General info

- The course contains some “lecture – like” part and a lot fo practical exercises
- The material for this course is summarised on a web size (lectures and exercises):
  - https://microcontoller-course.web.cern.ch
- If you want you can also have the slides, but they do not contain more information
- During the course you are allowed to take the microcontroller home and work with it if you are motivated.
- I will ask you twice to read a document (data-sheet) at home so that we can be more efficient during the exercises and do not loose time with reading.

Contents of the course

- Characteristics of Microcontrollers
  - Includes some basics of memory-handling strategies (hardware and software)
- Peripherals
  - With details on I2C and SPI and usage of an OLED display
- Networking
  - Includes a “crash course” of networking in general
- Asyncio
  - Widely used and part of the python-language; a light-weight way of scheduling
- MQTT
  - A popular simple network protocol used in IoT and elsewhere

Microcontrollers

Part I

Comparison: Computer - Microcontroller

Basic concepts

Examples

Development environment

(Christoph.Schwick@cern.ch)

Computers vs Microcontroller systems

	“standard” computer	Microcontroller system
Usage	• Universal tool running different software applications for different tasks/problems	• Dedicated design for a specific tasks • Aimed at mass products
OS	• User interface • Drivers to access different hardware • Services to execute different applications (also concurrently)	• No OS or • Minimal OS with very limited purpose (e.g. FreeRTOS)
Scales	• Laptop to rack mounted servers • O(100W) to some kW • O(300Euro) to some 10kEuro	• O(10cm^2) • 1...some W • < 10 Euro
Updates	• Frequent update of applications or OS to fix issues or include new functionality	• No updates or rare updates via specialized procedures
Peripherals	Plugged into the computer	A lot of built-in peripherals

Technical Characteristics : CPU

	“standard” Computer	Microcontroller system
Cores	Up to O(100)	1 or 2
Frequency	Some GHz	Up to some 100 MHz
Power	O (100W)	1W-5W
Instruction Set	Large	Small to medium
Caching	Several levels	No caching or primitive 1 level
MMU	Always present	Not available
RAM Memory	Up to 100s of GB	Bytes to 100s of Kilobytes

- Why is the frequency of microcontrollers so much lower?
- What is cache memory?

Intermezzo:

How is memory organized?

How is memory allocated?

What is a MMU

Microcontroller and Memory architecture

- Architecture : Harvard vs Neumann
  - In Microcontrollers programs are stored in “flash” memory chips (“permanent storage”). Programs are executed directly from the flash. Flash memory is cheap.
    - However, writing to flash memory is very slow. In addition flash memory “wears out” (limited amount of write cycles before they break) → no good media for data storage during operation
  - Therefore Microcontrollers usually have different memory types (and electrical signals) for Program (FLASH) and Data (RAM) → **Harvard architecture**
  - Computers use the same RAM for Program and Data → **Neumann architecture**
    - Programs are loaded into RAM before execution (e.g. from harddisk)
- Cache memory
  - Very fast (and expensive) memory close to the CPU.
  - In computers instructions or data are “loaded” first into the cache and from there into the CPU
    - If they are used repetitively (loops in programs, re-used data structures) the do not need to be re-loaded → huge performance increase
  - In computers multiple layers of cache exist
  - Microcontrollers do not have or only have very limited amount of cache memory

Memory Management (I): basic allocation types

(We talk about management of the RAM memory for data)

- **Static memory allocation**
  - RAM memory can be allocated (=reserved) at compile time (i.e. at the time of the compilation it is clear how much memory needs to be allocated and this block of memory is then reserved when the program starts. It is kept for the entire time the program is running.
  - Static memory allocation is save (once the program starts successfully nothing can go wrong anymore) but inefficient (you have to allocate all memory which will be used in the worst case for the entire time the program is running. Memory can not be given back to the system in order to be used for another purpose → in general memory is wasted
  - Due to the predictability and the simplicity safety relevant applications often only allow static memory allocation

- **Dynamic memory allocation:**
  - At run-time (i.e. during the execution) memory si requested and reserved for a temporary specific purpose. Then it is given back (freed) to the system.
  - Example: A jpeg file needs to be loaded into the memory to do some processing. Afterwards the memory is given back to the system for other use.
  - Memory is used much more efficiently
  - There is a risk that during the run-time a memory request cannot be fulfilled since not enough memory is available in the system → in general a program crash follows…
  - Easy to introduce bugs: dangling pointer to memory region that was “freed”

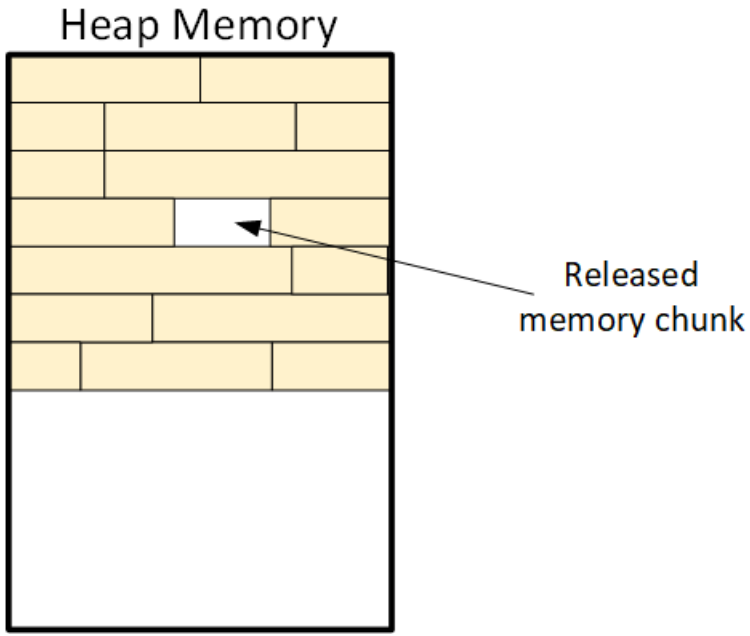
Memory Management (II): types of RAM memory

- **Stack memory**
  - A chunk of memory which is allocated for every execution thread. It is used to store local variables of subroutines, or return addresses of subroutines.
  - The stack size is fixed at compile time. It can be configured when compiling the program
  - The stack is usually starting from a high address and growing down to lower addresses.

- **Heap memory**
  - In general the heap memory is all other RAM memory which is not used by the Stack or the program instructions itself
  - It is used to allocate blocks of memory during the program execution (at “run-time”)
  - Example: A jpeg file needs to be loaded into the memory to do some processing. Afterwards the memory is given back to the system for other use.

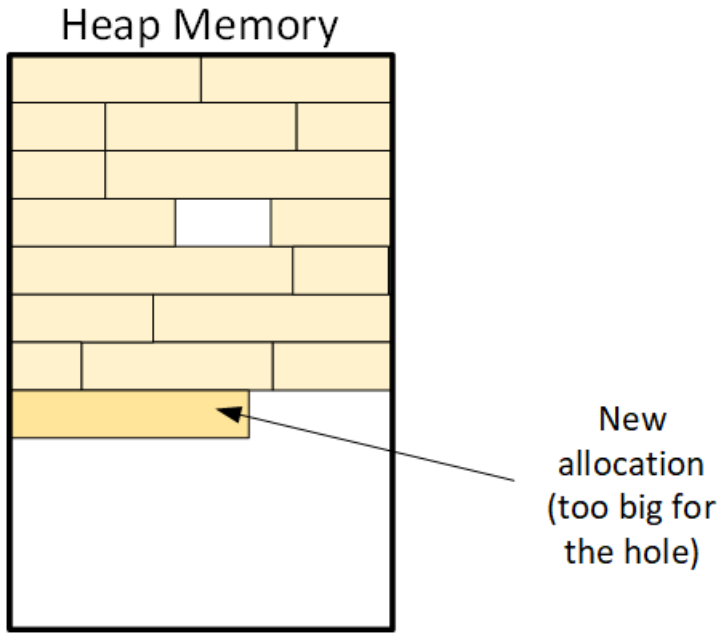
Memory Management(III): Fragmentation

- Memory fragmentation of the HEAP
  - **Heap memory** is the memory from which the application allocates memory at “run time” (i.e. during it’s lifetime): dynamic memory allocation
  - In general the blocks are freed during execution time, when they are not used anymore.



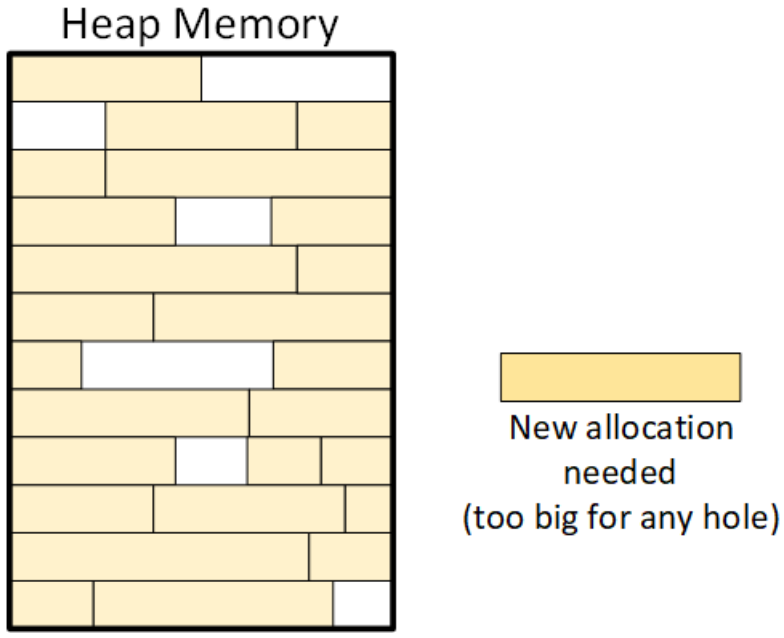
Memory Management(III): Fragmentation

- Memory fragmentation of the HEAP
  - **Heap memory** is the memory from which the application allocates memory at “run time” (i.e. during it’s lifetime): dynamic memory allocation
  - In general the blocks are freed during execution time, when they are not used anymore.



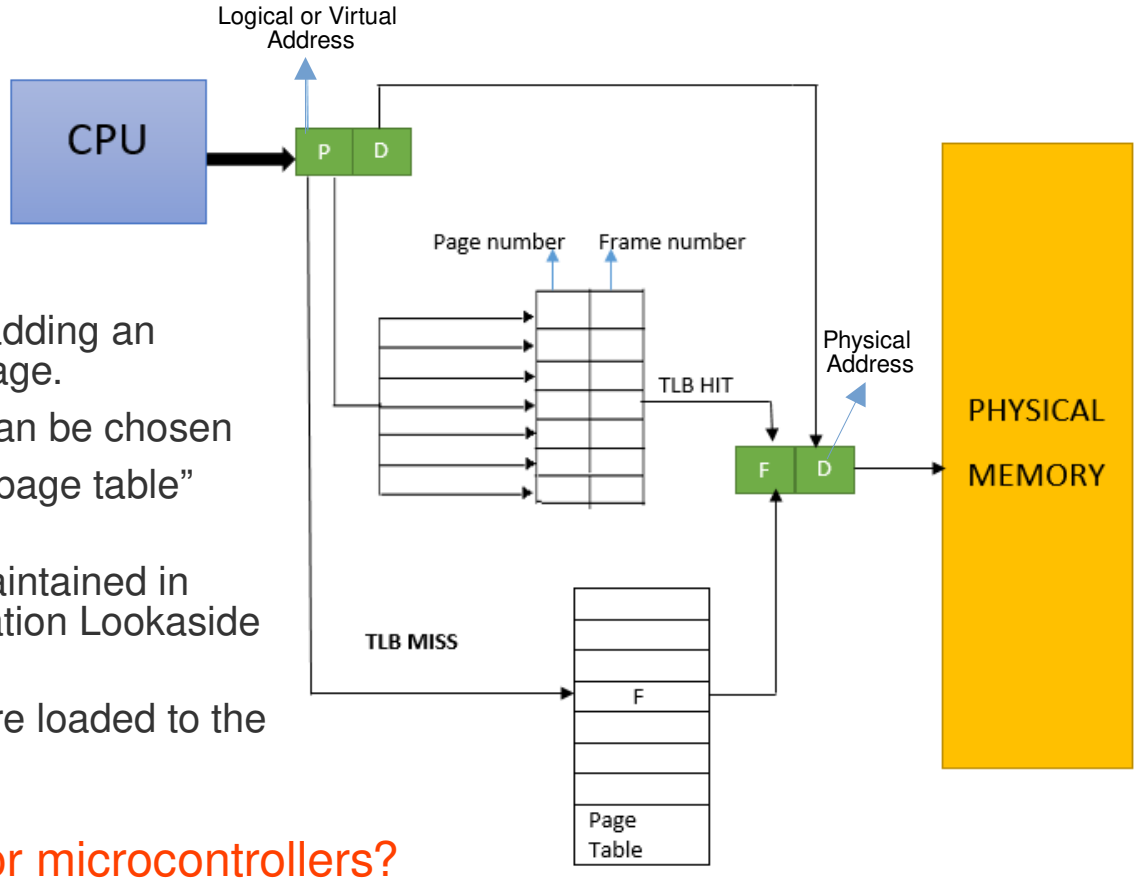
Memory Management(III): Fragmentation

- Memory fragmentation of the HEAP
  - **Heap memory** is the memory from which the application allocates memory at “run time” (i.e. during it’s lifetime): dynamic memory allocation
  - In general the blocks are freed during execution time, when they are not used anymore.
  - In a fragmented heap, memory allocation **can fail** even though the sum of all free memory is larger than the requested memory. The size of the free blocks must be larger or equal to the size of the requested blocks



Memory Management: MMU

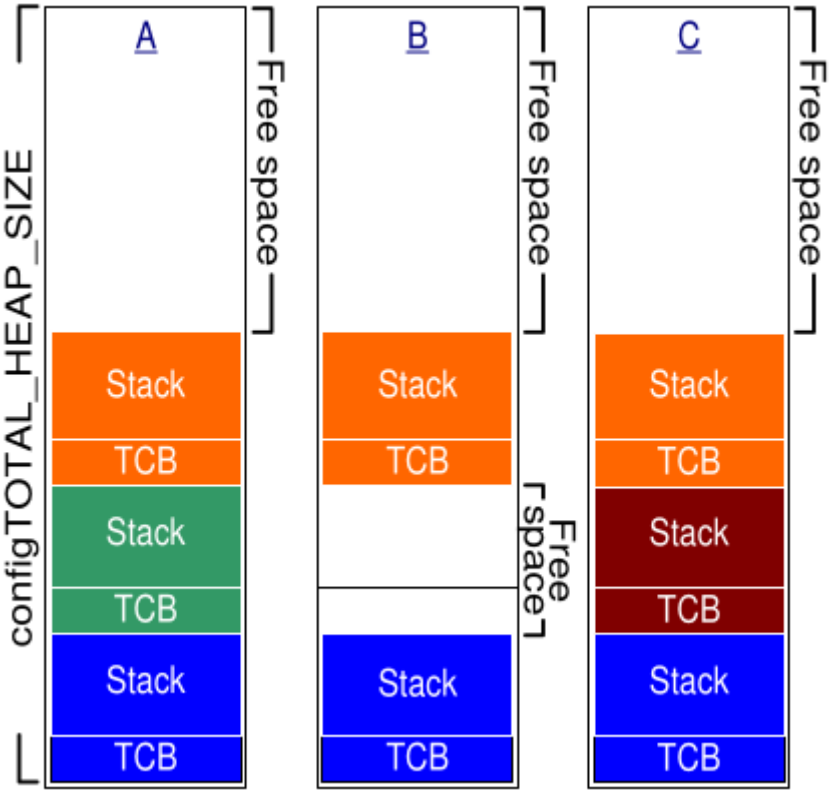
- A memory management unit helps to mitigate the effect of fragmented memory
  - It works with fixed memory blocks (typically 4kB)
  - It translates from Physical addresses (of memory chips) of the blocks to virtual addresses (seen by applications/CPU)
  - The translation means simply adding an offset to each address of the page.
  - For each page another offset can be chosen
  - Offsets of all pages are in the “page table” (maintained in memory)
  - A subset if the page table is maintained in a fast special RAM (the Translation Lookaside Buffer (TLB)). It is expensive…
  - Always the most used pages are loaded to the TLB (caching mechanism)



Is this a good technique for microcontrollers?

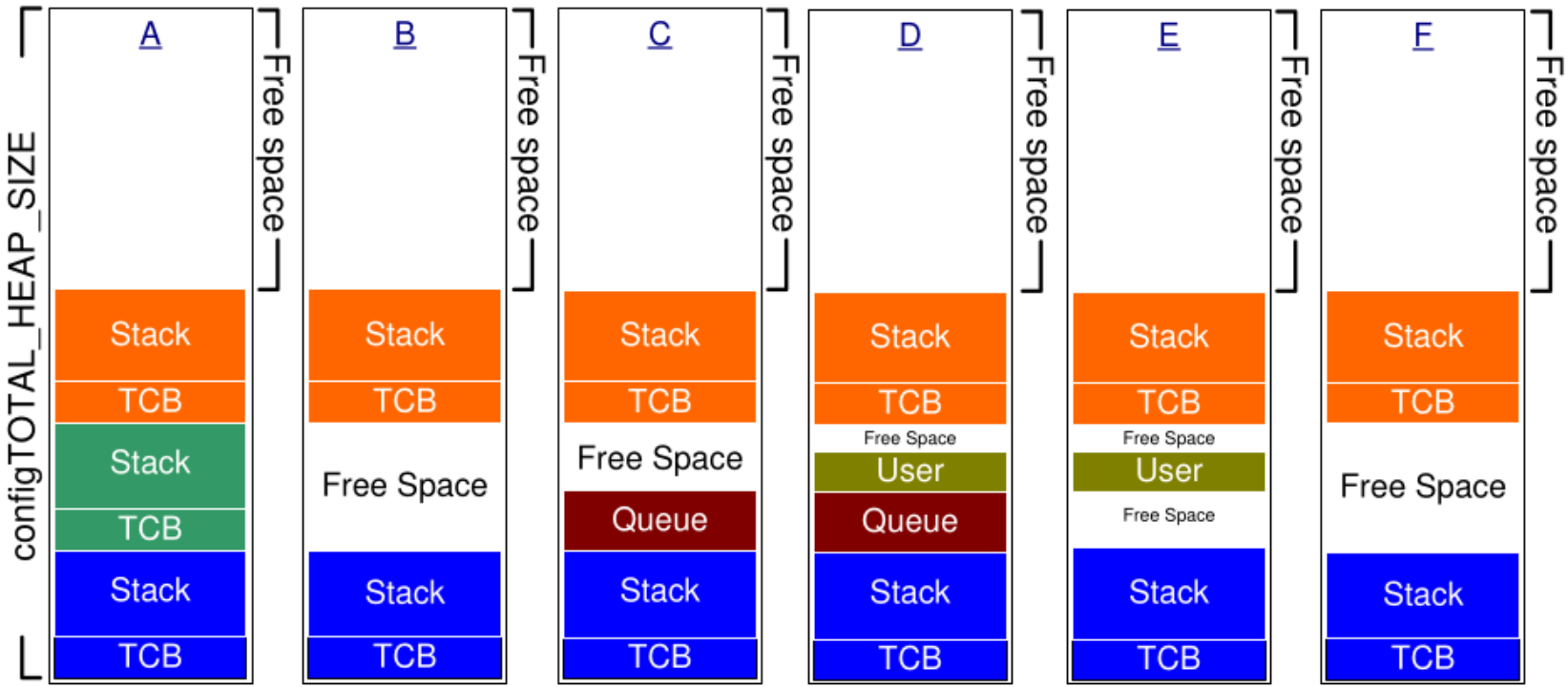
Memory management in microcontrollers: no MMU

- HEAP_2 algorithms in FreeRTOS
  - Statically allocate a large block of data at startup (this becomes the HEAP)
  - Always look for the “best fit” when an allocation request comes (i.e. waste as little as possible)
  - Works very well when blocks of constant size are allocated and freed.
  - Less good when the block size changes continuously
  - Very simple and fast



Memory management in microcontrollers: no MMU

- HEAP_4 algorithms in FreeRTOS
  - Similar to HEAP_2 but fuse neighbored blocks to larger chunks → less fragmentation
  - Better for varying block sizes, however, more complex and slower.



Peripherals

Peripherals: Computer

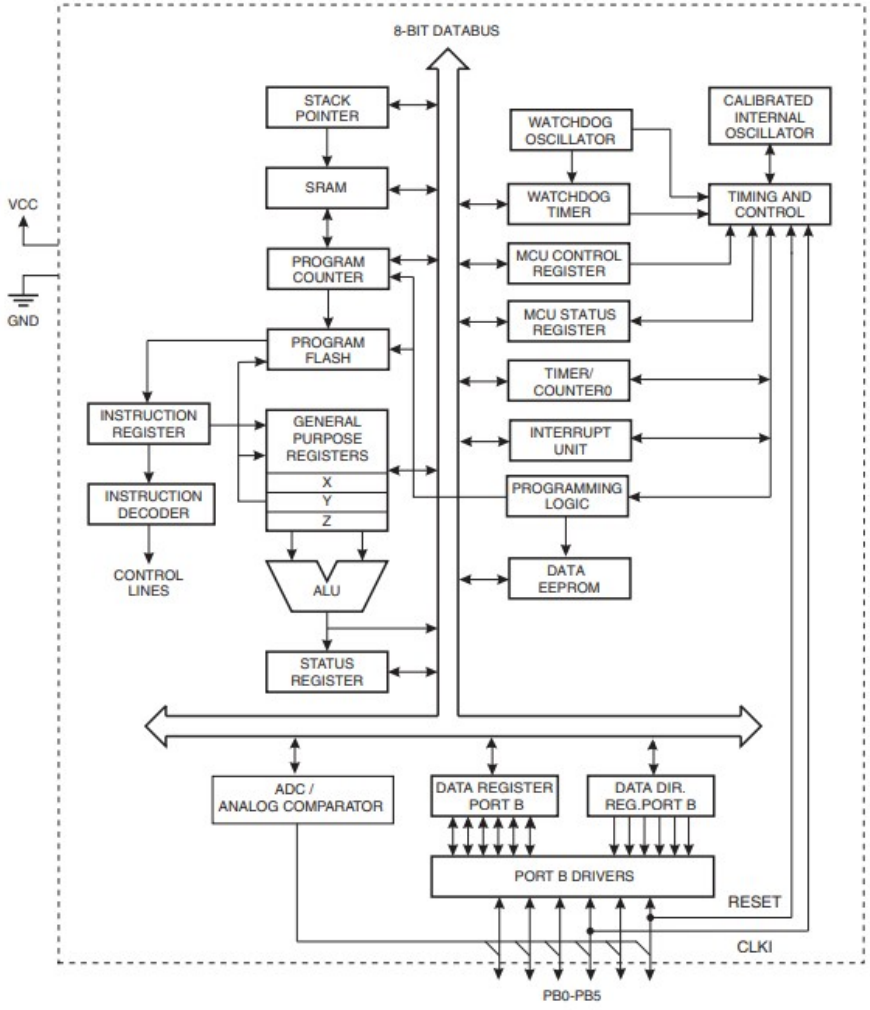
- Any computer MUST have peripherals to be useful
  - Some form of input and output is necessary. Otherwise a computer can only be used as an expensive heating…
- Computer
  - Peripherals are devices which must be added to the computer
    - Keyboard, harddisk, Graphic card and screen, sound card, networking electronics, …
    - There are standardized interfaces to attach peripherals
      - Parallel Port, RS232, SCSI, PCI, PCIe. USB, VGA, HDMI,thunderbolt, …
      - These are all electronic specifications how to exchange data between digital devices.
    - Software components called “Drivers” have to be loaded into the Operating System (OS) in order to use the devices. The knowledge about how to operate these devices is in these drivers. The OS and the applications use the API of the drivers to use the peripherals. (API: Application Programming Interface: Essentially a documented library of functions which can be called by applications to use the devices.)
    - Usually devices can be plugged and unplugged during the operation of the computer: “hot plug” (This was different in the old times…)
    - The user buys the peripherals which are needed.

Peripherals: Microcontroller

- Microcontrollers already contain many peripherals.
  - Given that they are often used in consumer products there is a standard set of peripherals which are needed very often. The chip producer integrate these in the microcontroller.
  - Standard interfaces have been developed by the industry to connect electronic devices and let the communicate (I2C, SPS, UART, I2S, …) These are also considered “peripherals” in the microcontroller language. But they need another digital electronics module at the other end to be useful: like a sensor, display, ...)
  - Typical peripherals in microcontrollers are:
    - Counters, Analogue Digital Converters (ADCs) to measure voltage, Digital to Analogue Converters (DACs), Real Time Clock (RTC), Timers, Touch Sensors, Pulse Width Modulators (PWM)
  - Typical interfaces which are implemented are:
    - UART, I2C (a two wire bus for digital communication), SPI (similar to I2C), I2S (similar to I2C but specifically for transmitting digital audio), General Purpose Ios (GPIOs)

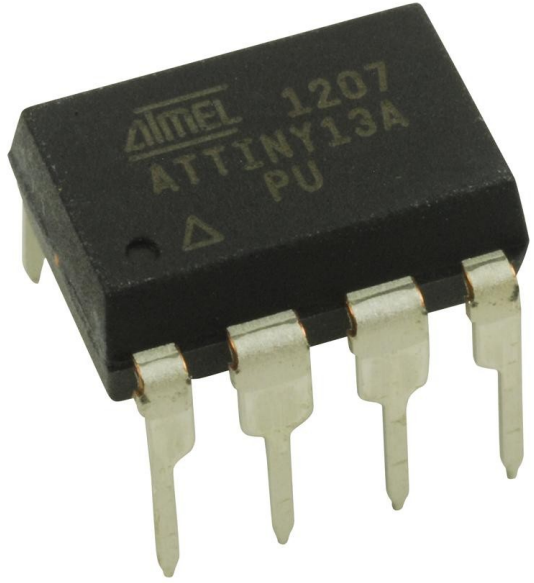
Examples of microcontrollers

Examples of Microcontrollers

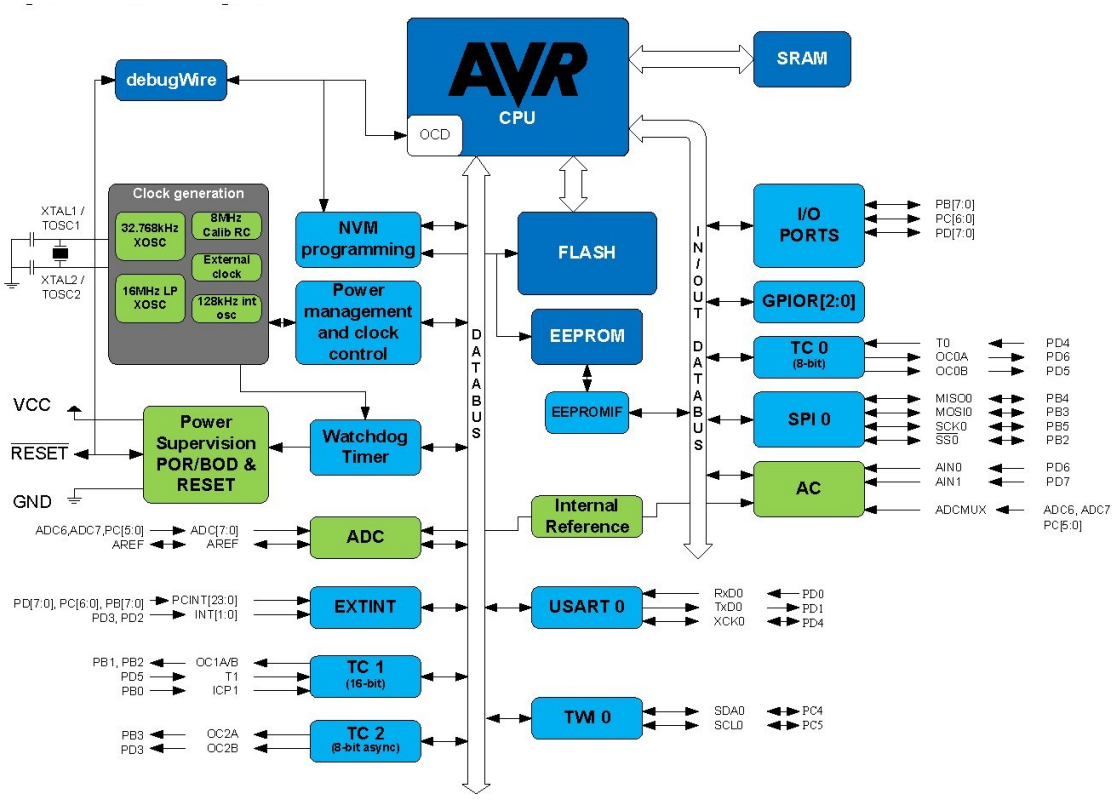


A very simple µController: ATTiny 13A (ATMEL)

- 1kB Flash Memory
- 64 Byes RAM
- 32 registers
- 20MHz clock, 1 instruction per clock cycle
- 4 channel ADC (10bit)

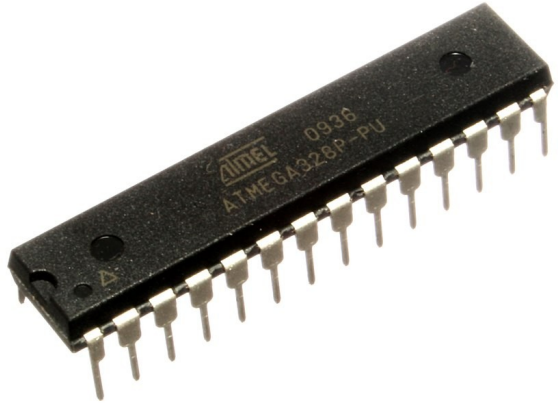


Examples of Microcontrollers

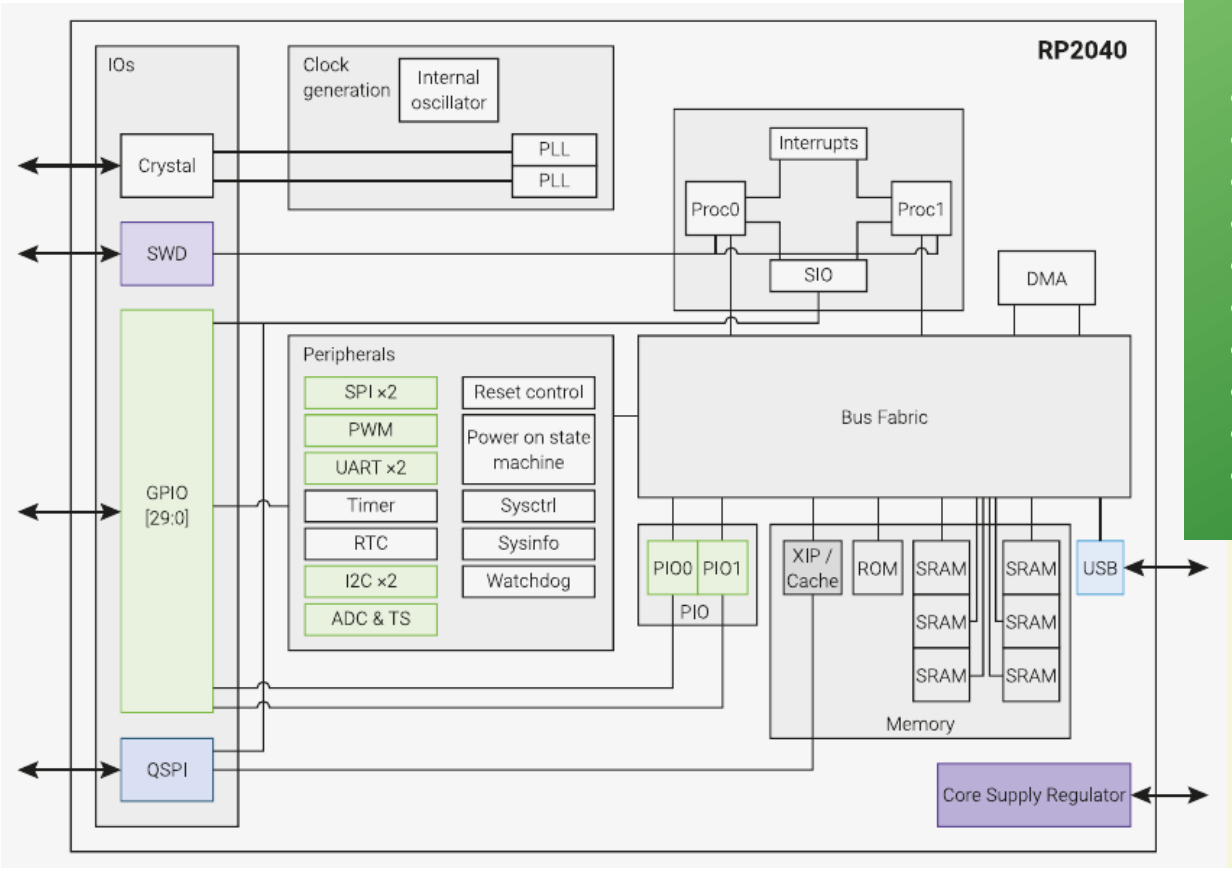


Slightly more complex µController: Atmega328 “Arduino” (ATMEL)

- 32 kB Flash Memory
- 2 kB RAM
- 32 registers
- 20MHz clock, 1 instruction per clock cycle
- 6 channel ADC (10bit)
- UART
- 2 wire interface (SPI, I2C)
- 16 touch sensors (capacitive)

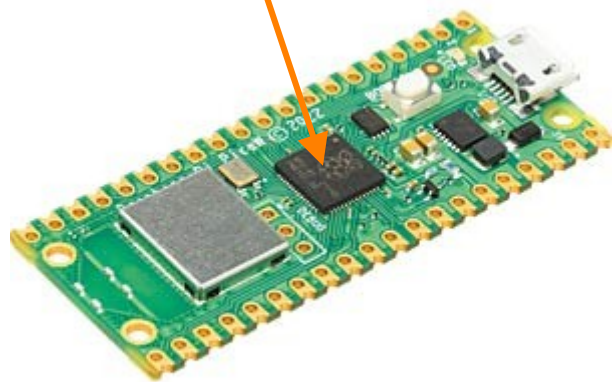


Examples of Microcontrollers



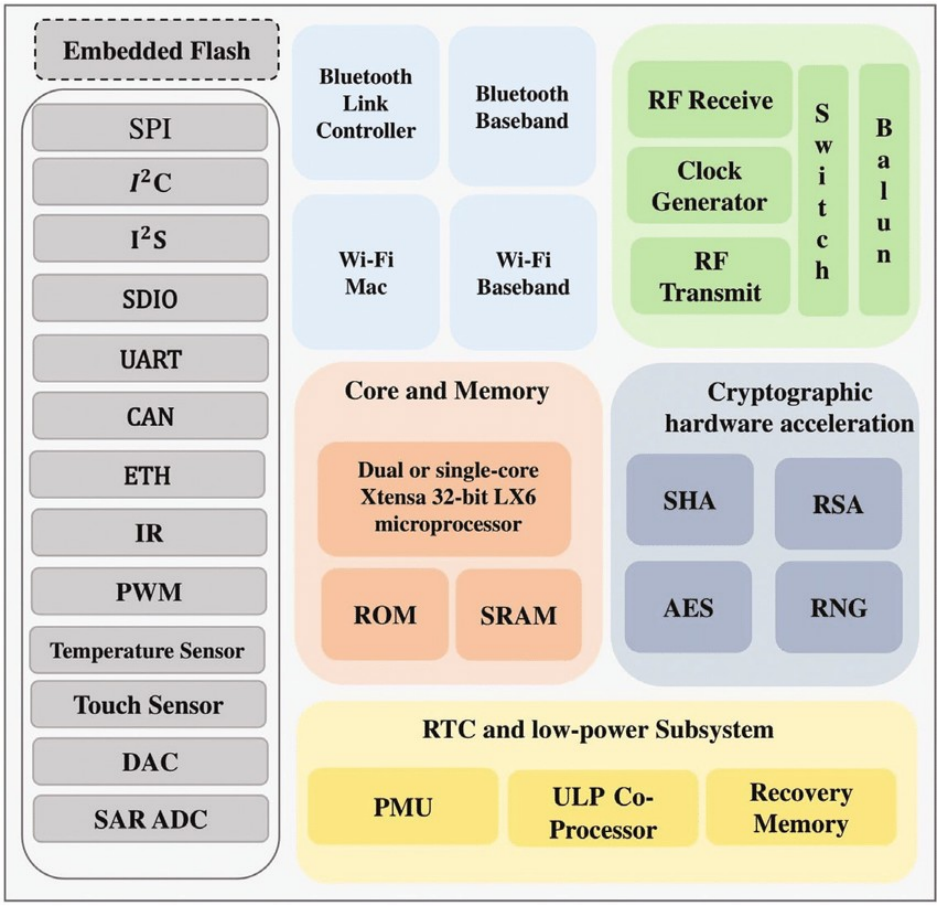
Raspberry Pi PICO RP2040

- external Flash Memory (up to 16MB)
- 264 kB RAM
- ARM Cortex M0+ processor core
- 133 MHz clock
- 6 channel ADC (12bit)
- 2xUART, 2xI2C, 2*SPI
- 8 Programmable IO (PIO) state machines
- 2 interpolators
- ADC channels, PWM channels, …
- USB interface



Examples of Microcontrollers

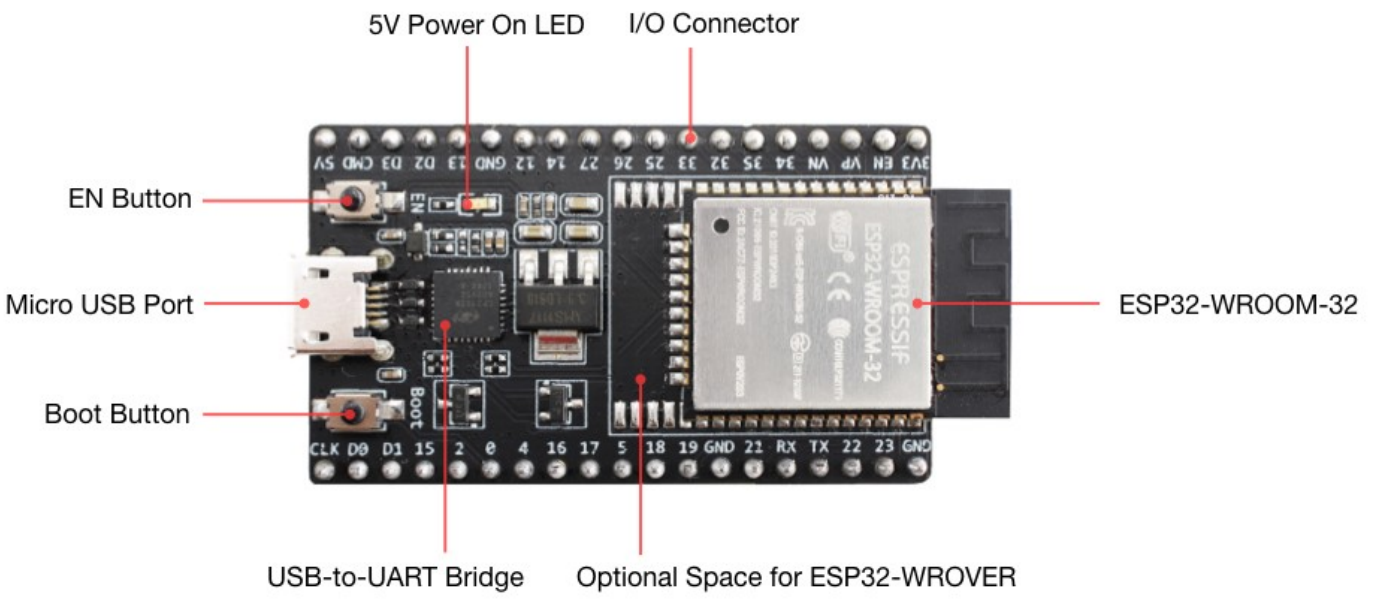
ESP32 Function Block Diagram

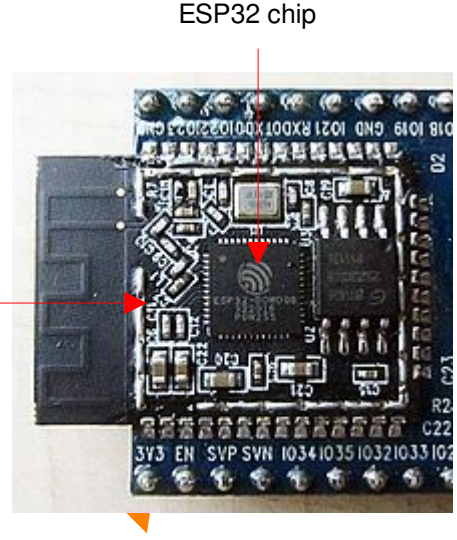


ESP32

- external Flash Memory (up to 16MB)
- 520 kB RAM
- Xtensa LX6 dual core processor
- 240 MHz clock
- 16 channel ADC (12bit)
- Peripherals: see block diagram (gray column)
- Ultra Low Power processor (ULP) for activities in sleep mode
- Wireless RF interface integrated :
  - WIFI or Bluetooth, or custom protocols

Examples of Microcontrollers





Development systems

Programming a microcontroller

- Mostly programmed in “C” or “C++” programming language
- Using a “Cross compiler” on a “Host computer”
  - Microcontroller not powerful enough to compile
  - Allows to use the convenient environment on a “normal” computer.
  - Much much faster…

Typical Development cycle

1) Develop (=edit the program) and compile on the computer
2) Transfer compiled program to Microcontroller via USB cable (special tools for this)
3) Run the program and get debug info (=printout) over the USB cable to the computer

In this course: Use python

- Recently microcontrollers have become powerful enough to be able to compile and run Python programs
  - Usually they use a simplified version of python, tailored to the microcontroller architecture
  - Python modules to access the peripherals are included.
  - **Advantage:** Easy programming, Fast development cycles
  - **Disadvantage:** Slower programs, requires more memory

- One widely used python environment is “micropython”. We will use it in the course.
- The python compiler needs to be installed on the microcontroller.
- On the host computer some tools need to be installed to communicate with the micropython environment on the microcontroller
  - Need to transfer and install python files, python modules, …
  - Need to run programs

Homework

- Read the data sheet of the Sensor we want to use tomorrow.
  - See what you understand and what not so that you can ask the relevant questions during the exercise tomorrow.
  - The datasheet you find in the first reference at https://microcontroller-course.web.cern.ch/Exercises/I2C_sensor/BME280.en/
  - The “First Step” section gives you some indication of the most important chapters in the data sheet.