Joint ICTP, SAIFR and UNESP School on Systems-on-Chip, Embedded Microcontrollers and their Applications in Research and Industry

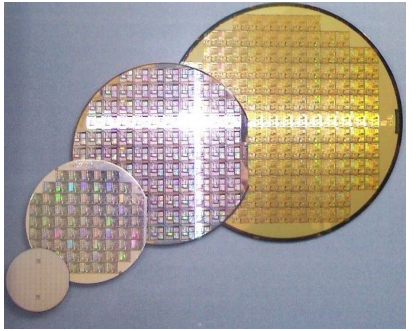
Microcontroller technology, architecture and peripherals

Dr. Luigi Calligaris (SPRACE/UNESP)

Day 1 - 18/10/2021

Brief prologue: semiconductor devices

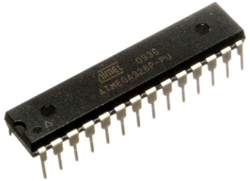
- Semiconductor devices based mainly on Si technologies
  - Well-established industry with decades of experience
  - Many chip designers + a number of huge and smaller factories
- Common production process in brief
  - Quartz sand → remove oxygen → Si + impurities
  - Purify Si → long Si monocrystal ingots → cut into wafer
  - Process wafers to build components & circuits → dice into **chips**
- Making **smaller chips** has many advantages
  - Increase #chips/wafer and % yield → lower cost for same HW
  - Miniaturization (think about wearable or injectable electronics)
  - Thermal, power and frequency performance…
- Use devices with **just what you need → save $$$, space, pwr**
  - Large variety of devices tries to match with application needs



General-purpose processor devices

AMD Epyc: a server-oriented CPU

BCM2711B0 in an RPi4



ATMega328P: basis of ArduinoUNO

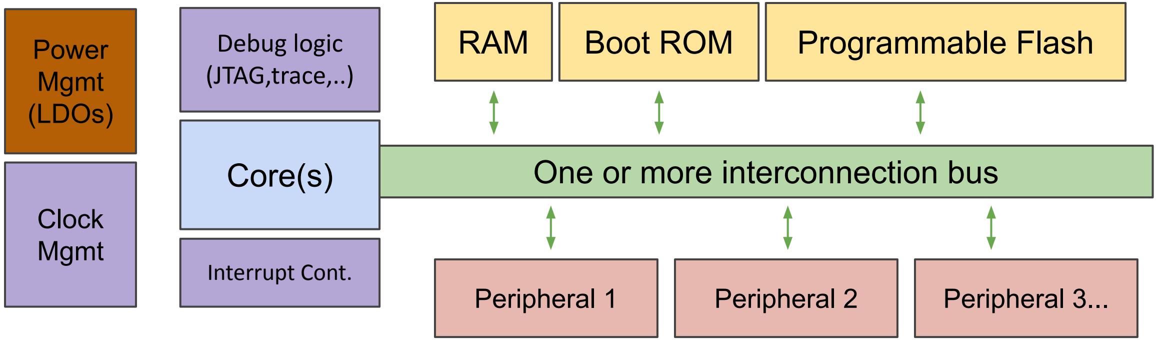
Standalone CPU	**System on a Chip (SoC)**	**Micro Controller Unit (MCU)**
Computers & servers	Complex embedded systems	Simple embedded systems
Depends on external devices (usually on motherboard/cards)	Most of functions in the package (usually bulky DRAM sits outside)	Every function tries to be in the same package (RAM included)
Large processing power	Wide variety, usually mid-powered	Small-medium processing power
Designed to run an OS	Often OS, RTOS or bare-metal	Only bare-metal or RTOS

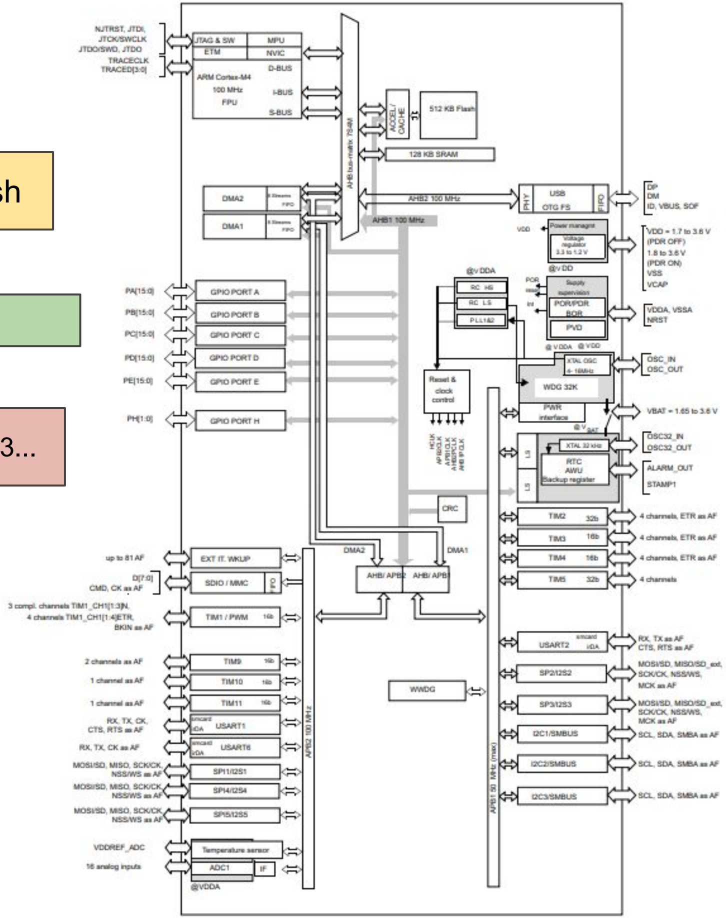
→ low-power, efficiency, small

high-power, performance, big ←

- NOTE: market variety is blurring these definitions
- In this school Dr. Christian Sisterna will present extensively SoCs with programmable FPGA logic

Microcontroller structure





A real example: STM32F411

- Core, clocks, debug, interrupt, power
- Memories (Flash, RAM, boot ROM)
- Peripherals
  - Accessory functions (crypto unit, DMA, …)
  - Interface with the external world (GPIO, I2C, SPI …)

Fundamental components

MCU Core

- It’s where your compiled application code runs
  - Orchestrates the operation of the various MCU peripherals
  - Defining aspect of a microcontroller product family
  - In MCUs, often instructions **can be executed straight from flash memory**
- There are hundreds of different cores available for use
  - 8-, 16- or 32-bit address designs
- Different architectures are used (even by the same company)
  - Von Neumann → instructions and data use the same memory
  - Harvard → strict separation of instruction and data memory spaces
  - Mod. Harvard → various compromises in between (most common)
  - These distinctions can be important, e.g when estimating memory use
- Note: MCU companies often outsource core design to IP providers
  - e.g. XTensa (Tensilica), 68000 (Motorola), **Cortex-M (ARM)**, **SiFive**
  - The design is customized to the needs of the MCU designer
  - Some documentation you need may be on the IP provider site

Tensilica designed the L106/LX6 used by ESP8266/ESP32

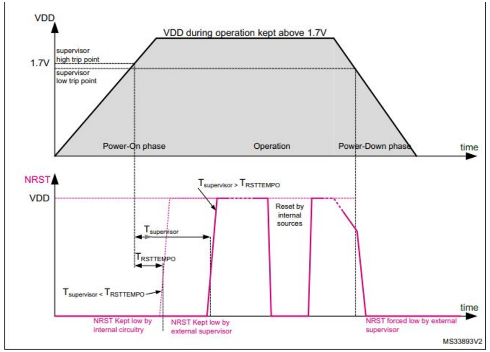
ARM Cortex-M are the most common mid-high performance MCU cores used today

SiFive licenses its RISC-V core IPs (or can design entire chips)

The CIP-51 designed in-house by SiLabs derives from Intel is the basis of the BusyBee MCUs

Nordic & then Atmel developed in-house the AVR cores, basis of e.g. ATMega328P of Arduino

Reset/boot logic & boot ROM

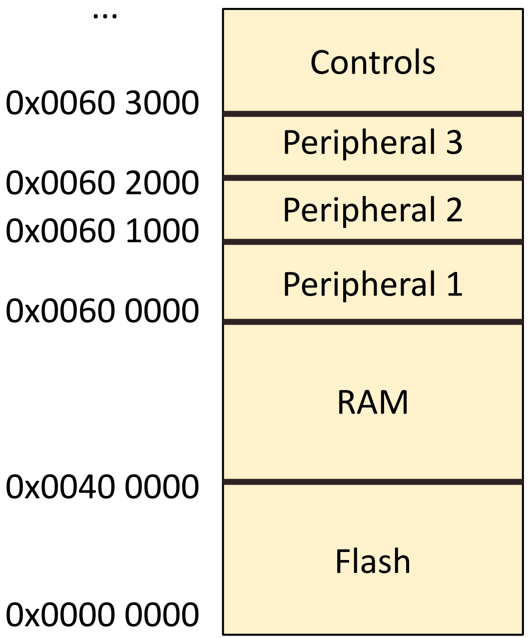


PoR timing diagram for an STM32F4 MCU

- Reset state
  - Well-known initial state for the MCU and (some of the) peripherals
  - Power-on reset (PoR) generator → triggers reset at power-up
  - Package pin or internal register → triggers reset while running
- Usually the MCU after reset runs boot code stored in ROM during fabrication
  - This early bootloader sets up the device based on pin state & non-volatile regs
    - *Should I receive a new firmware image on UART/SPI/I2C? How?*
    - *Should I fetch the firmware in an external flash memory? How?*
    - *Which is the first execution address in flash memory?*
- The boot process is a critical element of firmware & system design
  - You can play lots of nice tricks to update firmware & load it in stages
    - The presentations by André (OpenIPMC) and Oliver (ZynqMP boot) will show some magic

A general note on controlling components

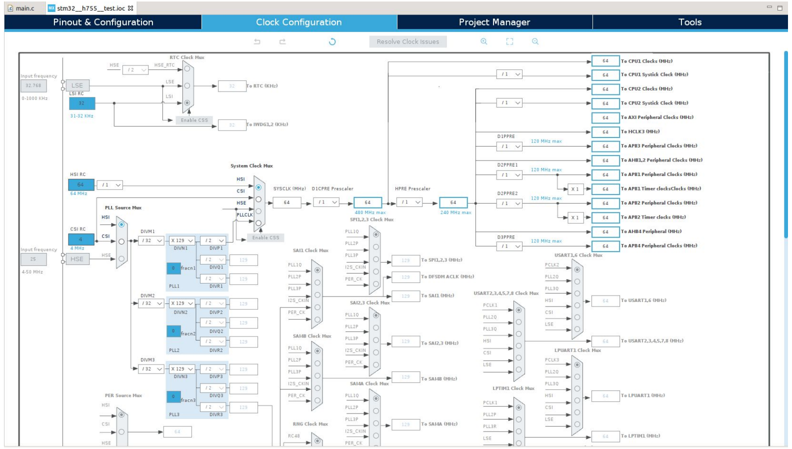
- In MCUs the logic present in the silicon is usually mapped to addresses
- Everything that you can control from your code maps to a void*
  - Flash memory
  - RAM
  - Peripherals
  - Power tree control regs
  - Clock tree control regs



- The address map is specified in datasheets
- Modern IDEs do the work for you
  - You usually don't have to deal with addresses
  - The addresses are hidden behind a C macro

Clock management

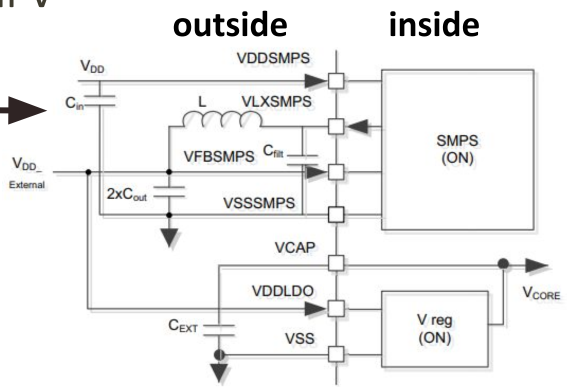
- MCUs usually use a simple & slow RC internal oscillator at startup
  - Quite imprecise, but compact and suitable for some applications
  - Clock source can be later switched to external crystal/oscillator
- Clock trees can be highly configurable
  - Use multiple external clock sources
  - Generate new (faster) clocks with a PLL
  - Clock-gating parts of the chip to save power
  - Dynamic frequency setting
  - Export clock signals outside via a pin
- Clocks usually configured via registers
  - C API provided by the manufacturer in SDK



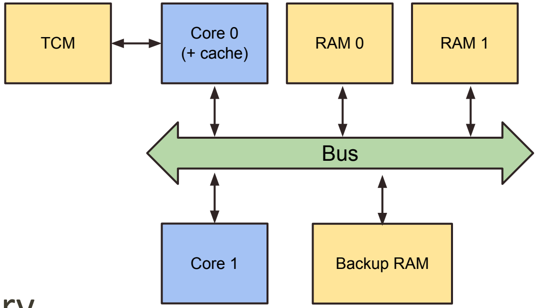
Clock configuration utility for an STM32H755 MCU

Power management

- Many MCUs split the device in multiple *power domains*
  - These regions can be powered on/off to optimize power
  - Often there will be separate powering pins for some domains
    - e.g. for noise-sensitive peripherals like ADCs
- MCUs usually come with internal voltage regulators
  - Generate core (usually very low) and peripheral internal V
  - Some MCUs can provide power to external devices
  - Some high-end MCUs come with switching mode PS →
    - More efficient and better thermal performance
- Power tree is usually controlled via registers
  - C API provided by the manufacturer in SDK



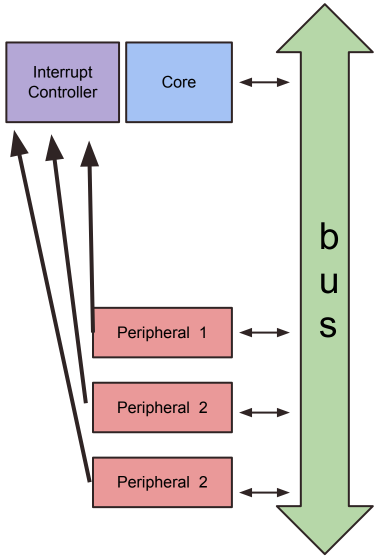
RAM



- MCUs can have one or more internal RAM banks
  - General-purpose data and instruction RAM
  - Back-up RAM → survives resets, can be powered by backup battery
  - High-performance MCUs often have
    - Low-latency RAM bound to main core (TCM, Tightly Coupled Memory)
    - Low-latency cache memory
  - Typical RAM memory sizes: from ~16 **bytes** to ~few MiB
- Few MCUs add a memory controller to use external RAM
  - Grey zone between SoC and MCU
- When designing firmware, care must be taken about RAM access:
  - In multi-core MCUs some RAM can be accessed just by specific cores / peripherals
  - RAM banks may be mapped at distinct & separate address ranges
    - Big chunks of data may not fit into the chosen bank

Interrupt controller

- Note: later there will be a class specific to this topic
- Think about timescales in an MCU
  - Core and RAM: 1’000’000 Hz - 500’000’000 Hz
  - Terminal comm: 10’000 Hz - 1’000’000 Hz
  - Button press: few Hz?
- Core wastes a lot of cycles waiting for slow events
  - It could be doing something else while waiting for the event
  - How does the core know when to go back to the old job?
- Interrupt controller
  - Gets the attention of the core when a rare event happens
  - Fundamental in making efficient use of resources



Debug and test

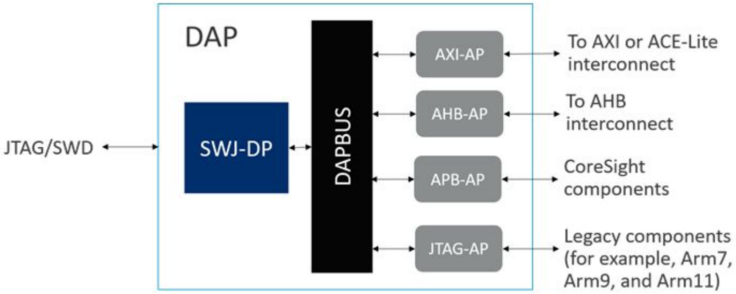
- Needs for FW development and HW production
  - Debug the execution flow in the device
    - e.g. follow step-by-step, set breakpoints, view variables…
  - Test device HW & program in-factory
    - e.g. boundary scan of the external pins, flash the product firmware

- Some MCUs: standard **JTAG Test Access Port** (TAP) controller
    - JTAG → 4/5 wires (very common, but many wires)
    - cJTAG → 2 wires (rather uncommon)

- **ARM** use proprietary CoreSight **Debug Access Port** (DAP)
    - Serial Wire Debug→ 2 wires
    - Compatible with JTAG

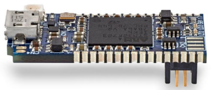
- Some devices offer TRACE ports as well
  - Sample live execution flow → code analysis & profiling

- Interface the MCU with a PC is through a USB debug adapter
  - Proprietary ones, or can be emulated in software (FTDI)

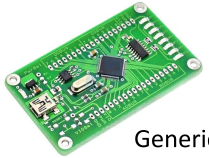



Segger JLink


Atmel AVRISP


ST STLink

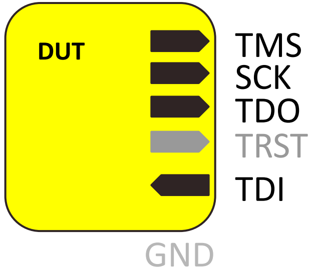

NXP MCU-Link

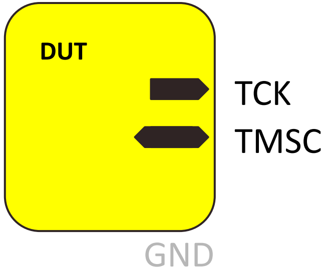

Generic FTDI board

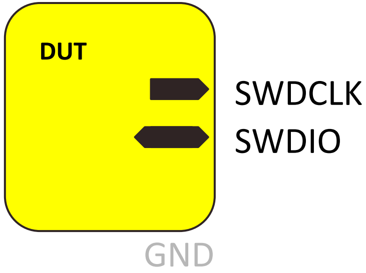
JTAG and SWD pinout

4/5-wire JTAG (IEEE 1149.1)	2-wire JTAG (IEEE 1149.7)	ARM Serial Wire Debug
TMS (Test Mode Select)	TCK (cJTAG clock)	SWDCK (SWD clock)
TCK (JTAG clock)	TMSC (cJTAG data)	TMSC (SWD data)
TDO (master-in **slave-out**)		
TDI (master-out **slave-in**)	(this is a **rare** interface)	
TRST (Test Reset) - optional		
GND	GND	GND

OTHER MANUFACTURERS HAVE SIMILAR PROPRIETARY DEBUG INTERFACES (e.g. TI Spy-Bi-Wire)







On-chip buses

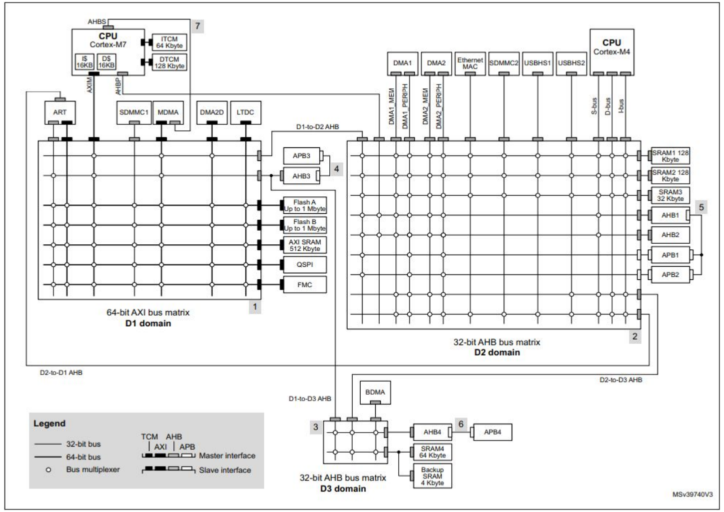
- The various components of the MCU need to communicate
  - Usually accomplished with multiple common buses + interconnects
  - On-silicon buses: optimized for low latency and high bandwidth
- Convergence towards bus standards
  - Majority of MCUs uses a few common internal bus types
  - Easier to design with **IP cores** of different manufacturers
- **Advanced Microcontroller Bus Architecture** (AMBA)
  - Freely-available and open standard by ARM

    - Advanced Peripheral Bus (APB)
    - Advanced High performance Bus (AHB)
    - Advanced Extensible Interface (AXI)
    - AXI Coherence Extensions (ACE)
    - Coherent Hub Interface (CHI)

Simple
Low bandwidth
Many peripherals

Fast & low latency
Scalable
RAM/Processors

- **All the devices you'll use in this school use AMBA buses! :)**

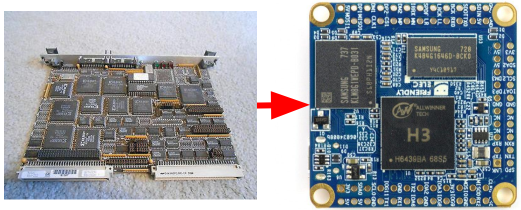


Bus topology in an STM32H745

Brief digression about on-PCB communication

Connecting devices on the PCB

- Devices get more integrated ⟶ saves space, power, $
  - Microcontrollers, SoCs, SIPs with many functions



- Some system features provided by a separate device
  - Need standard communication protocols allowing interoperability
  - One MCU peripheral to talk to a million types of different devices
- Protocols/interfaces differ based on application
  - High bandwidth (e.g. PCI-e) and/or long-range (e.g. Ethernet)
    - Complex, expensive, energy-hungry, occupy lots of PCB space

Fast & furious

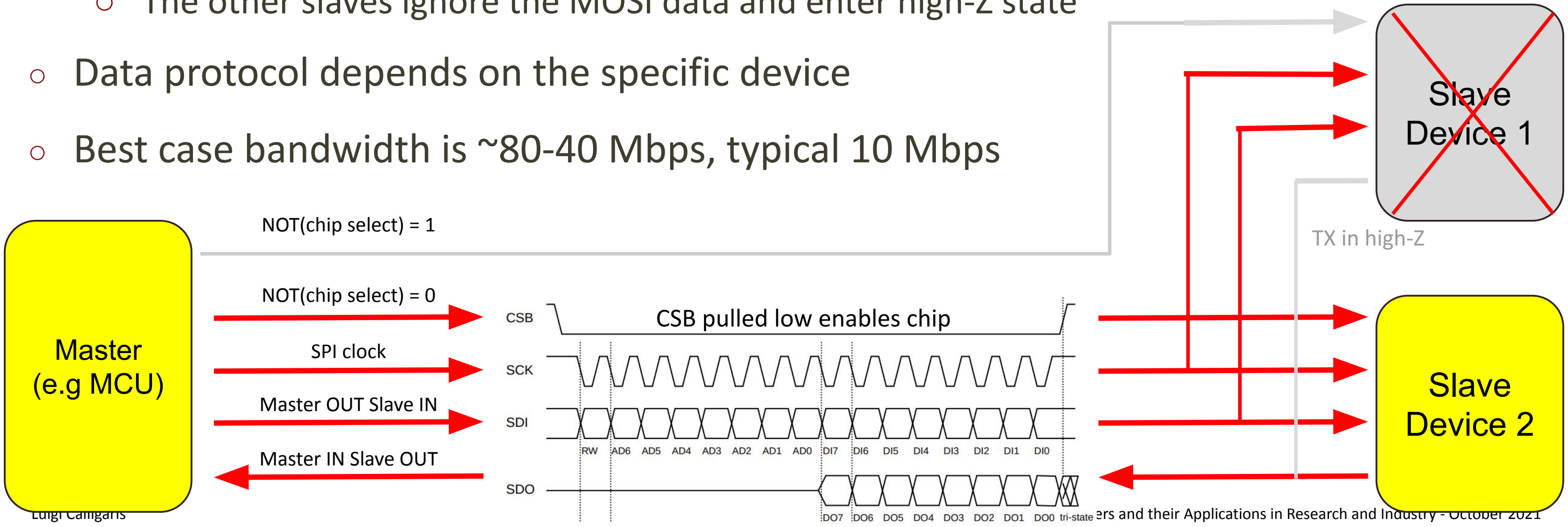
  - <u>Low-bandwidth and short-range</u>
    - Simple, cheap in silicon, low power, PCB space-efficient

Smol & efficient

- Most common on-PCB "slow" protocols
  - SPI, I2C, UART, single-wire

Serial Peripheral Interface (SPI)

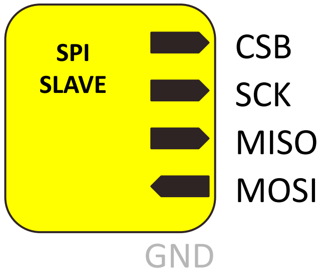
- Master-Slave with monodirectional (4-wire) or bidirectional data wire (3-wire)
  - Clock and data wires are shared between 1 master and one/many slaves
  - One additional wire per chip is needed to select it
    - Serves to signal the slave device that the communication is intended for him
    - The other slaves ignore the MOSI data and enter high-Z state
- Data protocol depends on the specific device
- Best case bandwidth is ~80-40 Mbps, typical 10 Mbps

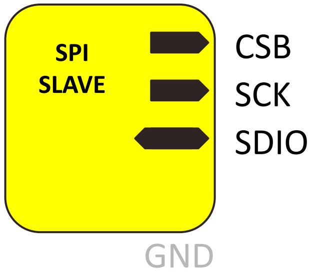


Serial Peripheral Interface (SPI) pinout

4-wire SPI (most common)
CSB or CSN (chip select bar/not)
SCK (serial clock)
MISO (master-in slave-out)
MOSI (master-out slave-in)
GND (PCB ground, no need for wire)

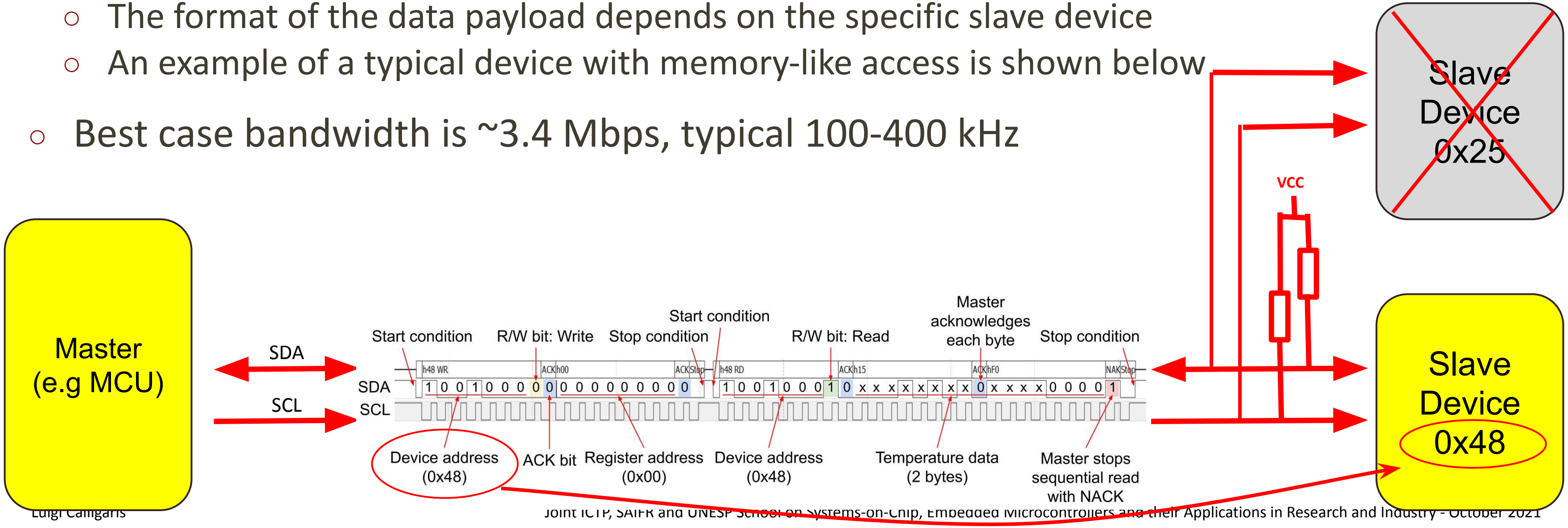
3-wire SPI (less common)
CSB or CSN (chip select bar/not)
SCK (serial clock)
SDIO (serial data in-out)
GND (PCB ground, no need for wire)





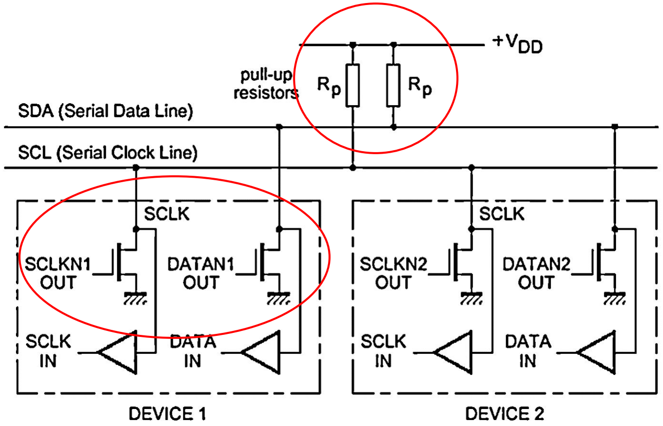
Inter-Integrated Circuit (I2C, IIC or I^{2}C)

- Master-Slave with shared clock & data wires (i.e. 2 wires)
  - Open drain mode (next slide)
  - Chips selected with a 7- or 10-bit address as first data
- I2C protocol defines the start/stop conditions, address, r/w bit, ACKnowledge
  - The format of the data payload depends on the specific slave device
  - An example of a typical device with memory-like access is shown below
- Best case bandwidth is ~3.4 Mbps, typical 100-400 kHz



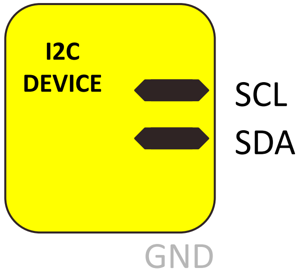
Open Drain

- When many devices drive the same wire there is a risk of short circuit
  - e.g. one device pulls a ‘1’ and another a ‘0’
- Putting a protection resistor in series with each device would not help
  - What if 126 devices pull a ‘0’ and one device pulls a ‘1’ → maybe minority device fries?
  - What if 2 devices pull a ‘0’ and 2 devices pull a ‘1’ → undefined middle state
- Solution: one state (usually ‘1’) is pulled by a common global resistor
  - The other state is just the transition of one device from high-Z to GND
  - This way the current does not depend on the number of devices
- I2C and some other interfaces use Open Drain
- Line capacitance tends to be large w.r.t. pull up resistors
  - The slow rise time of the line limits the min symbol period
  - Open-drain is slow compared to a CMOS gate
    - That’s why I2C is slower than SPI or UART



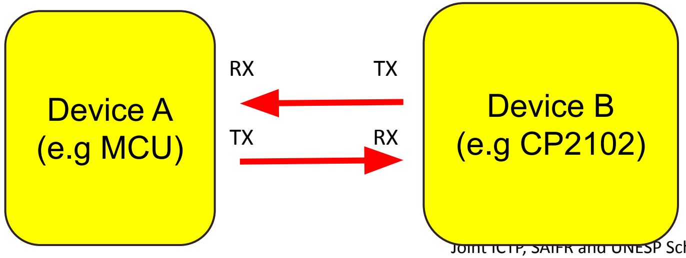
Inter-Integrated Circuit (I2C, IIC or I^2C)

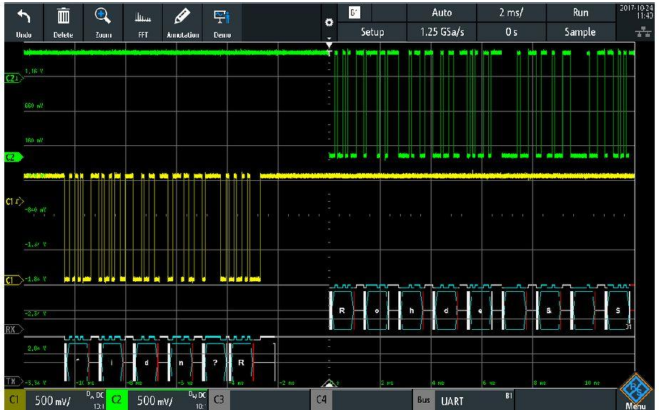
I2C (uses just two wires)
SCL (serial clock)
SDA (serial data)
GND (PCB ground, no need for wire)



Universal asynchronous receiver-transmitter (UART)

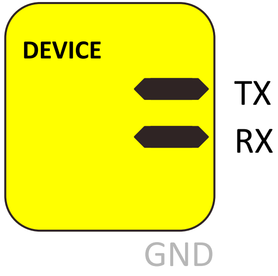
- Symmetric role (no master/slave) and two wires: A(tx)⟶B(rx), B(tx)⟶A(rx)
  - Symbols (e.g. bytes in the case of 8-N-1 mode) are sent one at a time
- UART is asynchronous, uses independent clocks in A and B
  - When B(rx) gets A(tx) start condition, it starts sampling at the expected points
    - e.g. in a 115200 bps connection, every 1/115200 of a second after the start condition
- This works if A and B clock's frequency are not too off (max ~10% disagreement)
- UARTS are everywhere you have a Linux embedded system (routers, cellphones, ...)
- There is a synchronous version (USART)
  - One wire carries a clock from the master





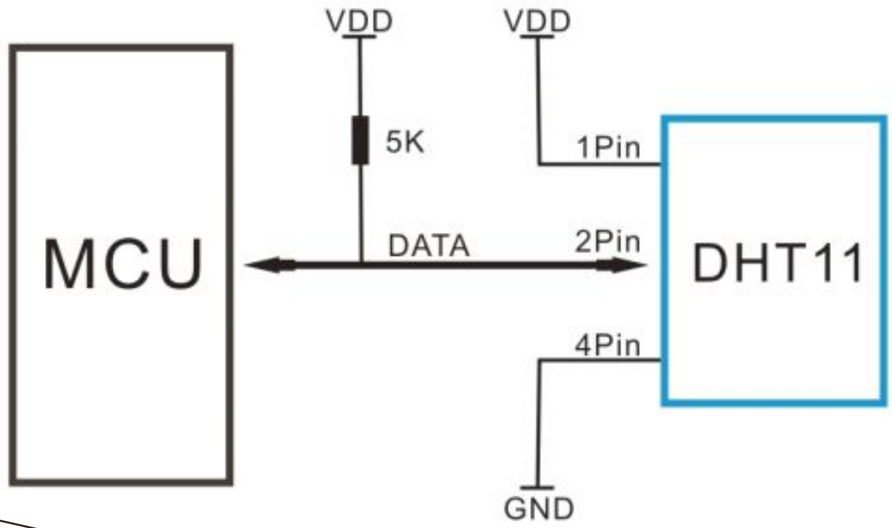
Universal asynchronous receiver-transmitter (UART)

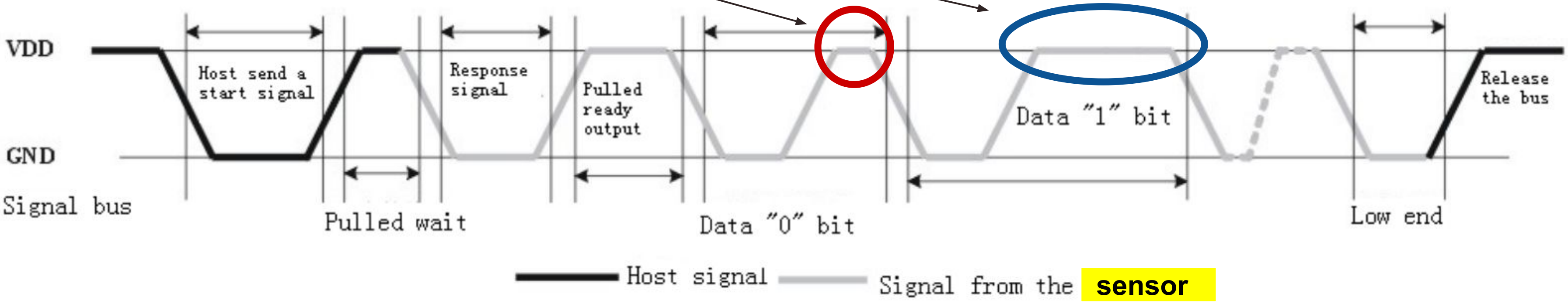
UART (uses just two wires)
TX (data out)
RX (data in)
GND (PCB ground, no need for wire)



Single wire interface

- Some very simple devices use a single wire
  - Uses very little PCB space
  - Usually a rather slow connection
  - Often open collector bus
  - e.g. the DHT11, DHT22 and AM2302 thermometer sensors use this interface
- Usually data encoded by pulse length
  - **“1” = long high pulse**
  - **“0” = short high pulse**

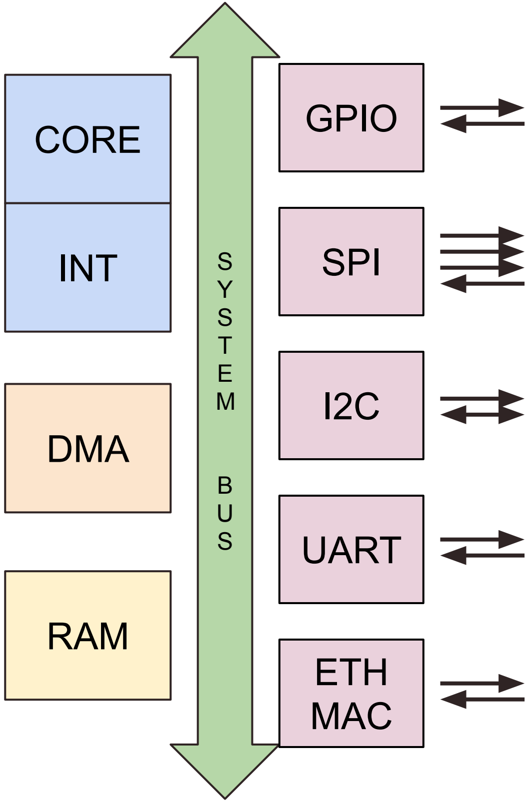




Some common peripherals

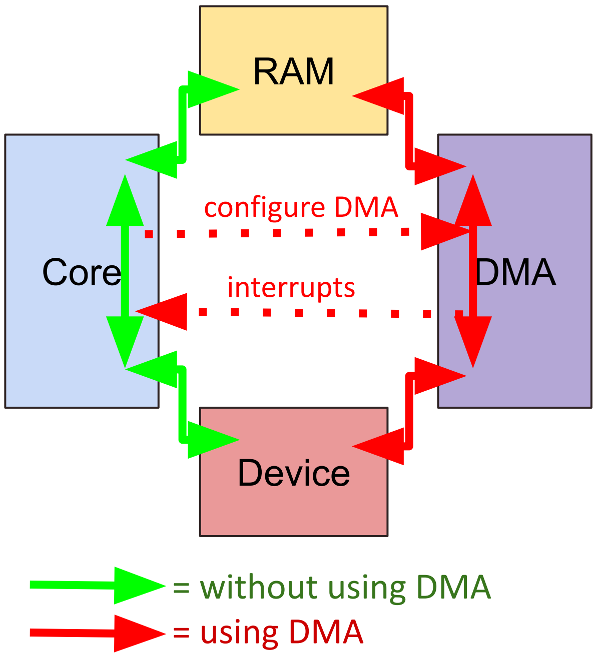
GPIO, SPI, I2C, UART, CAN, Ethernet...

- MCUs need to interact with the world
  - They always have General Purpose IO (GPIO) capability
    - Directly control the HIGH/LOW state of a pin
    - You can emulate any digital protocol
    - (you may pay a timing and speed penalty)
  - They often have plenty of communication peripherals
    - In-silicon protocol intelligence → fast & reliable
    - Usually restricted to the most popular protocols
    - SPI, I2C, UART, CAN, Ethernet MAC, …
- These peripherals are slow wrt the core
  - 100 kHz - few MHz vs 10-500 MHz
  - They are a good case for the use of INT and DMA
  - We will see some examples in the lab exercises



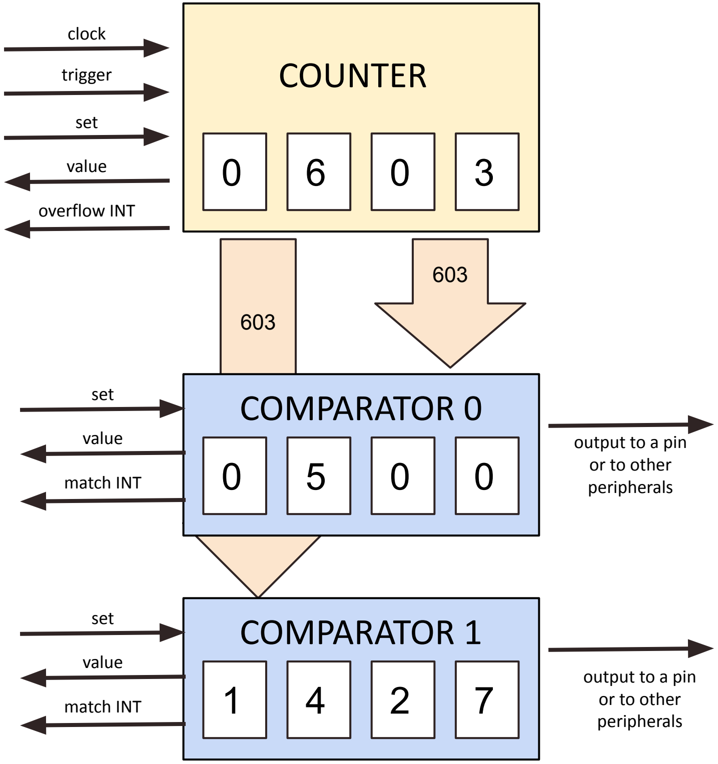
Direct Memory Access (DMA) controller

- Device that can act as a memory/reg master like the core
  - Used to offload data copy/move operations from the core
- Programmable to copy data between addresses
  - Copy ADC reads into memory (e.g. FIFO, ring buffer, …)
  - Play data from a memory buffer into a DAC
  - Transmit a large amount of characters in a buffer through UART
  - Receive a slow I2C data stream into memory
- The DMA controller usually has multiple channels
  - When configuring them, each channel is given a priority
  - The DMA controller serves requests triggered by peripherals
    - e.g. the UART receive buffer on the peripheral is close to full
- An MCU can have more than one DMA controller



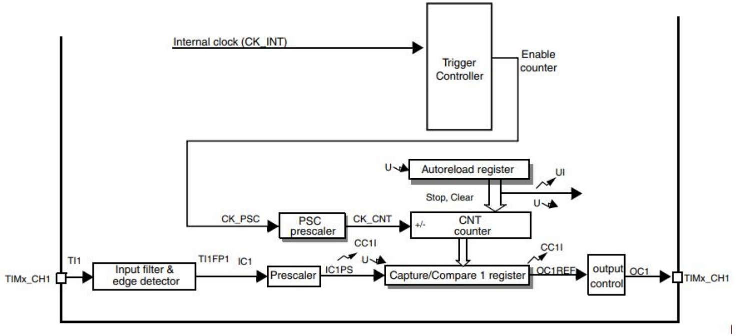
Timers

- Counter + one or more comparators
  - Counts up/down/up+down in a range [0, N]
  - Can be set to auto-reload or just do one round
  - Can be free-running or triggered by signals
  - Can generate interrupts upon under/overflow
- Comparators store a fixed value
  - Output high if counter <, =, > the value
  - Can generate interrupts in case of match
- Timers used in many real-time applications
  - e.g. FreeRTOS scheduler tick, data protocols, ...
  - They allow to offload the core from expensive polling
- Also, timers can be used for PWM

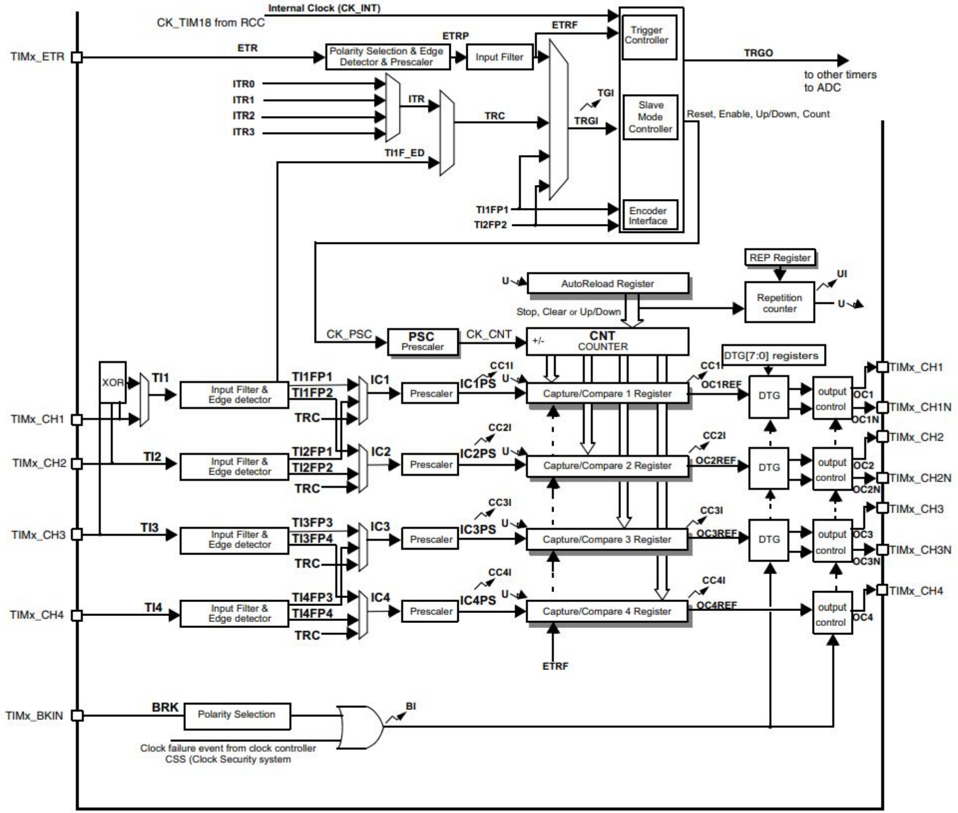


Timers

- MCUs can have different timers
  - 8-, 16-, 32-bits (shorter/longer period)
  - Advanced/simple features
  - One/many channels
  - Features/space compromise
- Choose the right TIMer for the job
  - The manual/datasheet holds the details



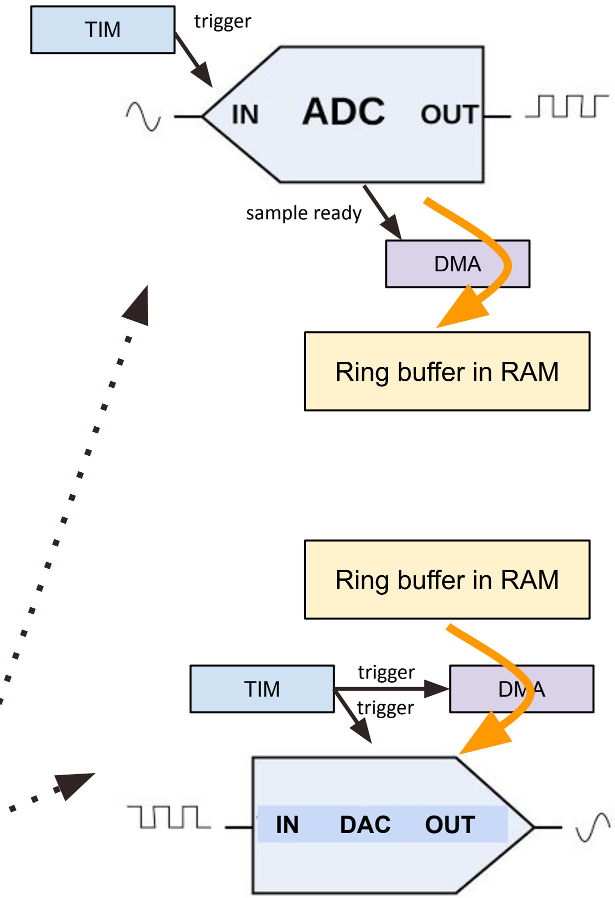
Block diagram of a STM32F411 General Purpose timer (TIM10/11)



Block diagram of the STM32F411 Advanced control timer (TIM1)

DACs and ADCs

- Digital-to-Analog Converters
  - Convert a stream of data to an analog V or I
  - Many different types of DACs ⟶ speed vs precision vs complexity
- Analog-to-Digital Converters
  - Sample digitally an analog input signal
  - Many different types of ADCs ⟶ speed vs precision vs complexity
- Some MCUs have highly configurable analog stages
  - e.g. look for the **Cypress PSoC 5LP** Architecture TRM
- ADC sampling and DAC conversion: hard real time
  - Cannot get **timing** wrong sampling/generating a **fast waveform**
  - Excepting slow signals, very often ADC/DACs applications will use
    - An internal **timer** or external pulse to trigger the ADC/DAC
    - A **DMA** controller to copy data to/from a ring buffer in RAM

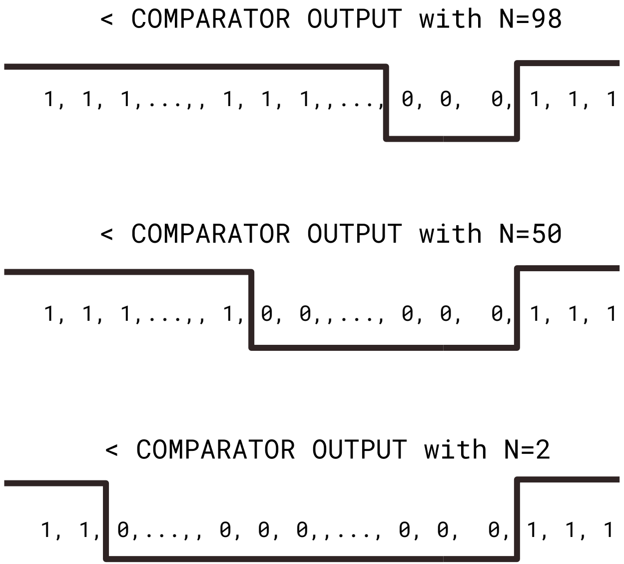


Use of timers for PWM waveform generation

- DACs can output very clean waveforms but..
  - Many applications do not need fast&clean waveforms
    - We can save silicon with a simpler circuit
  - Some applications prefer '1' and '0' states
    - example: power electronics (IGBTs, SiC transistors, ...)
      '0' state → high R, no conduction → no dissipation
      mid state → mid R, conduction → **high dissipation**
      '1' state → low R, conduction → low dissipation
- A timer can be used generate a square wave
  - example: TIM period set to 100
  - comparator set to N, output high for counter < N
  - → **we control the duty cycle with the comparator!**

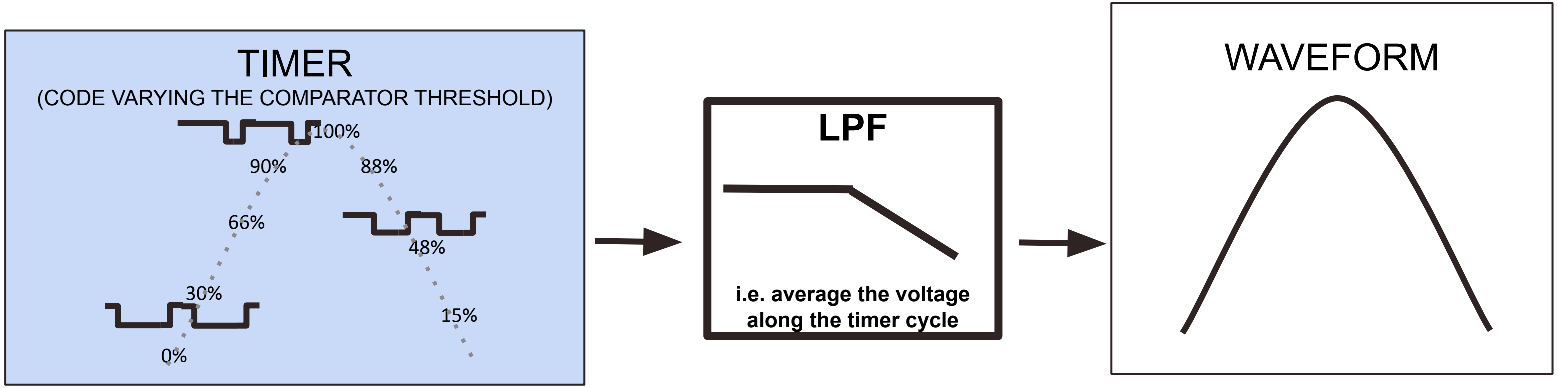
COUNTER STATE

0, 1, 2,....,49,50,51,,...,98,99,100, 0, 1, 2



Use of timers for PWM waveform generation

- You want to generate a PWM waveform out of your system?
  - Set up the timer clk frequency >> typical frequency of your signal
  - Have the code set comparator value proportional to desired signal value
    - Now you are modulating the pulse with with the desired waveform
  - Filter the high frequency component of the switching with a Low Pass Filter
    - Voilá! You got the waveform you wanted!



Applications of timers for PWM

- Most of modern power electronics
  - High power traction motors for trains, metros, elevators, cars
  - AC/DC converters in electrical transmission
  - DC/AC converters in solar photovoltaic generation
  - E-bikes, E-scooters, hoverboards, drones
  - Sounds you hear when you ride a train accelerating/decelerating
    - PWM modulations of the motor winding voltages (VVVF drive)

- YouTube has many demo videos showing the waveforms live on an oscope
  - Search for example for “PWM motor” or “VVVF drive”

- Note: I am NOT an expert on PWM techniques and motor control
  - There are entire books just about the magic you can do with it
  - I just think this is a really cool topic on which to close this talk!

[School home](#)

Thank you!

Questions? :)