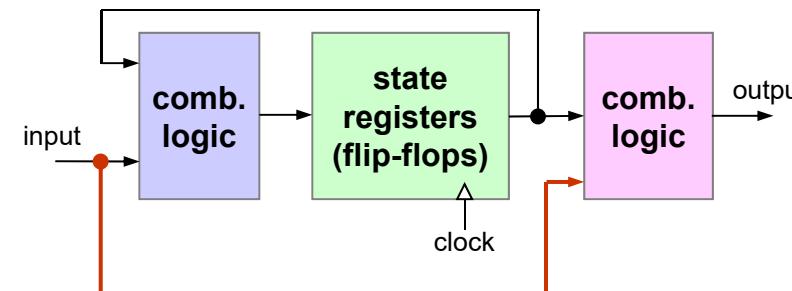
■ Moore

- Input signals influence state only



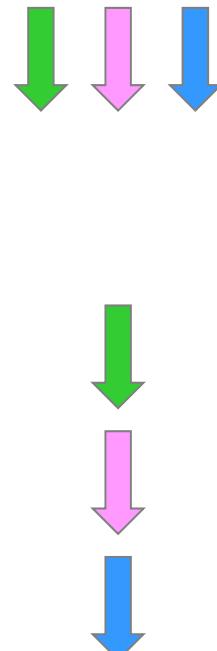
■ Mealy

- Input signals influence state AND output signals



■ Why software is different

- **Hardware** is intrinsically parallel
 - Several FSMs can be processed in parallel using the same clock – Large number of flip-flops and gates evaluate simultaneously
 - Use of common clock as the only event
 - Evaluate signal levels of inputs at clock edges
 - Unchanged signal levels of inputs do not create overhead
- **Software** is intrinsically sequential
 - CPU has to process one FSM after the other
 - "HW approach" would require a function call on each clock edge
 - All FSM inputs would have to be evaluated on each function call even if they have not changed → creates a large processing load for CPU
 - Cooperating FSMs: Using a "synchronous clock approach" as in hardware creates a lot of synchronization issues in sequential system



→ use a different approach for software

■ Reactive system → State-event model

- Responds to external events (input)
 - Event-driven
 - Only evaluate the FSM if an input changes
- Internal state
 - Memory of what happened before
- Actions
 - Influence the outside world
- Each event may or may not
 - Change the internal state
 - Trigger actions



source www.thediligentadvisor.com

Depending on the current state an event may have a different effect.

■ State-event applications

Process control

- Washing machines
- Vending machines
- Heating systems



Communication protocols

- Opening and closing connection



Human-machine interface

- Recognition and validation of user inputs



Parsing

- Programming languages



■ UML – Unified Modeling Language

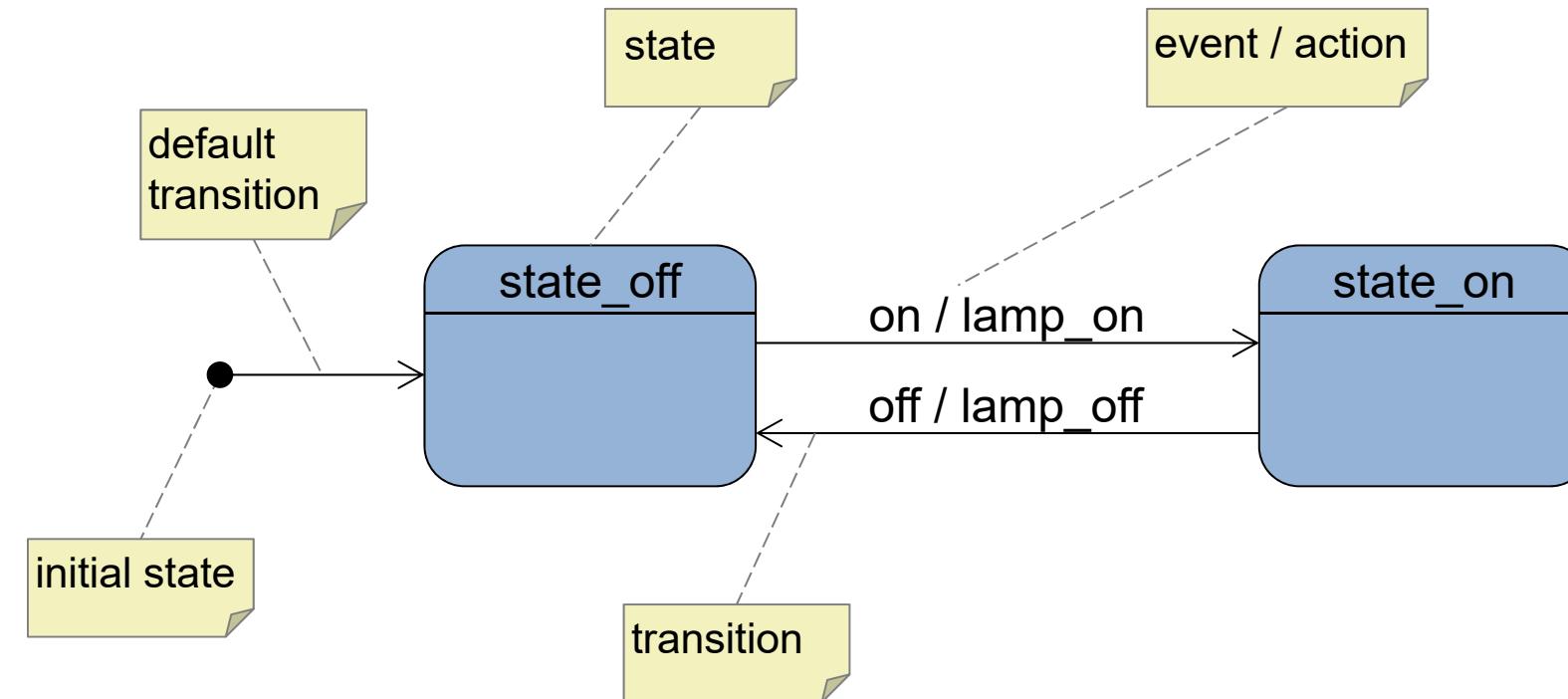
- Graphical modeling of software systems
- Object oriented concepts
- Introduced in the 1990s
 - Since 1997 maintained by Object Management Group (OMG)
 - Since 2000 certified by ISO
- State diagrams
 - Only a part of UML
 - Describe the reactive behavior of classes
 - Based on notation of Prof. David Harel



We only cover a small part of the available state diagram notations

Modeling State Machines in UML

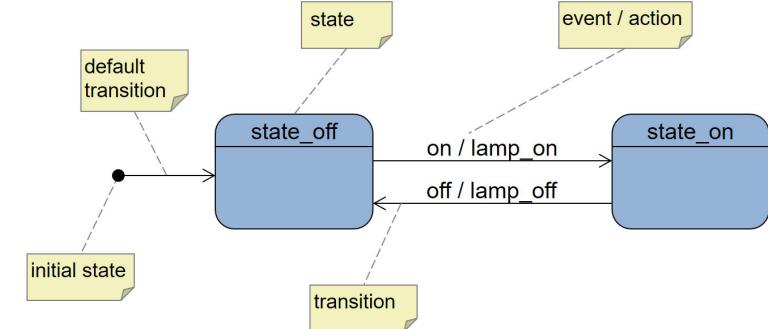
■ UML state diagram example



Modeling State Machines in UML

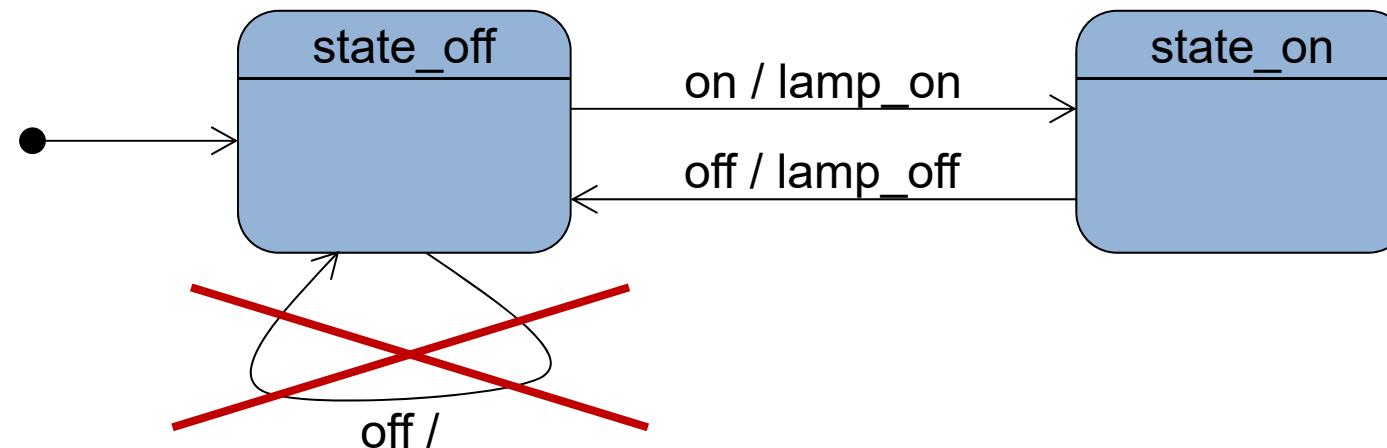
■ Terminology

- State
 - Internal state of the system in which it is awaiting the next event
- Event
 - Asynchronous input that may cause a transition
- Transition
 - Reaction to an event: May change the state and/or trigger an action
- Action
 - Output associated with a transition
 - ▶ Either a directly carried out operation or a
 - ▶ Message to another FSM (seen as an event by the receiving FSM)
- Finite State Machine (FSM)
 - Machine with a finite number of states and transitions



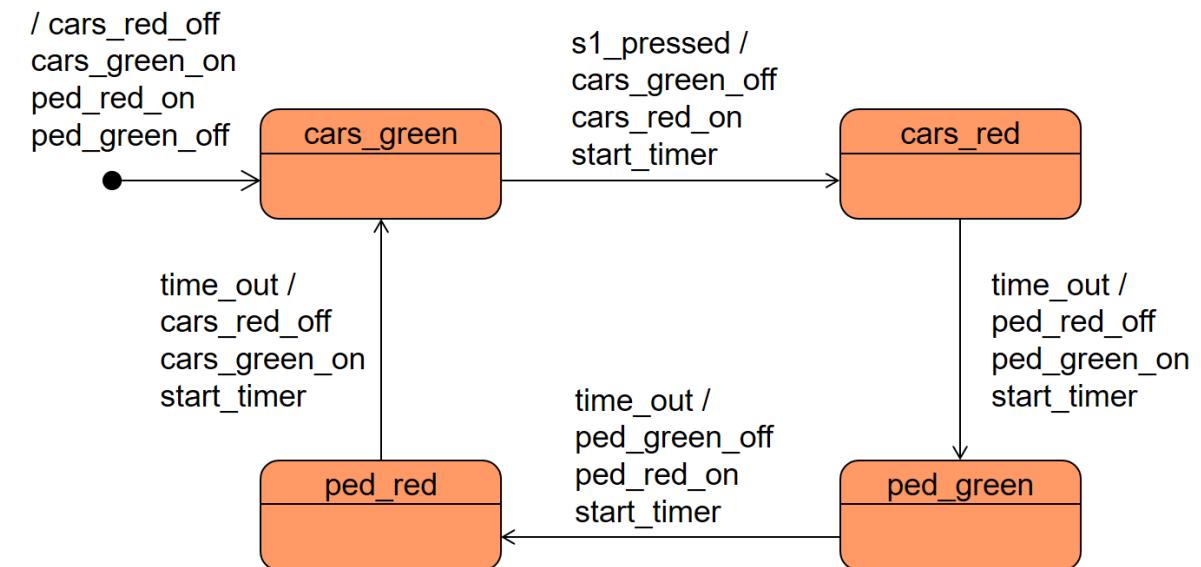
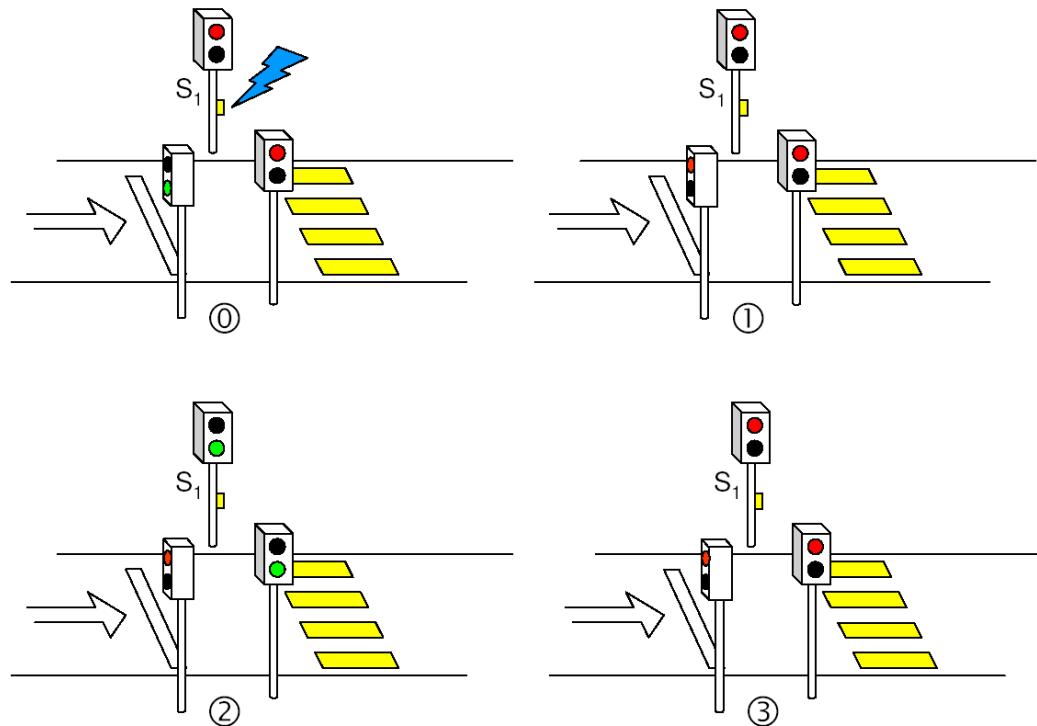
■ In contrast to Mealy notation

- Input asynchronous event
- Output actions → interaction with outside world
- Inputs without effect are omitted
 - If an event triggers neither a state transition nor an action it will not be drawn in the UML state diagram
 - Increases clarity of diagram
- UML contains Mealy and Moore notations as a subset



Modeling State Machines in UML

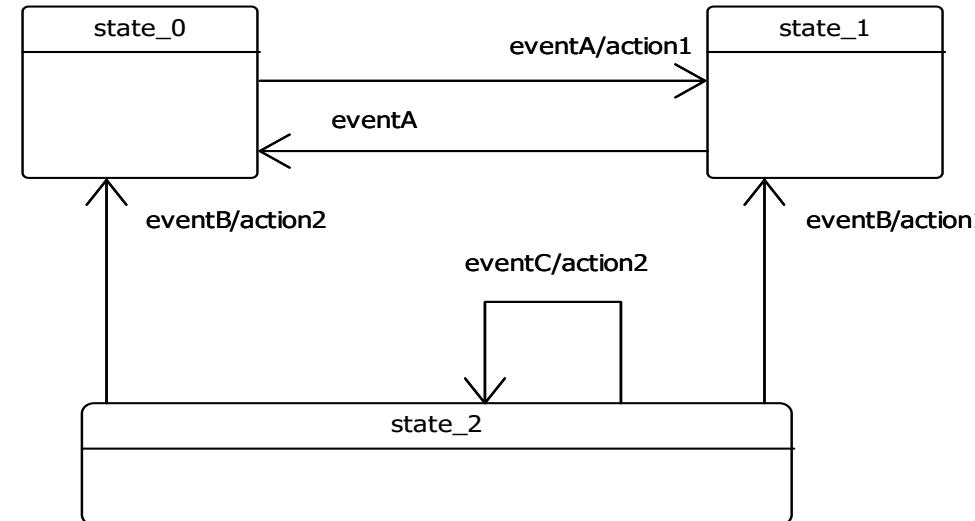
■ Example: Traffic light



■ Rules for UML state machines

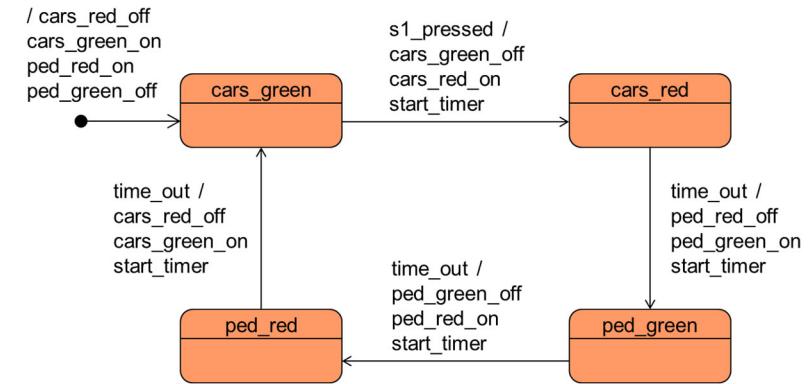
- Every state-diagram must have an initial state
- Each state has to be reachable through a transition
- The state-diagram has to be deterministic
 - i.e. for each event it has to be defined which transition is triggered

What's wrong?



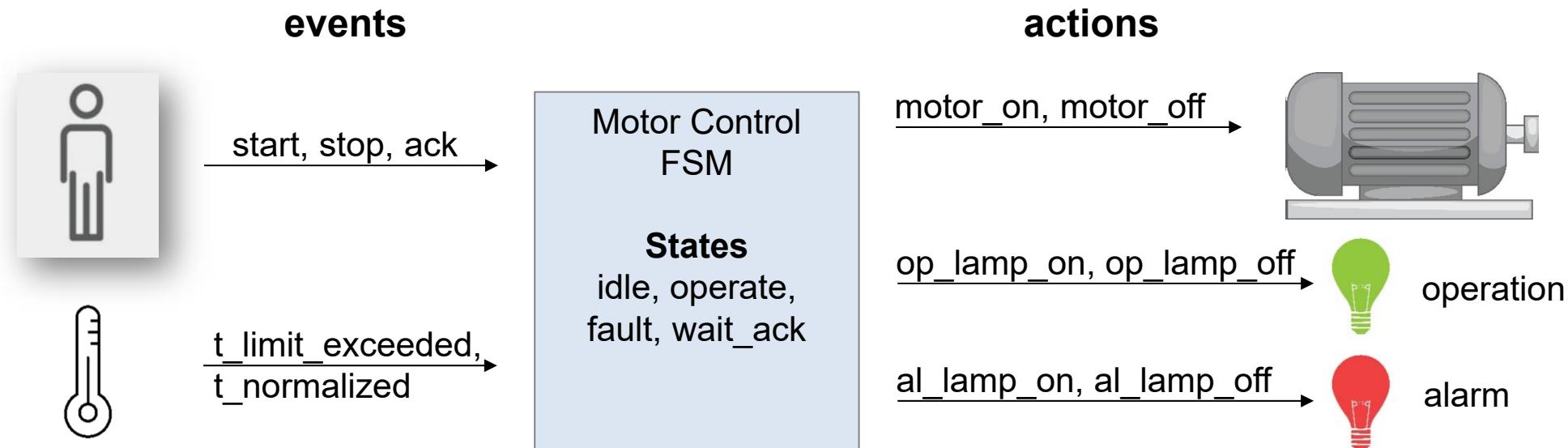
■ Semantics of an UML FSM → understanding the model

- FSM is passive
 - It only reacts to events from the outside
 - Does not initiate any actions on its own
- Always has a defined state
- Reaction to an event depends on the current state
 - State defines the corresponding transition for an event
 - Event is discarded (lost) if no transition for it exists in current state
- Run-to-completion
 - Once started a transition cannot be interrupted
- Strives to avoid querying additional input
 - The executed transition only depends on the event and not on additional inputs

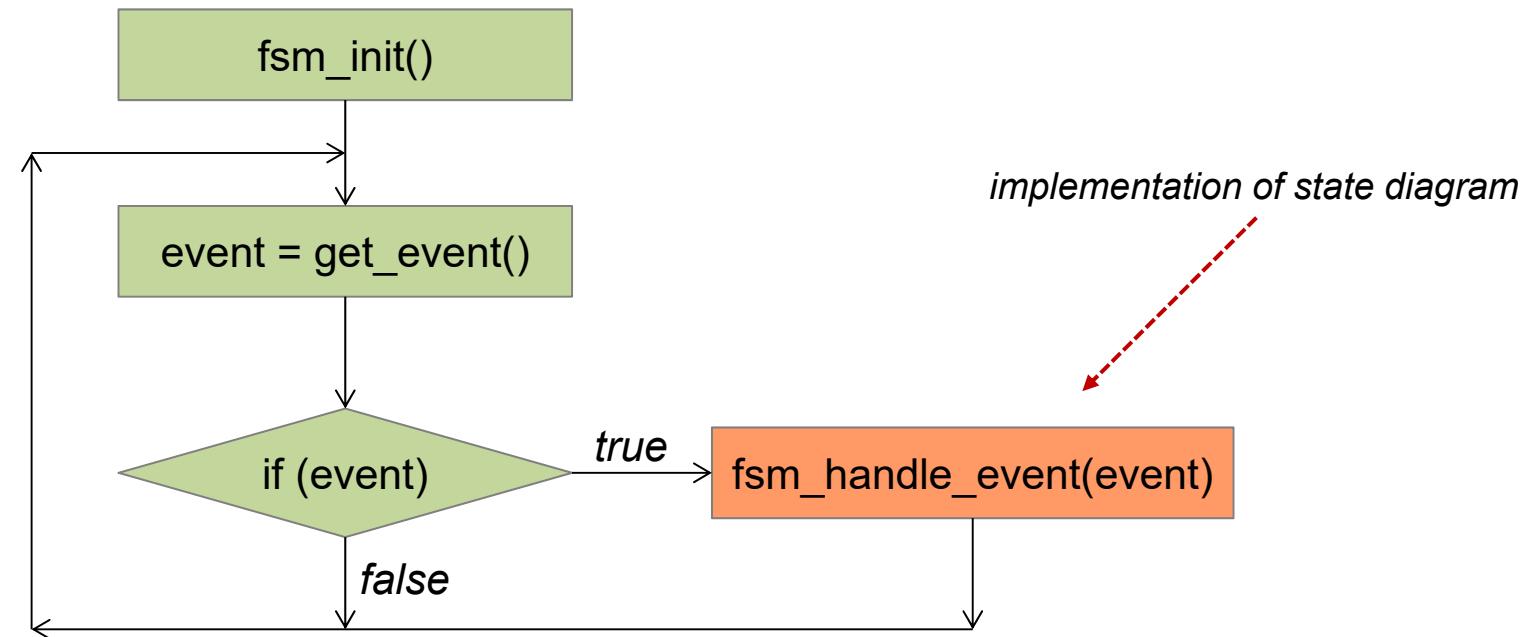


■ Exercise: Draw the UML-State-Diagram for “Motor Control”

- The operator shall be able to turn the motor on and off by pressing the start button and the stop button respectively.
- Motor control shall switch the motor off if the temperature limit is exceeded.
- Restarting the motor after a temperature shutdown shall require pressing the acknowledge button before pressing the start button.
- Normal operation shall be indicated with an operation lamp whereas an alarm condition shall be indicated with an alarm lamp.

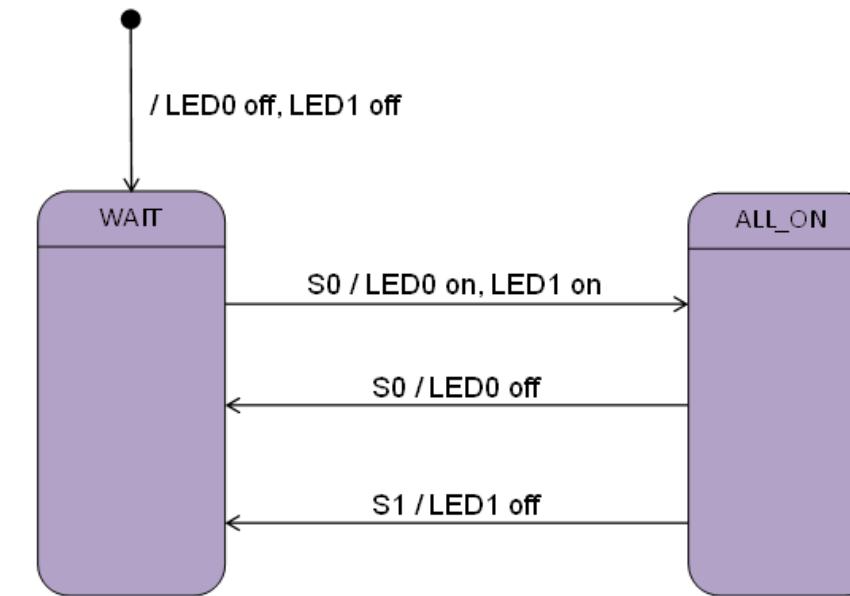
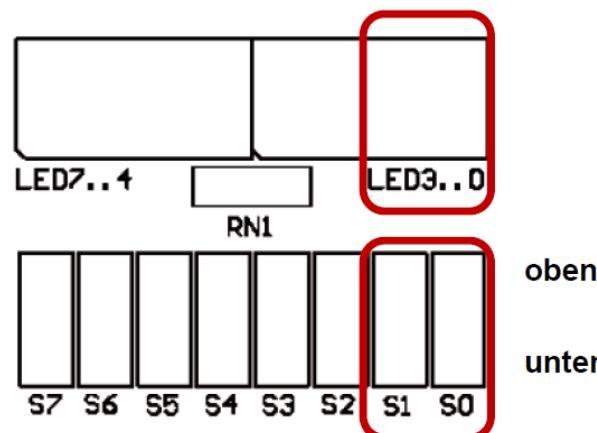


■ Simple FSM System



■ Example: LED control on CT-board

- Shifting switch Sx from 'unten' to 'oben' → event Sx
- $x \in \{ 0, 1 \}$



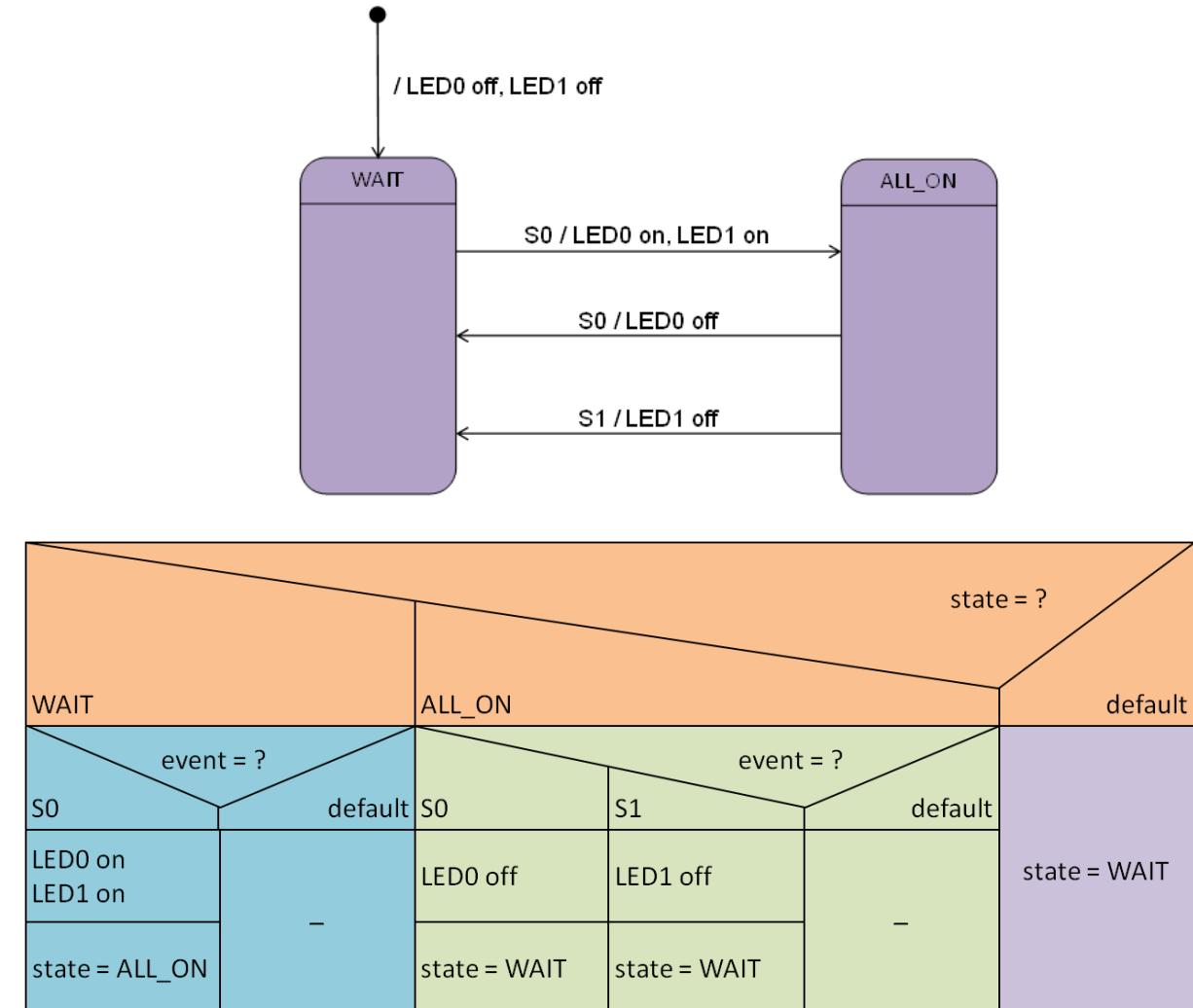
Implementation in C

■ Example: LED control

```
int main(void)
{
    event_t event;

    fsm_init();
    while (1) {
        event = get_event();
        if (event != NO_SWITCH) {
            fsm_handle_event(event);
        }
    }
}
```

→ see code example

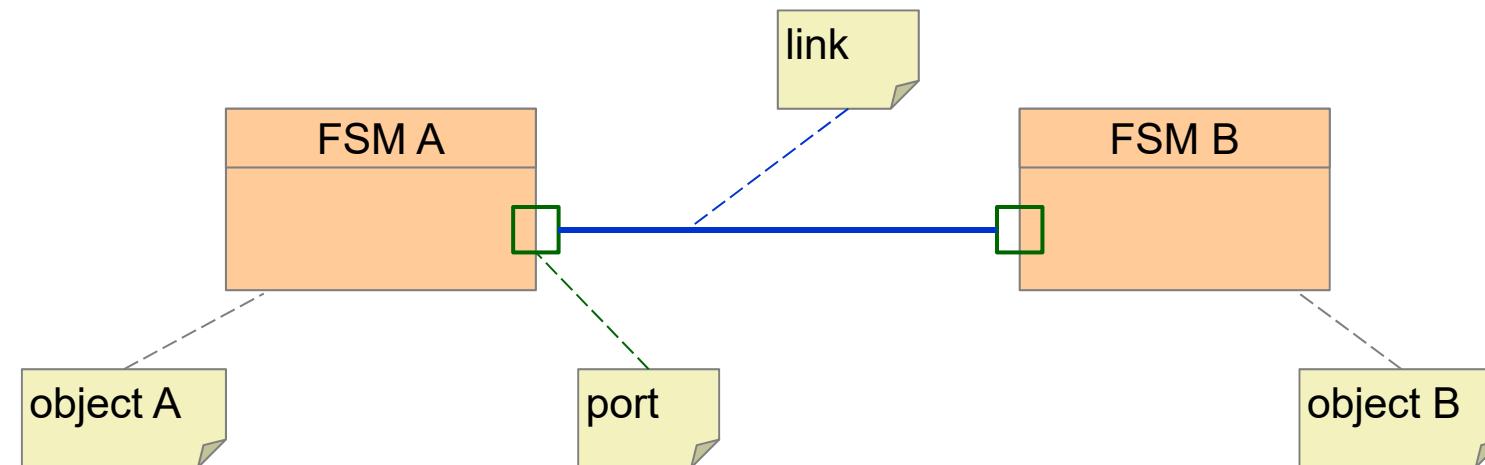


■ Port

- Defines the messages that can be sent and received by an FSM
 - Output message → action of the FSM
 - Input message → event of the FSM

■ Link

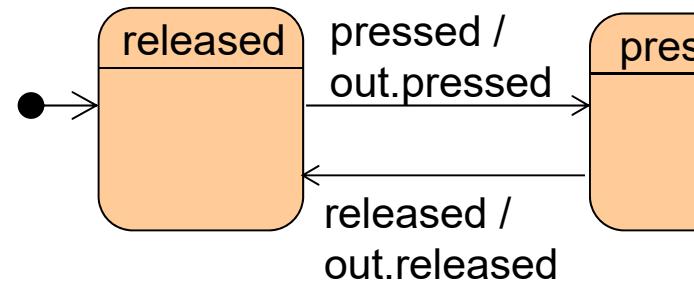
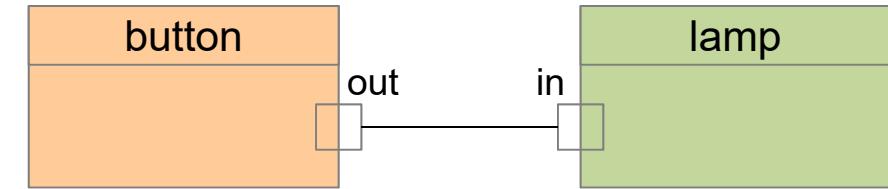
- Defines a connection for sending messages



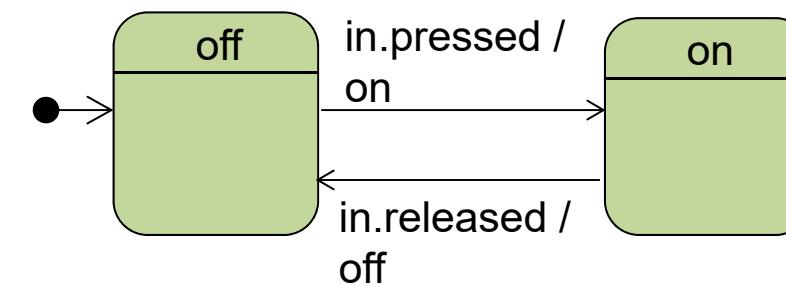
Interaction of FSMs

■ Example

- Reactive system partitioned into two FSMs
- Interaction of FSMs happens through event-messages



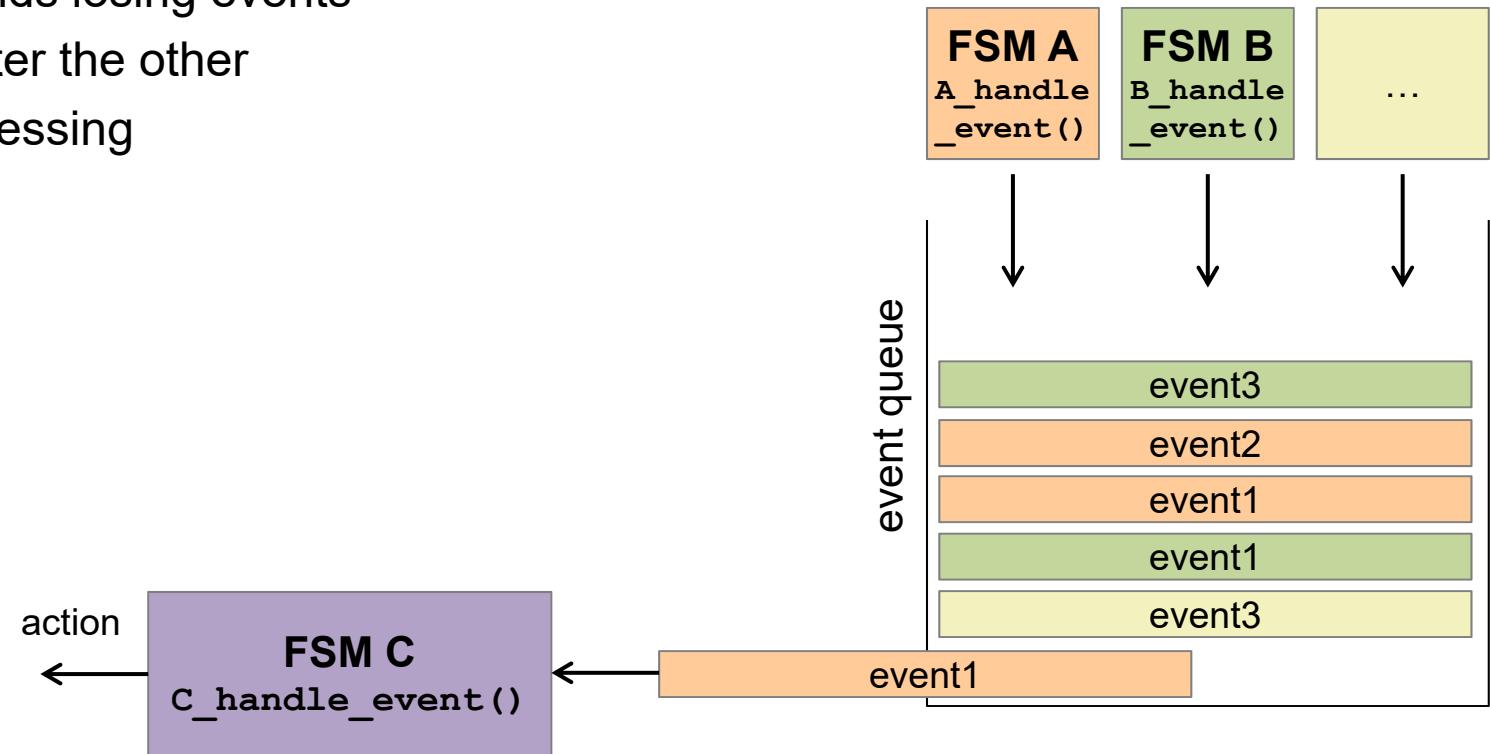
button produces
messages on port out



lamp reacts to events
on port in

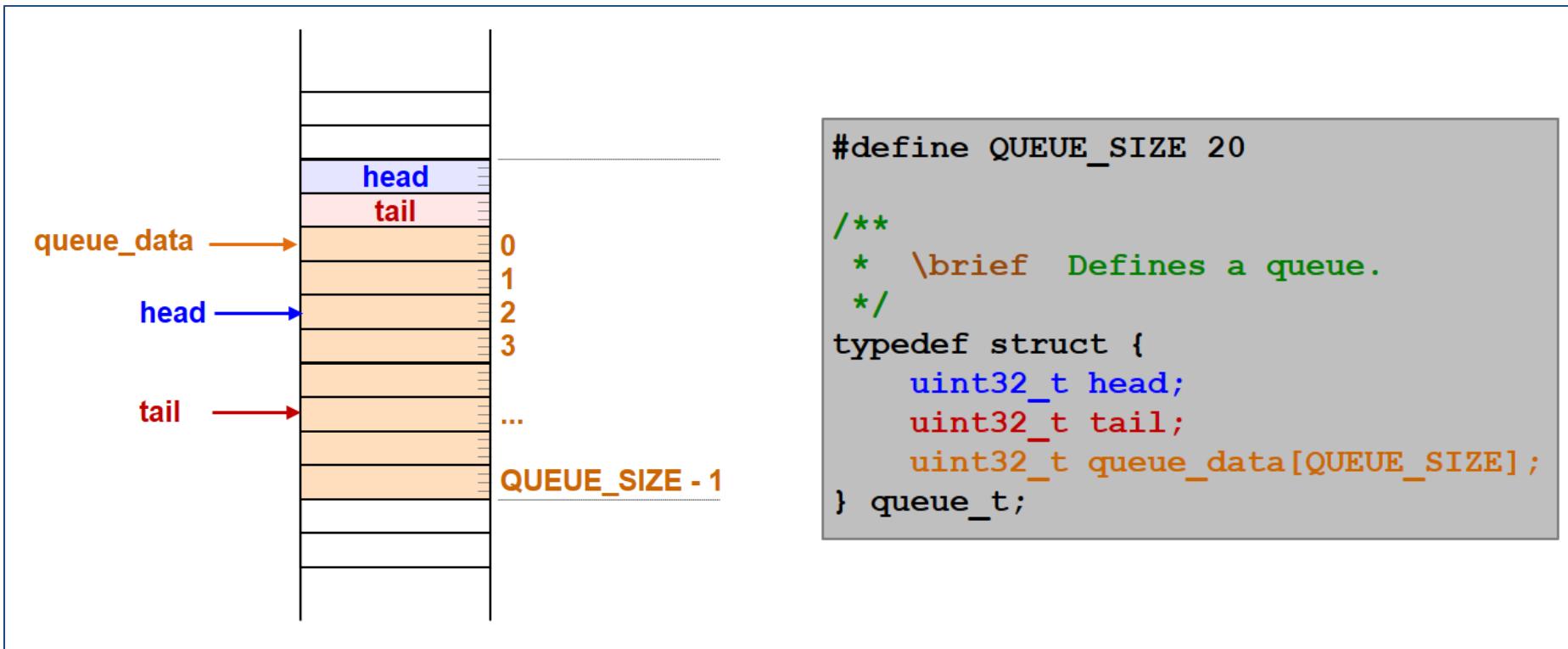
■ Event queue

- Collect events generated by different objects
- Buffered in event queue: Avoids losing events
- FSM processes one event after the other
- Events are deleted after processing



■ Event queues

- Covered in Microcomputer Systems 1 (MC1)



■ Finite State Machine (FSM)

- Allows modeling of reactive systems
- Hardware vs. software
 - Digital logic → clock driven
 - Software → event driven
- Modeling in UML
 - State / Event / Transition / Action
- Implementation in C
 - Switch case
- Interaction of FSMs
 - (Output-) actions of one FSM become (input-) events for other FSM
 - Event queue

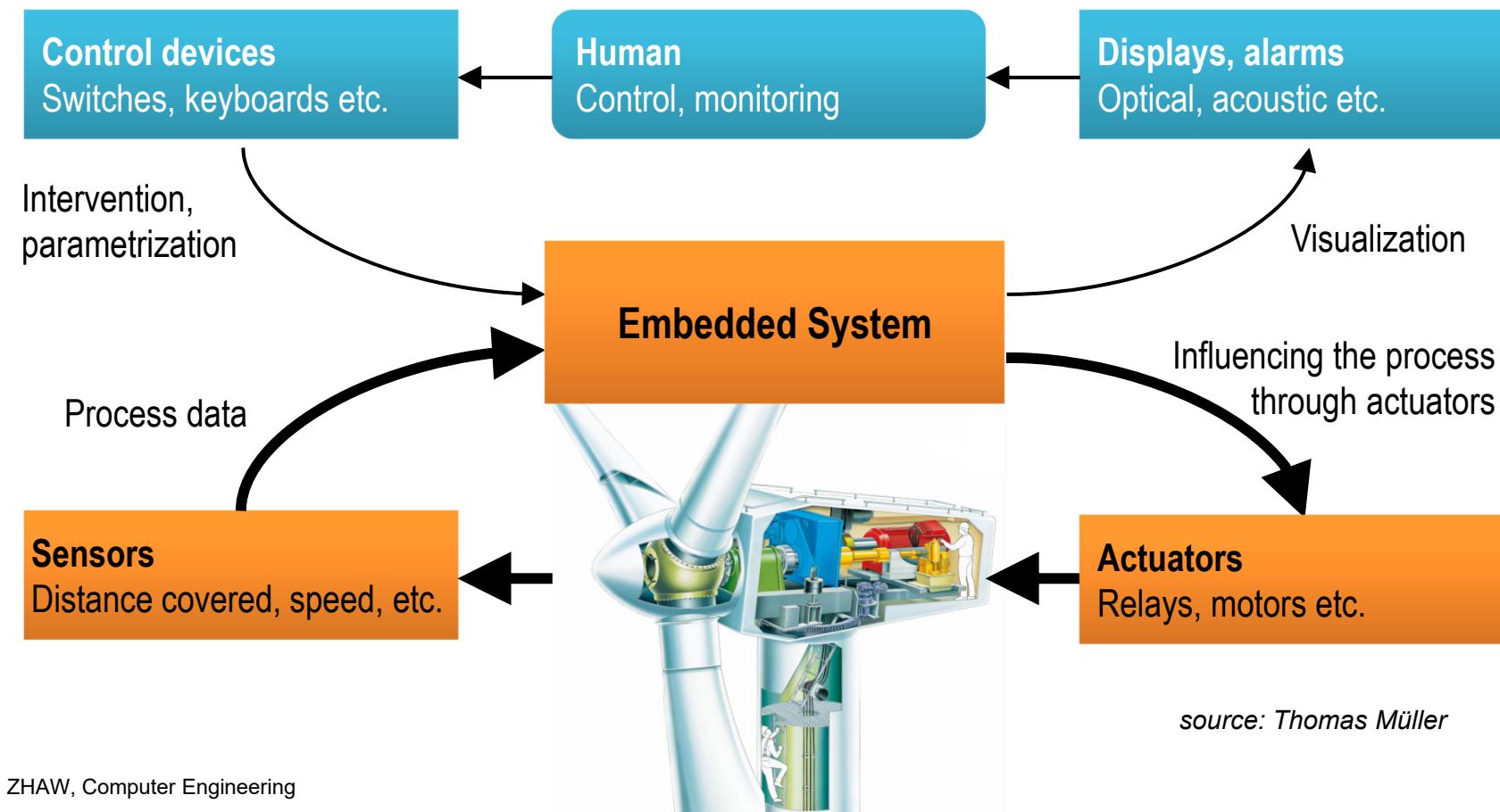
```
switch (state) {  
    case STATE_A:  
        switch (event) {  
            case S0:  
                action();  
                state = NEW_STATE;  
                break;  
            default:  
                state = WAIT;  
        }  
        break;  
  
    case STATE_B:  
        switch (event){
```

Detecting Events – Interrupt Performance

Computer Engineering 2

■ How do you recognize events?

- Cyclic queries or interrupt-driven?



Agenda

- **Polling**
- **Interrupt Driven I/O**
- **Interrupt Performance**
- **Interrupt Latency**
- **Managing Latency**
- **Interrupt Driven FSM**

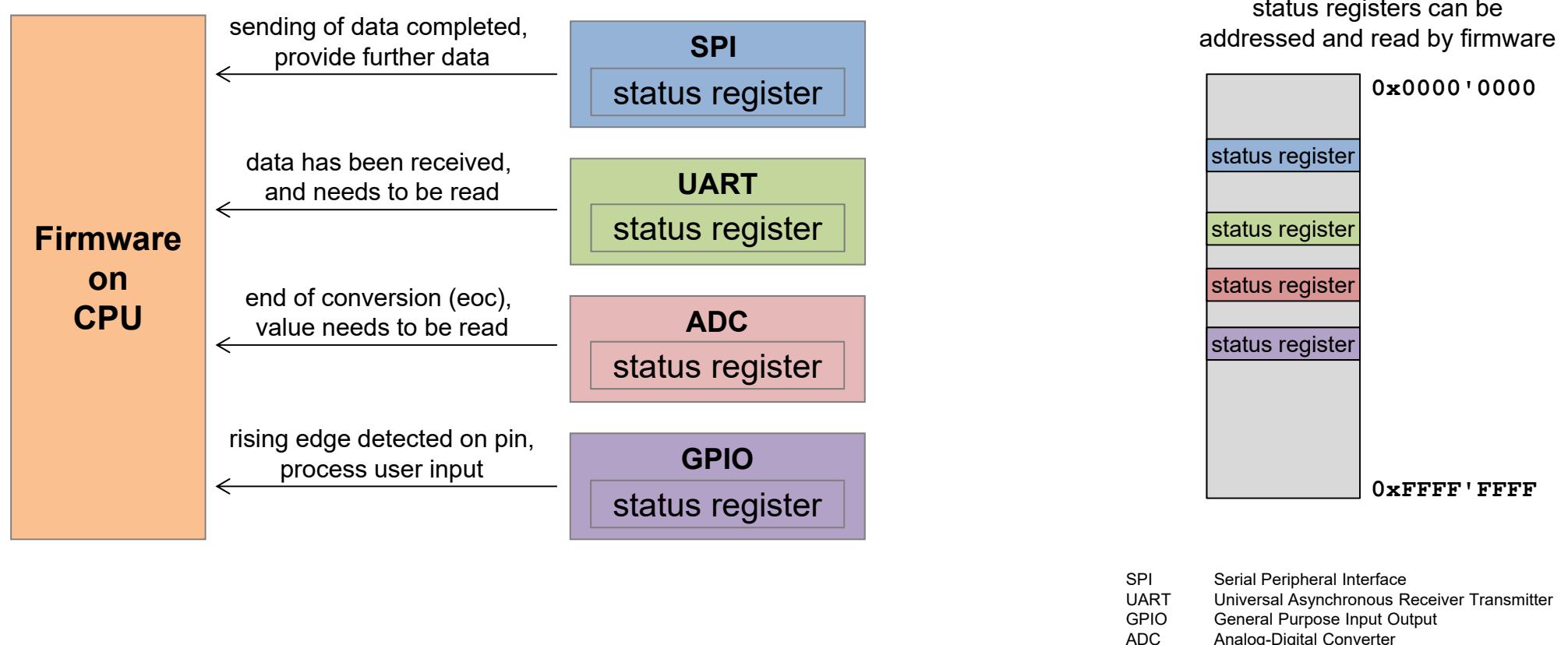
At the end of this lesson you will be able

- to explain the methods "Polling" and "Interrupt Driven I/O"
- to name important factors for the two methods
- to enumerate advantages and disadvantages of the methods
- to evaluate for a given situation whether polling or interrupt driven I/O is appropriate
- to quantify timing aspects for interrupt driven I/O
- to comprehend the term "Interrupt Latency" and name potential sources
- to explain the term 'pre-emption'
- to understand approaches to manage interrupt latency
- to name the components and explain the structure of an interrupt driven FSM

Firmware Has to Act on Events

■ Peripherals Signal Events to Firmware

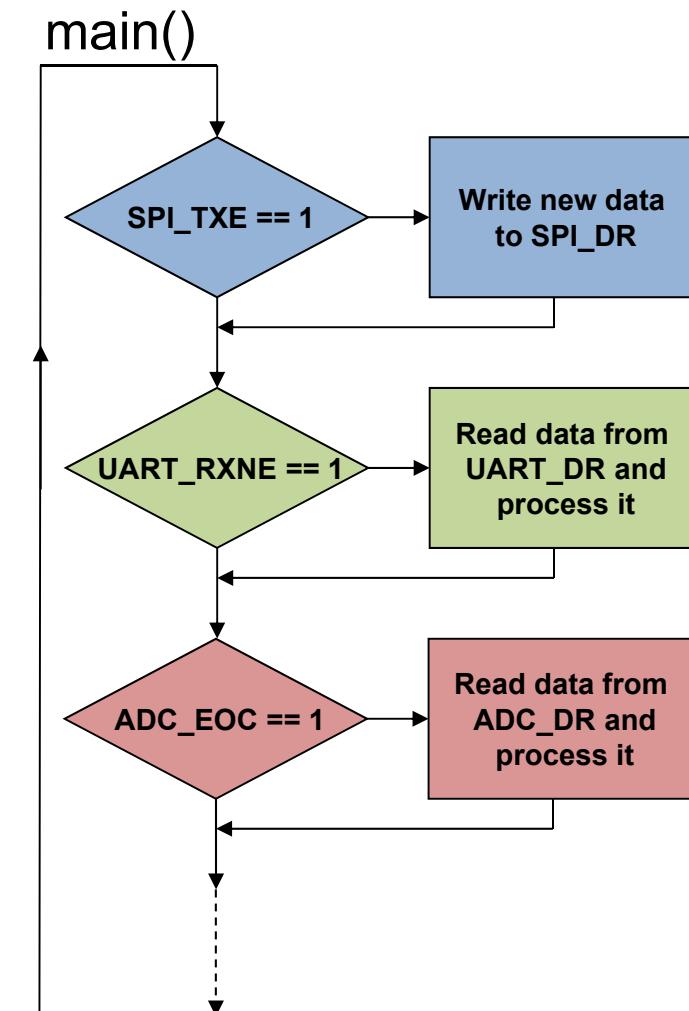
- Something happened that requires servicing in firmware



■ Periodic Query of Status Information

- Synchronous with main program
- Advantages
 - Simple and straightforward
 - Implicit synchronization
 - Deterministic
 - No additional interrupt logic required
- Disadvantages
 - Busy wait → wastes CPU time
 - Reduced throughput
 - Long reaction times
in case of many I/O devices or if the CPU is working on other tasks

¹⁾ to poll → abfragen



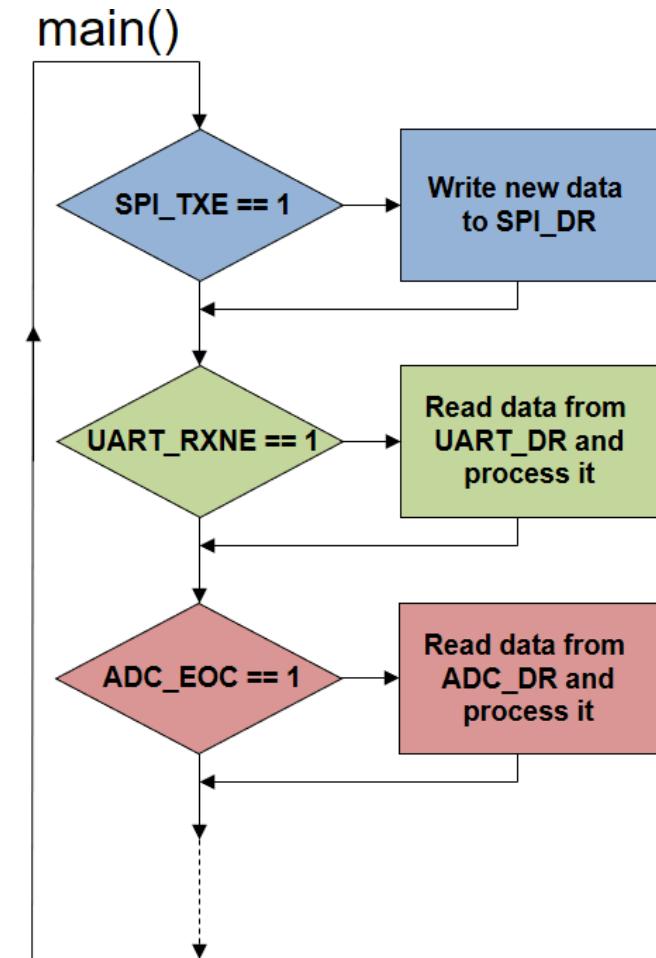
■ Implementation

```
while(1) {
    if (spi_is_txe_set()) {
        ...
        spi_write_data(...);
    }

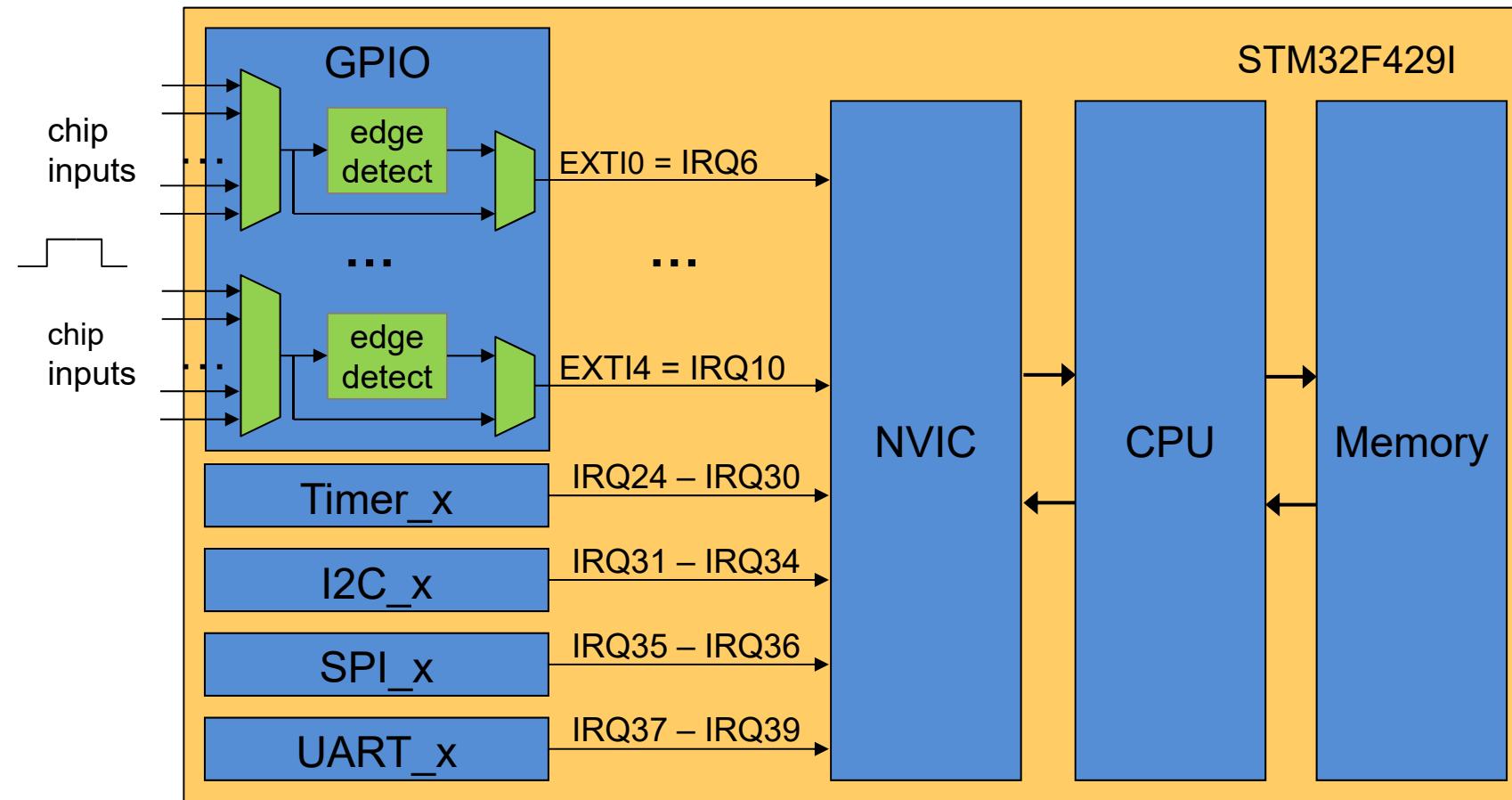
    if (uart_is_rxne_set()) {
        uart_data = uart_read_data();
        ...
    }

    if (adc_is_eoc()) {
        adc_data = adc_read_data();
        ...
    }

    ...
}
```



■ Interrupt System STM32F429I



■ Vector Table and ISR

```
; Vector Table Mapped to Address 0 at Reset
AREA    RESET, DATA, READONLY
EXPORT   __Vectors
EXPORT   __Vectors_End
EXPORT   __Vectors_Size

__Vectors      DCD      __initial_sp           ; Top of Stack
                DCD      Reset_Handler        ; Reset Handler
                DCD      NMI_Handler         ; NMI Handler
                DCD      HardFault_Handler  ; Hard Fault Handler
                DCD      MemManage_Handler  ; MPU Fault Handler
                DCD      BusFault_Handler   ; Bus Fault Handler
                DCD      UsageFault_Handler; Usage Fault Handler
                DCD      0                  ; Reserved
                DCD      0                  ; Reserved
                DCD      0                  ; Reserved
                DCD      0                  ; Reserved
                DCD      SVC_Handler        ; SVCall Handler
                DCD      DebugMon_Handler  ; Debug Monitor Handler
                DCD      0                  ; Reserved
                DCD      PendSV_Handler    ; PendSV Handler
                DCD      SysTick_Handler   ; SysTick Handler

; External Interrupts
                DCD      WWDG_IRQHandler    ; Window WatchDog
                DCD      PVD_IRQHandler     ; PVD through EXTI Line detection
                DCD      TAMP_STAMP_IRQHandler; Tamper and TimeStamps through the EXTI line
                DCD      RTC_WKUP_IRQHandler; RTC Wakeup through the EXTI line
                DCD      FLASH_IRQHandler   ; FLASH
                DCD      RCC_IRQHandler     ; RCC
                DCD      EXTI0_IRQHandler   ; EXTI Line0
                DCD      EXTI1_IRQHandler   ; EXTI Line1
                DCD      EXTI2_IRQHandler   ; EXTI Line2
                DCD      EXTI3_IRQHandler   ; EXTI Line3
```

```
/*
 * Interrupt service routines
 */
void EXTI0_IRQHandler(void)
{
    // clear flag in status register of peripheral

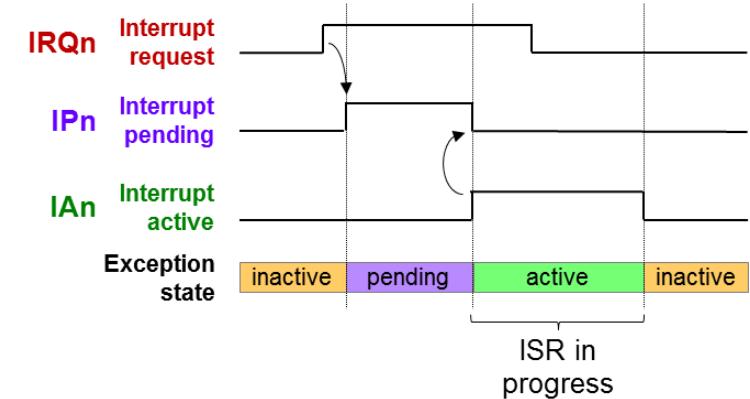
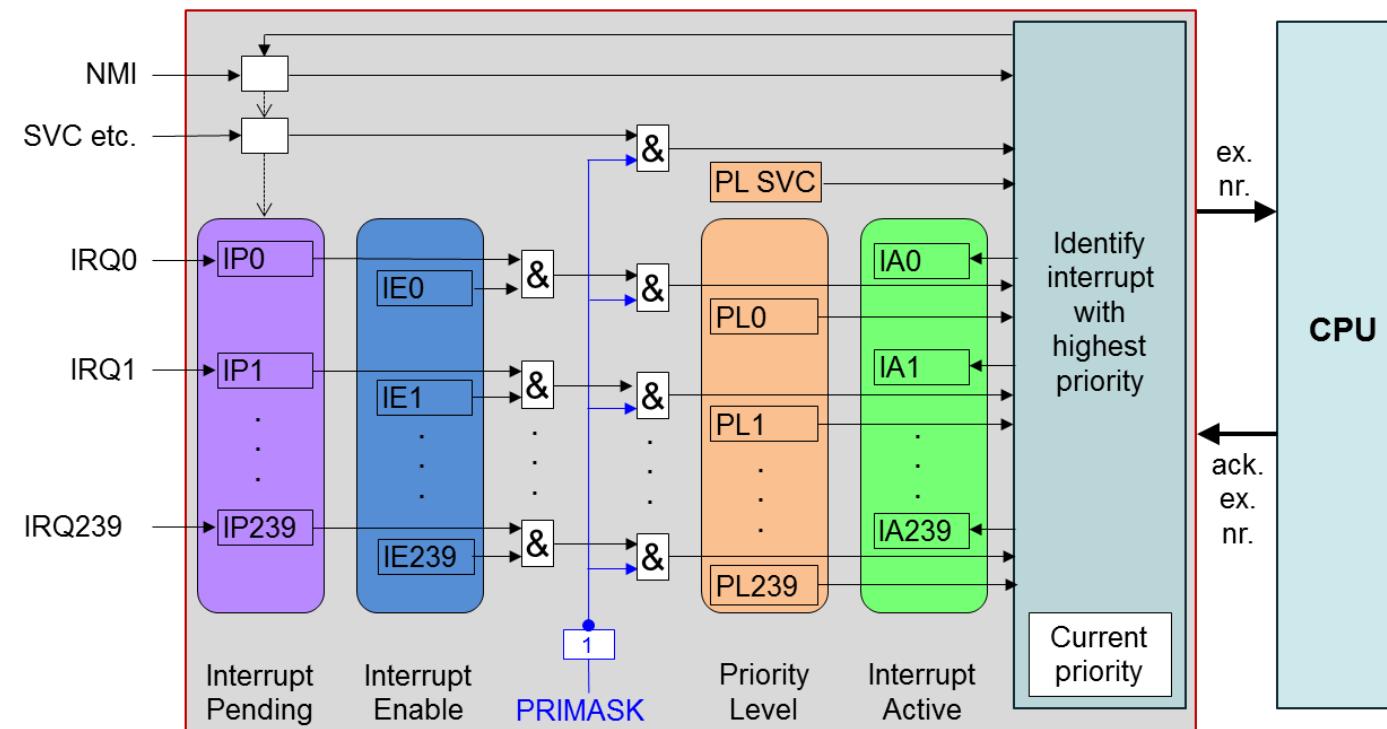
    // service interrupt
}
```

Interrupt Driven I/O

CT1 repetition

■ NVIC on Cortex-M

- Nested Vectored Interrupt Controller



- **Interrupt flags in Status Registers (SR) are hardware set and software reset**

- Software has to enable IRQxx line in peripheral

3. Event in peripheral occurs.

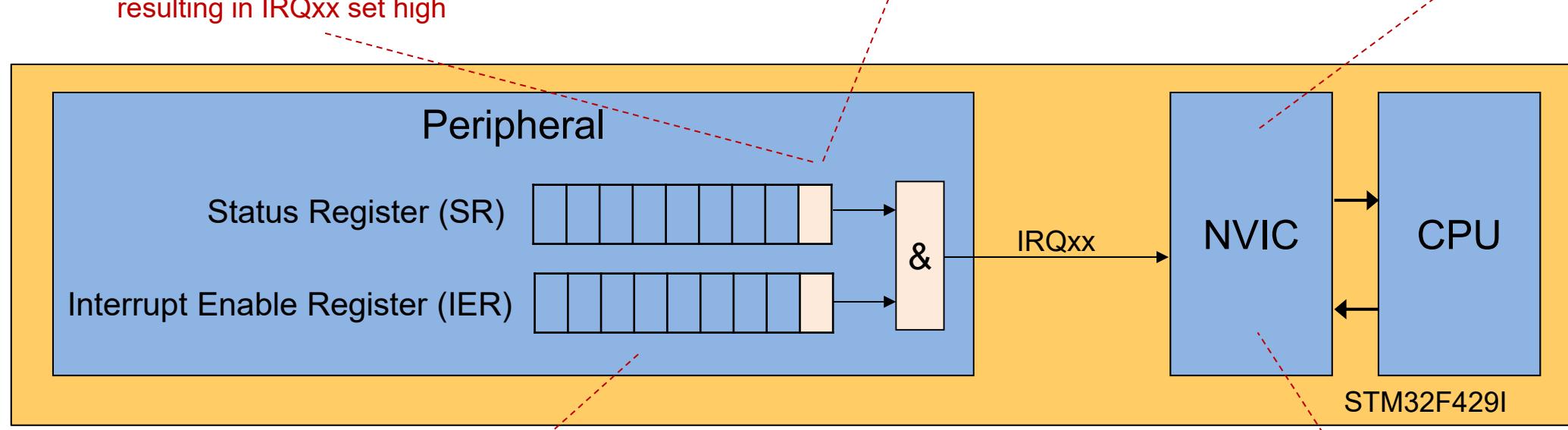
Hardware sets interrupt flag in SR
resulting in IRQxx set high

5. Software in ISR clears interrupt

flag in Peripheral → IRQxx set low

1. Software configures IRQxx in NVIC

→ Priority level, interrupt enable

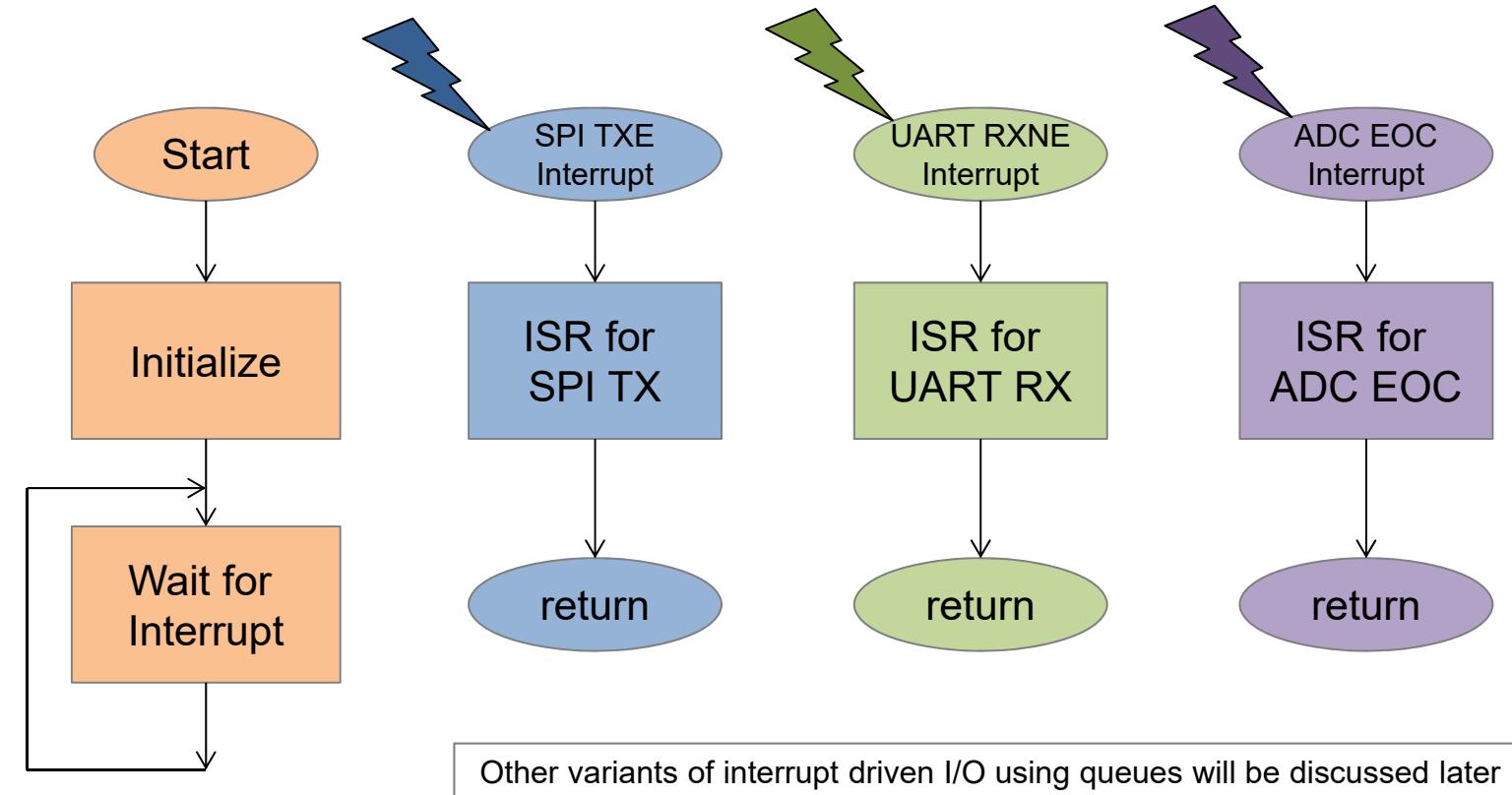


2. Software configures Peripheral and
enables IRQxx through bit in the IER

4. NVIC triggers execution of Interrupt Service
Routine (ISR) in CPU (based on vector table)

■ Interrupt driven I/O

- Extreme case: All processing done in interrupt service routines



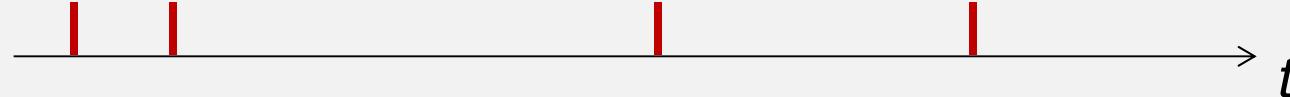


■ Interrupt frequency

- How often does an interrupt occur
- Varies from source to source, e.g.

f_{INT}

Keyboard → max. ~ 20 interrupts per second in irregular intervals



Serial interface → one interrupt every 8 bit
e.g. 230'400 baud $230'400 / 8 = 28'800$ interrupts per second

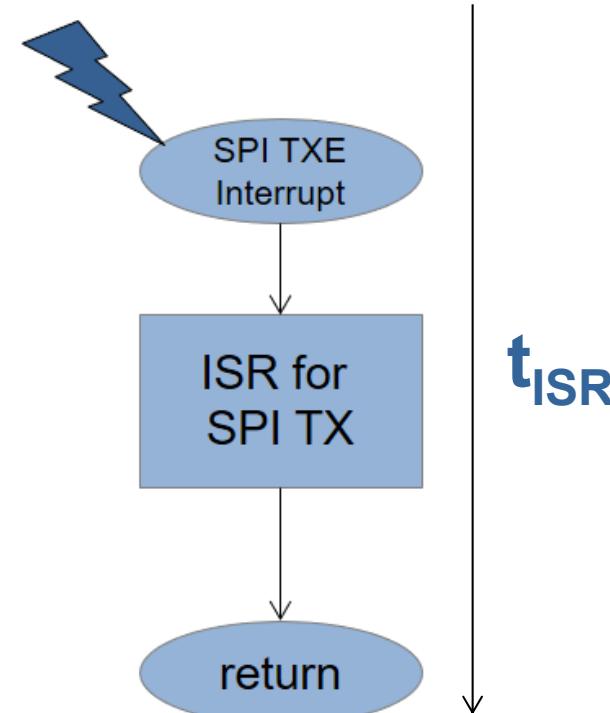




■ Interrupt service time

t_{ISR}

- Required time to process an interrupt
 - i.e. execution time of ISR
- Depends on
 - Number of instructions in ISR
 - Required number of clock cycles per instruction
→ depends on CPU architecture
 - CPU clock frequency
 - Time for switching to and returning from ISR





■ Impact on system performance

- Percentage of CPU time used to service interrupts

$$\text{Impact} = f_{\text{INT}} * t_{\text{ISR}} * 100 \%$$

- Example keyboard

$$f_{\text{INT}} = 20 \text{ Hz} = 20 \frac{1}{s}$$

$$t_{\text{ISR}} = 6 \text{ us}^1)$$

$$\text{Impact} = 20 \text{ Hz} * 6 \text{ us} * 100 \% = 0.012 \%$$

- Example serial interface with 230'400 Baud

$$f_{\text{INT}} = 230'400 / 8 = 28'800 \text{ Hz}$$

$$t_{\text{ISR}} = 6 \text{ us}^1)$$

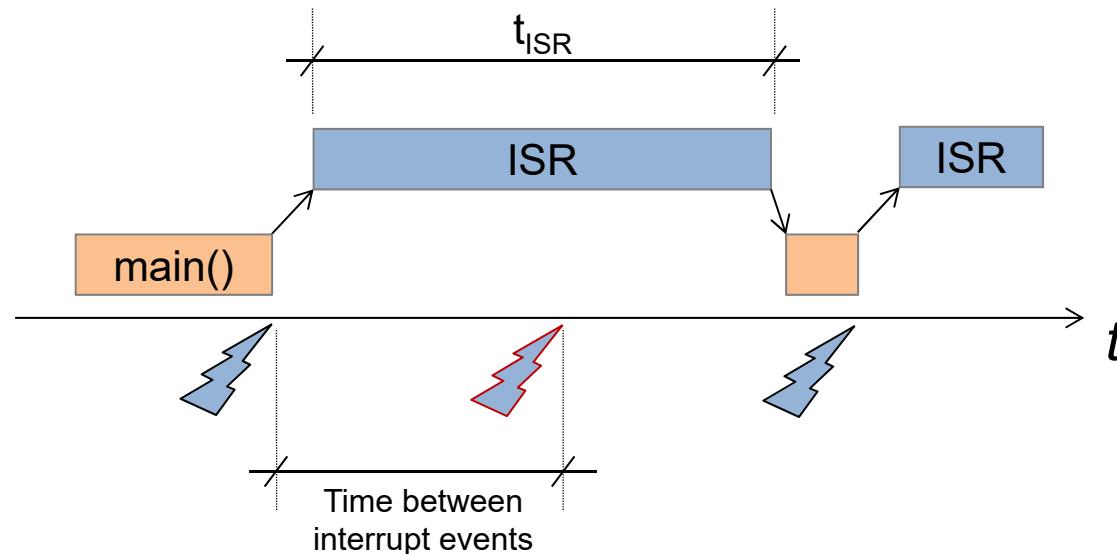
$$\text{Impact} = 28'800 \text{ Hz} * 6 \text{ us} * 100 \% = 17.3 \%$$

¹⁾ = assumed value: 6 us ≈ 1000 cycles @ 168 MHz on CT Board



- **$t_{ISR} > \text{"Time between two interrupt events"}$**

- Some interrupt events will not be serviced (lost)
 - Data will be lost
- f_{INT} as well as t_{ISR} may vary over time
 - Average may be ok, but individual interrupt events may still be lost



Caution in case of several interrupt sources

- Interrupts can occur simultaneously
- Computing power required for both ISRs

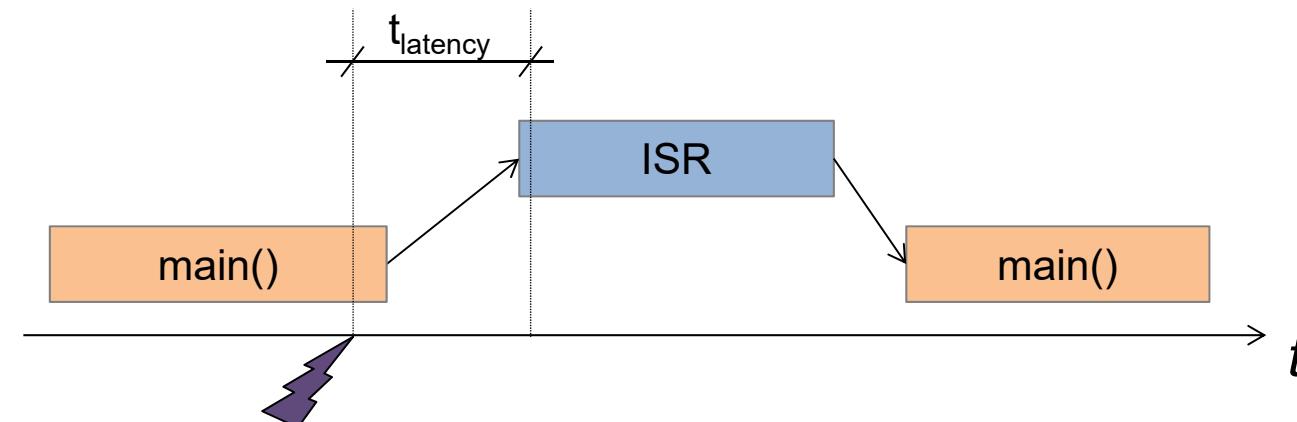


■ Strive for short ISRs

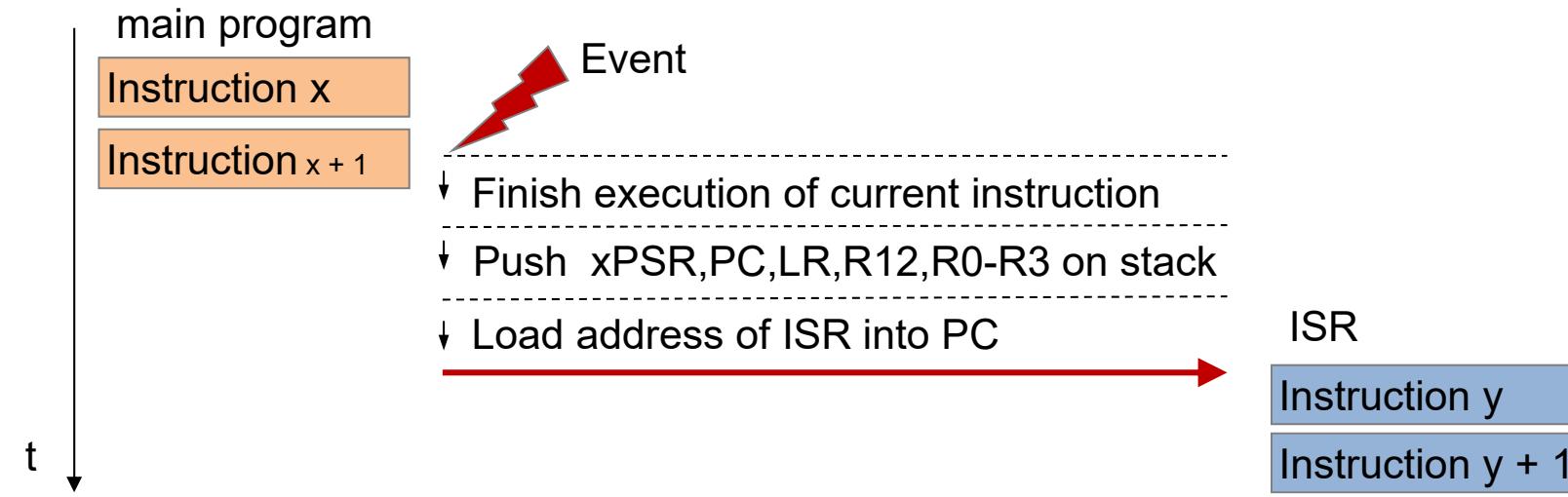
- Simpler debugging
- Execute only time-critical tasks in ISR
 - Move tasks with relaxed time-constraints to main loop
 - Makes ISR available for other time-critical tasks
 - Often simply feed interrupt to queue; dispatch queue in main loop

■ Interrupt latency definition

- Time between interrupt event and start of servicing by ISR
 - How long does it take until first 'useful' instruction in ISR is executed
- Range
 - From about 50 nanoseconds (ns) up to several milliseconds (ms)
- Relevant in cases where guaranteed service times are required
 - E.g. audio/video streaming



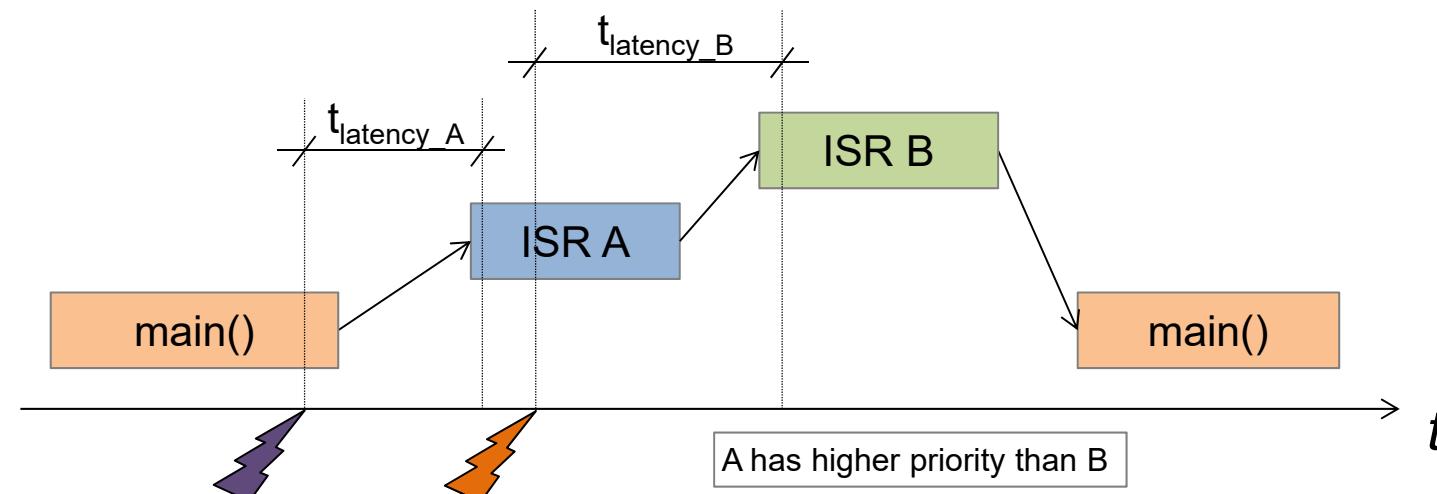
■ Latency influenced by hardware (CPU)



- Different instructions may have different execution times
- Multi-cycle instructions on Cortex-M3/M4
 - Some (e.g. SDIV/UDIV) are abandoned and restarted after ISR
 - Some (e.g. LDM/STM) are interrupted and resumed after ISR

■ Latency influenced by software (code)

- Saving additional registers on stack
- Process ongoing or higher prioritized ISRs
- Masked (disabled) interrupts → CPSID i / CPSIE i
- In case several sources are using the same interrupt line
 - SW has to use polling to know which peripheral requires servicing



- **Required response time for a specific event**
 - Certain peripherals require fast response (< 1us)
 - Real-time systems, e.g. anti-lock braking system
 - Cannot be guaranteed if maximum interrupt latency is too high
- **f_{INT} too high → Too many interrupts**
 - No CPU cycles left for data processing
 - System is busy calling ISRs
 - Neither ISR nor main loop can process any data
 - Performance of CPU is used only for latencies / context switching

■ Fast response times – low latency

- Fast CPU
 - High clock rate
 - Low number of clock cycles per instruction
- Extremely short polling loops can be fast
- Pre-emption with appropriate priorities
 - Priority levels programmed in NVIC
 - Real Time OS

Priority Level Registers for
IRQ0 – IRQ239



source: Techopedia Inc.



Definition - What does *Pre-Emption* mean?

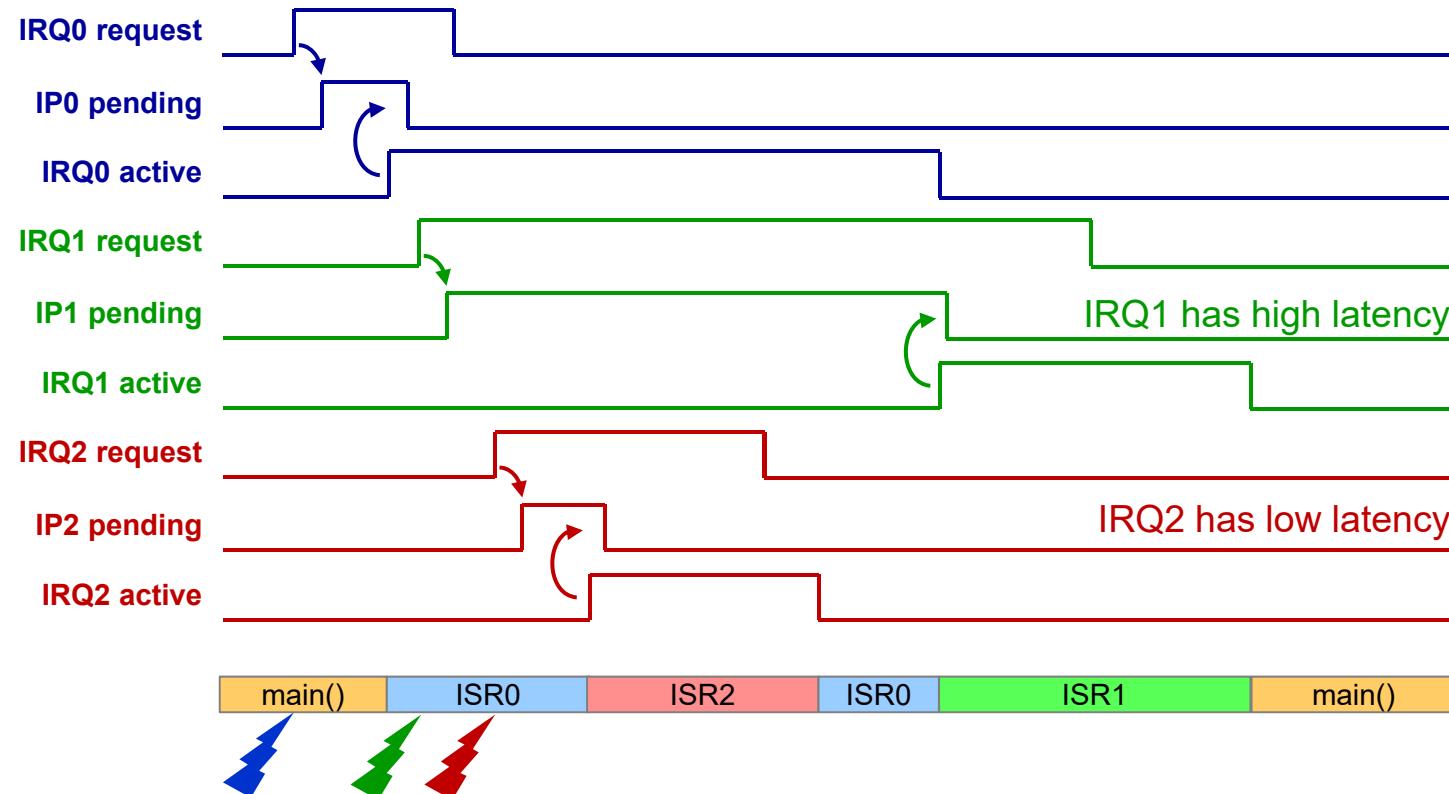
Pre-emption refers to the temporary interruption and suspension of a task, without asking for its cooperation, with the intention to resume that task later. This act is called a context switch and is typically performed by the pre-emptive scheduler, a component in the operating system authorized to pre-empt, or interrupt, and later resume tasks running in the system.

■ Example interrupt priorities

- ISR1 does not pre-empt ISR0
- ISR2 pre-empts ISR0

assuming

IRQ0	PL0 = 0x2	medium priority
IRQ1	PL1 = 0x3	lowest priority
IRQ2	PL2 = 0x1	highest priority

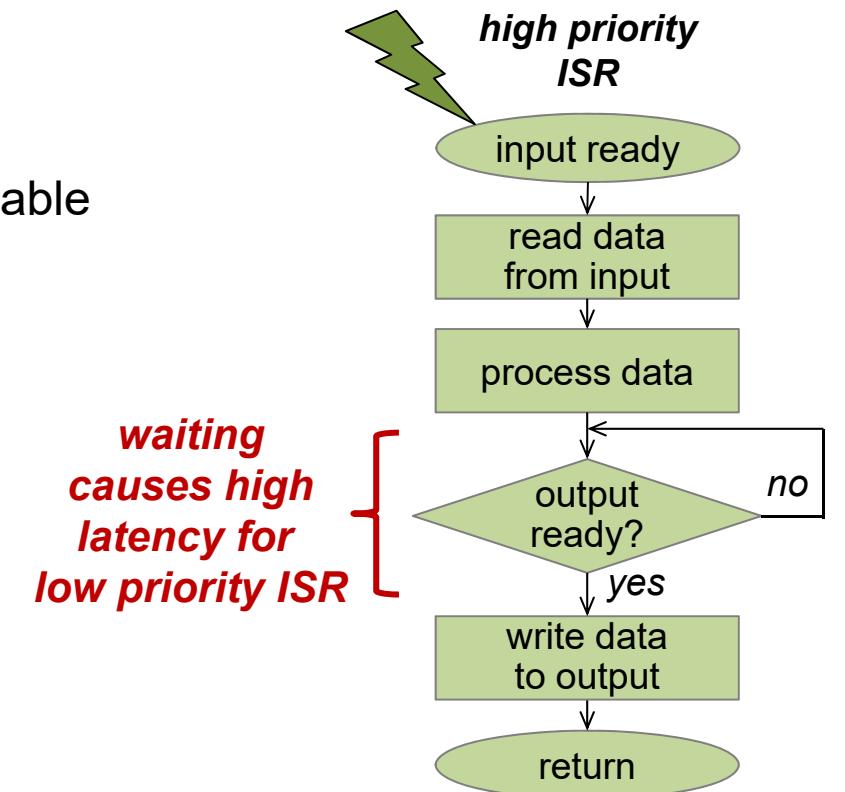


■ **Latency consistency**

- Some applications can tolerate considerable interrupt latency as long as it is consistent
 - I.e. same amount of latency from one interrupt event to next one
- Example: Measurements with periodic time intervals
 - Path measurements to detect whether an object is being accelerated, e.g. counting of step pulses on an incremental position encoder for a motor
 - Works only with a constant time interval

■ High priority interrupts may cause high latencies for lower priority interrupts

- Example for high priority ISR
 - Interrupt triggers reading of input data
 - Process data
 - Write to output
- Writing may have to wait for output device to become available
 - E.g. because SPI is still transmitting previous data



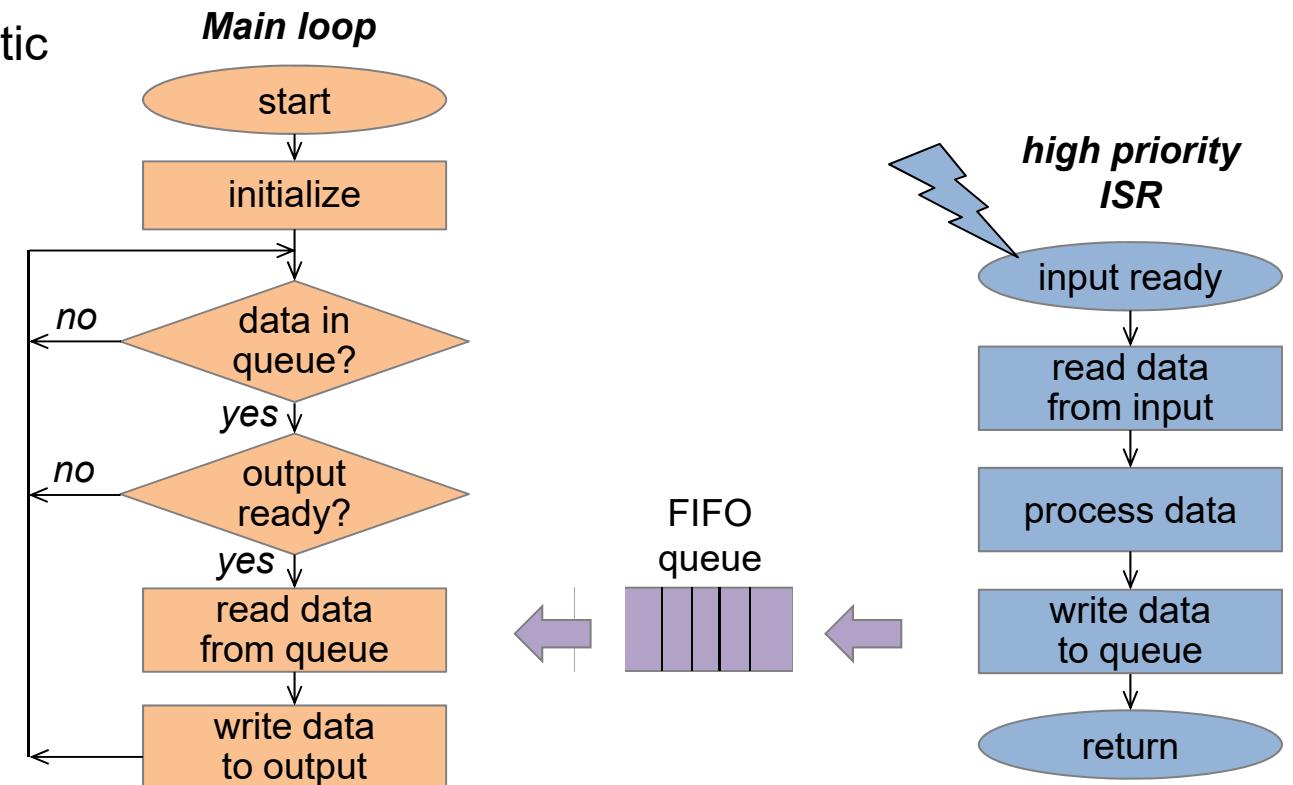
Managing Latency

■ Remedy: Move 'waiting loop' to main program

- Use queue (FIFO) for decoupling
- Makes input ISR short and deterministic

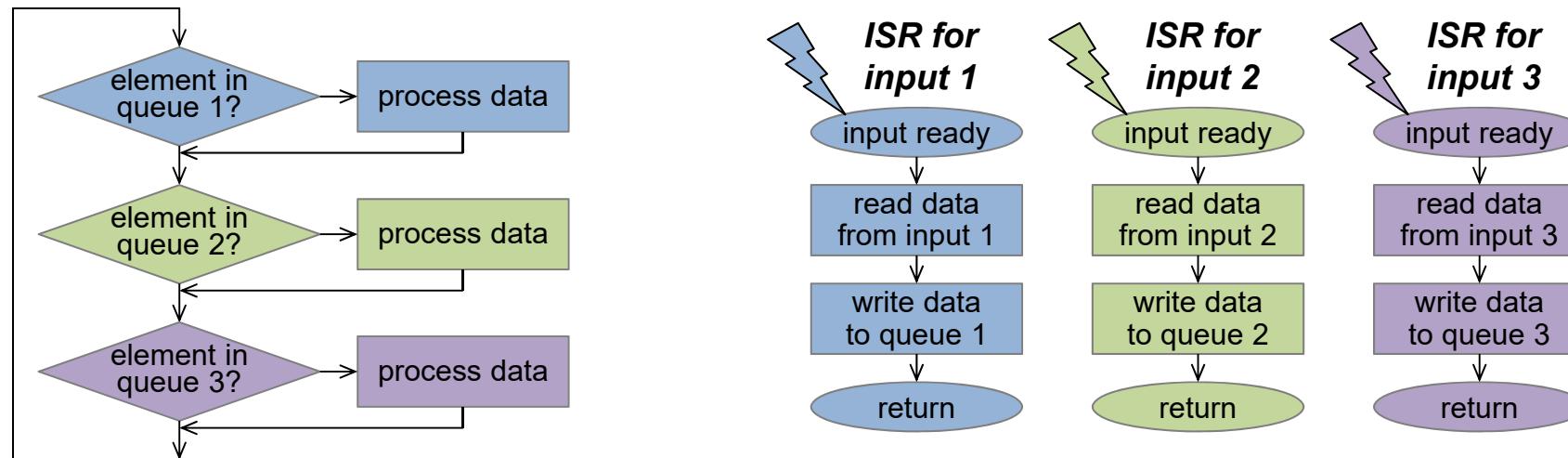
Example for functions to enter and retrieve data from a queue

```
/**\n * \brief Enqueues data at the tail of the specified queue\n * \param queue Points to the queue to which the element should be added\n * \param data\n * \return 0 if data NOT added (queue is full),\n * ~0 if data added (queue not full)\n */\nuint32_t queue_enqueue(queue_t *queue, uint32_t data);\n\n/**\n * \brief Grabs and removes the element at the head of the specified queue\n * \param queue Points to the queue from which the element should be polled\n * \return element at the head of queue, returns 0 if no element in queue.\n */\nuint32_t queue_dequeue(queue_t *queue);
```



■ Move non-time-critical work from ISRs to main loop

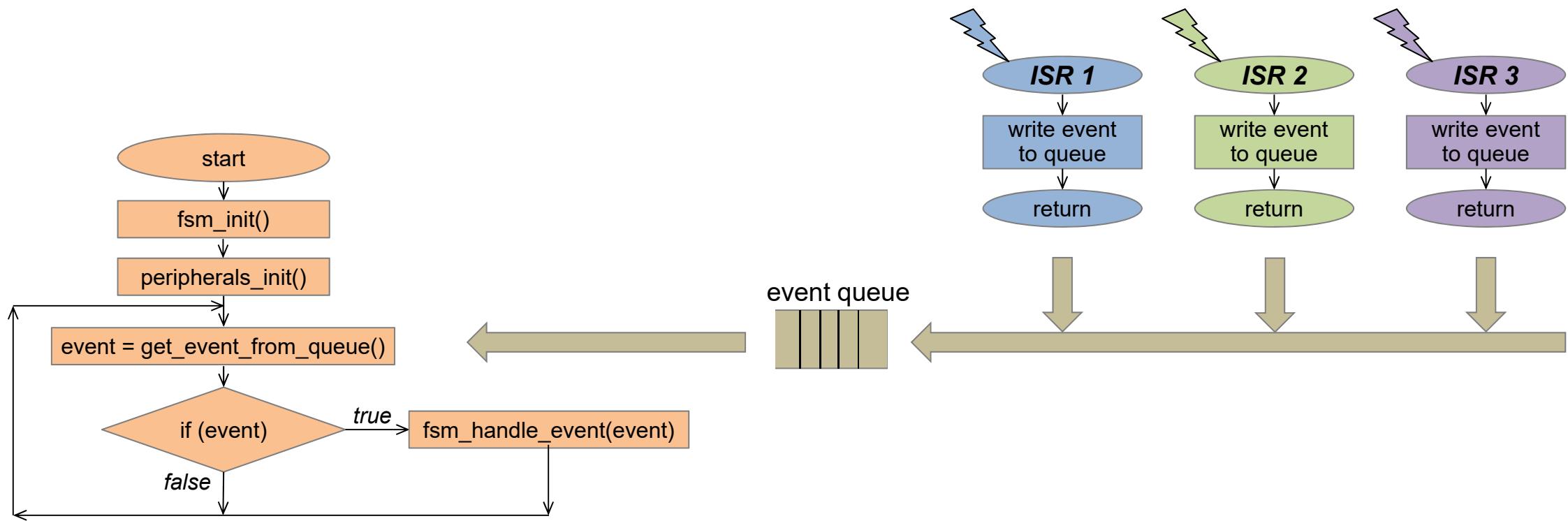
- Several ISRs write input data into their dedicated queues
- Main loop checks all queues and processes their contents
- May result in many tasks in main loop
 - Not all processing tasks have the same priority
 - Requires scheduling of tasks
 - May require real-time OS in case of many ISRs



Interrupt Driven FSM

Events Trigger Interrupts

- Several Interrupt Service Routines (ISR) enter events into queue
- FSM in main loop reads events from queue

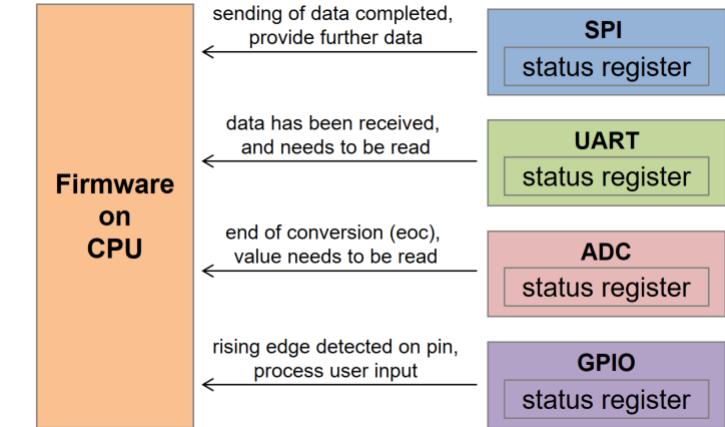


Conclusions

■ Detection of Events → Polling vs. Interrupt Driven I/O

■ Interrupt Performance

- Interrupt frequency f_{INT}
- Interrupt service time t_{ISR}
- Impact = $f_{INT} * t_{ISR} * 100\%$

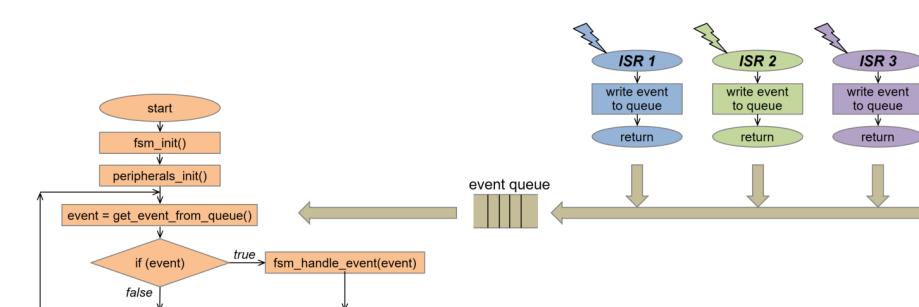


■ Interrupt Latency

- How fast does an application require a response?
- Pre-emption: High priority interrupts may cause high latencies for lower priority interrupt
- Move 'waiting loops' and non-time-critical work from ISR to main loop

■ Interrupt Driven FSM

- Interrupt Service Routines (ISR) enter events into queue
- FSM in main loop reads events from queue



■ Interrupts In General

- "The Art of Assembly Language Programming"
 - by Randall Hyde, Chapter 17
- "Fundamentals of Embedded Software with the ARM Cortex-M3"
 - by Daniel W. Lewis, Chapter 9

■ Interrupt Latency

- <http://www.segger.com/interrupt-latency.html>
- <http://www.ganssle.com/articles/interruptlatency.htm>