
# PROGRAMMERING AV INBYGGDA SYSTEM

## advanced C

Dario Salvi, 2024

# Materials

- Making embedded system: chapter 7 “Updating Code”
- https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/
- ESP32 Technical Reference Manual

# C program structure

- C programs are split into modules.
- Each module is compiled **independently**.
- All modules are then put together by the **linker**..

# Definition vs Declaration

- **Definition** means the place (the only one) where a variable or function is created. It is reserved space in memory
  - A function definition *contains the code* that describes what to do when called
  - This is usually done in .c files

- **Declaration** means the places where the type of the variable or function is described so that it can be used correctly
  - But no space is reserved in the memory for the variable
  - A function declaration ("function **prototype**") allows the compiler to check that the function is used correctly with the right type of argument in different modules
  - This is usually done in .h files

# Source vs Header files

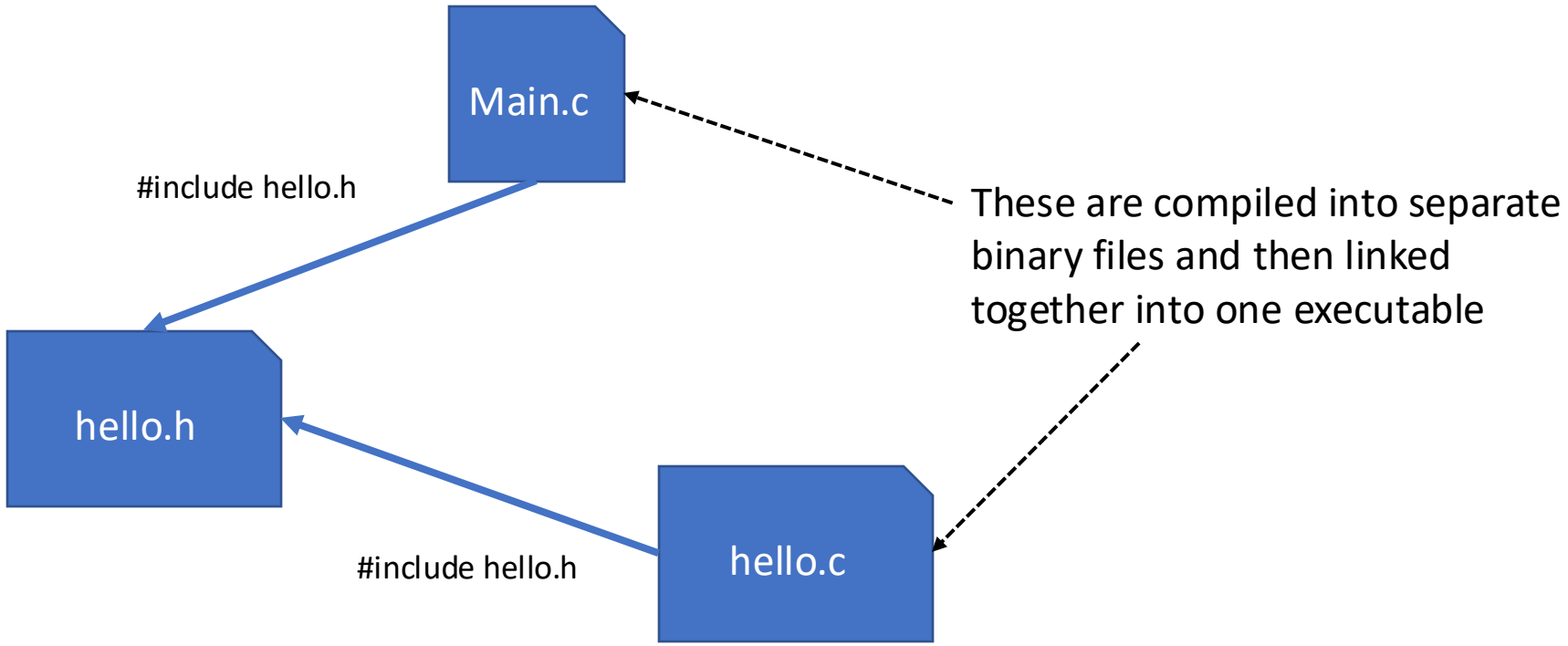
- A common convention in C programs is to write a header file (with **.h** suffix) for each source file (**.c** suffix) that you link to your main source code.
- The logic is that the .c source file contains all of the code and the header file contains the function **prototypes**, that is, just a declaration of which functions can be found in the source file.

- You **define** a variable in a c-file, usually not in a h-file.
- You **declare** a function in an h-file with the same name as the c-file where it is **defined.**

# Live coding

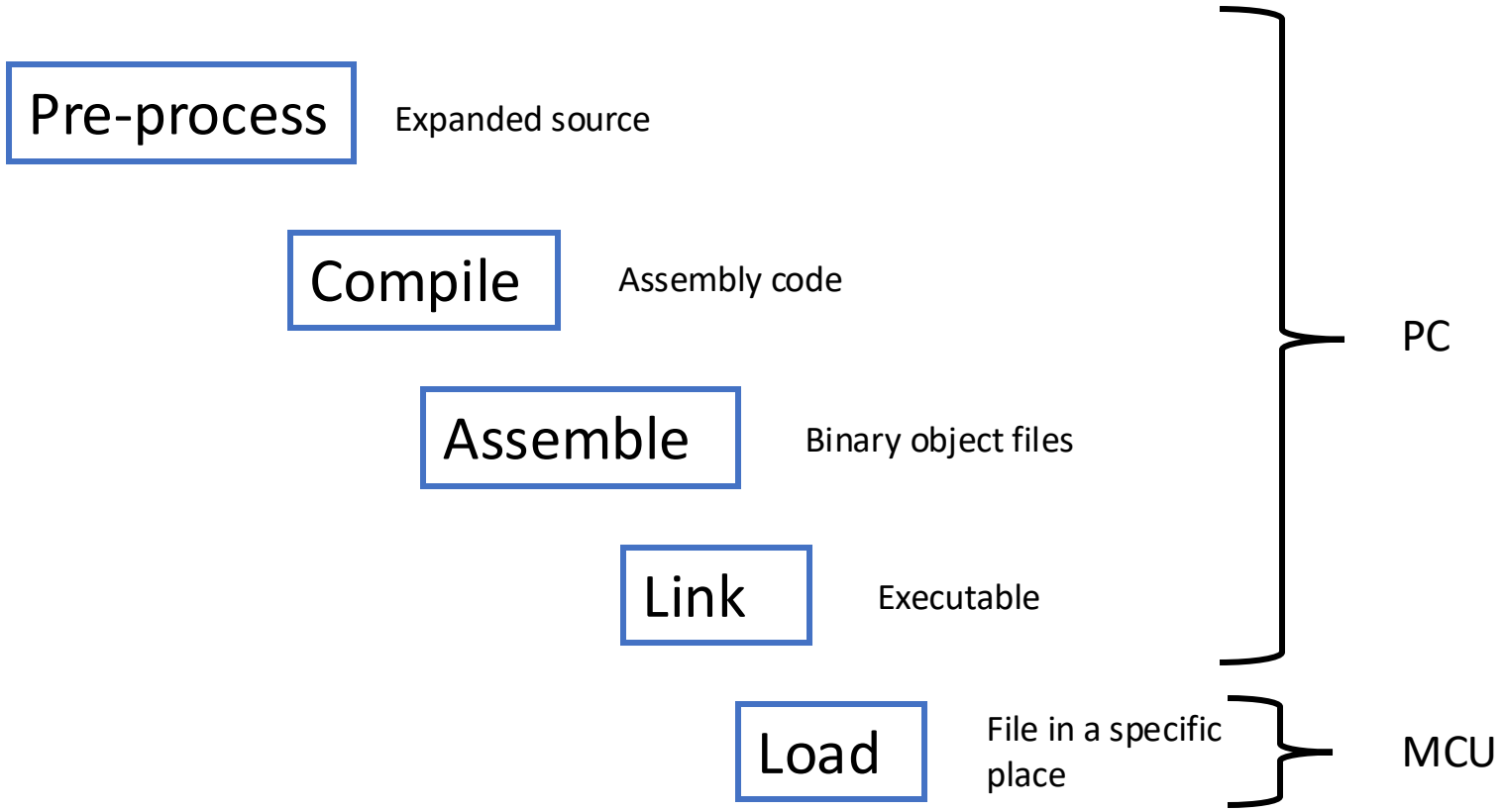
- Examples of how to create modules in C.
- To replicate it, check the document on canvas.

# Header and source files




https://stackoverflow.com/questions/5904530/how-do-header-and-source-files-in-c-work

# Compilation process in C



# C Preprocessor

- The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process.
- The Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.
- All preprocessor commands begin with a hash symbol (#).

# C Preprocessor

- The C Pre-processor searches for lines that begin with # and executes **macros** before calling the compiler.
- #define : replaces text with that name
- #include : includes source code
- #if, #elseif, #else, #endif : the code is compiled depending on conditions
- #ifdef, #ifndef : the code is compiled if a certain macro is defined

https://www.programiz.com/c-programming/c-preprocessor-macros

# #include

- #include “Myfile.h”
  Tells that the compiler should load all declarations found in the .h file before compiling the c file it is included with. Depending on where the file is located, the path may need to be specified,
  e.g. #include ”../Myfiles/ Myfile.h”
- #include <stdio.h>
  When the file is inside <>, the file is a **system one** and the compiler should know where to find it (no path is needed).

https://gcc.gnu.org/onlinedocs/cpp/Search-Path.html

# #define

- Defining **constants**:

  #define PI 3.14159

  All occurrences of PI are changed into 3.14159

- Defining **functions**:

  #define RADTODEG(x) ((x) * 57.29578)

  E.g. RADTODEG(1) is changed into (1 * 57.29578)

  Note: this function is not added to the stack! It becomes part of the executed code of the function where it appears.

  Disadvantage: being the code copied for each invocation of the function, the executable takes more memory.

# #ifdef

- An .h file can be imported multiple times in different files.
- Each time it is imported, the pre-processor re-copies it. This may lead to compilation errors.
- To avoid this, we can use #ifdef (“include guard”)

```
#ifndef DIGITALIO_H_
#define DIGITALIO_H_

void pinMode(int pinNumber, mode_definition mode); void digitalWrite(int pinNumber, int value);
#endif /* DIGITALIO_H_ */
```

https://en.wikipedia.org/wiki/Include_guard

# Live coding

- Example of how to use the pre-processor in C.
- To replicate it, check the document on canvas.

# Local vs global variables

- A variable declared *inside a function* definition is **local** and can only be used within the function

```
int func (void) {
   int local = 5;
   return local;
}
```

- A variable declared *outside a function* definition is **global**

```
int GLOBAL = 101;
```

- You can limit scope within a module with: {}

```
{
   int local = 1001;
}
```

# Static

The *static* keyword tells the compiler to reserve memory space outside the stack.

1. A **variable declared** *static* **within the body of a function** maintains its value between function invocations.
2. A **variable declared** *static* **within a module** (i.e. a file), but outside the body of a function, is accessible by all functions within that module. It is not accessible by functions within any other module. That is, it is a localized global.
3. **Functions declared** *static* within a module may only be called by other functions within that module. That is, the scope of the function is localized to the module within which it is declared.

https://www.geeksforgeeks.org/static-variables-in-c/ https://en.wikipedia.org/wiki/Static_(keyword)

# Extern

- A **variable** declared as *extern* in a module (c-file) means that the variable is **defined** in another module.
  - The memory space is thus reserved in another module.
  - But the compiler thus knows the type even though it does not reserve any memory space.
  - The variable is implicitly global.
  - If you initialise an extern variable, its memory will be allocated even if no definition is made elsewhere!
- When a **function** is declared, it is **by default extern** (even if not explicit).
  - In fact, functions are always global unless declared static.

https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/

https://en.wikipedia.org/wiki/External_variable

# To think about

- Think how would you use the keywords **static** and **extern**
- How can these keywords be associated to the concepts of “information hiding” (or abstraction) and “encapsulation”?

# Guidelines

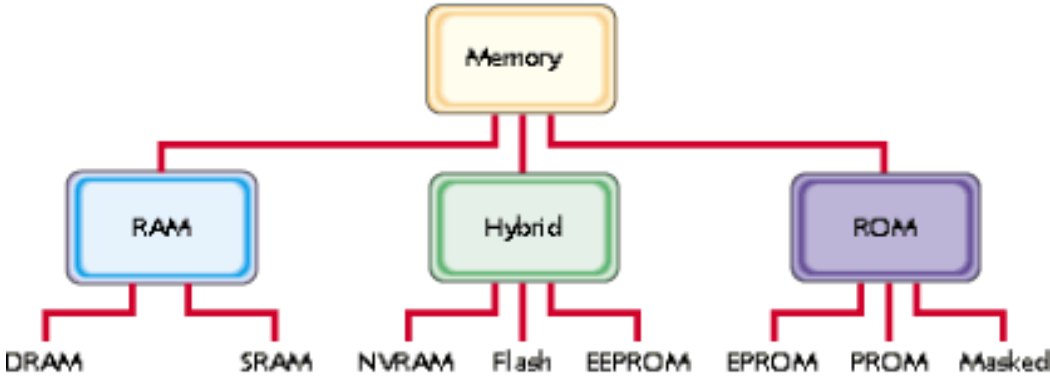
- A **header file only contains extern declarations of variables** — never static or unqualified variable definitions.
- For any given global variable, **only one header file declares it** (SPOT — Single Point of Truth).
- A source file never contains extern declarations of variables — source files always include the (sole) header that declares them.
- For any given variable, **exactly one source file defines the variable**, preferably initializing it too.
- The source file that defines the variable also includes the header to ensure that the definition and the declaration are consistent.
- A function should never need to declare a variable using extern.
- Avoid global variables whenever possible — use functions instead.

# Live coding

- Example of use of static and extern.
- To replicate it, check the document on canvas.

# Types of memory

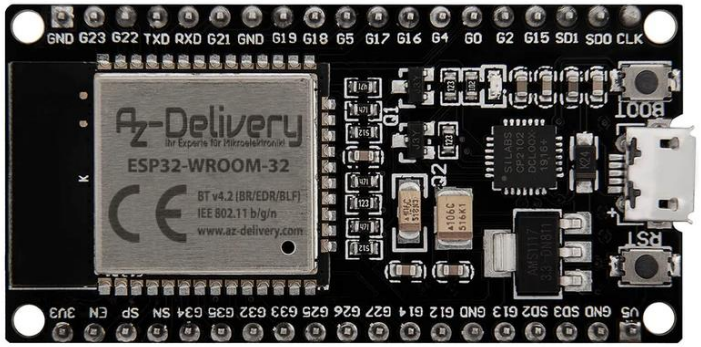
- (static) SRAM: memory is kept as long as power is on, expensive
- (dynamic) RAM: memory is lost every few ms and rewritten, cheaper
- Masked ROM: hardwired memory
- (programmable) PROM: writeable only once, keeps content without power
- (erasable and programmable) EPROM: can be deleted with ultraviolet
- (electrically-erasable-and-programmable) EEPROM: can be erased electrically, costly and slow
- Flash: r/w, but writes are slow, cheap
- (non-volatile) nvRAM: an SRAM with battery



| Type | Volatile? | Writeable? | Erase Size | Max Erase Cycles | Cost (per Byte) | Speed |
|---|---|---|---|---|---|---|
| SRAM | Yes | Yes | Byte | Unlimited | Expensive | Fast |
| DRAM | Yes | Yes | Byte | Unlimited | Moderate | Moderate |
| Masked ROM | No | No | n/a | n/a | Inexpensive | Fast |
| PROM | No | Once, with a device programmer | n/a | n/a | Moderate | Fast |
| EPROM | No | Yes, with a device programmer | Entire Chip | Limited (consult datasheet) | Moderate | Fast |
| EEPROM | No | Yes | Byte | Limited (consult datasheet) | Expensive | Fast to read, slow to erase/write |
| Flash | No | Yes | Sector | Limited (consult datasheet) | Moderate | Fast to read, slow to erase/write |
| NVRAM | No | Yes | Byte | Unlimited | Expensive (SRAM + battery) | Fast |

https://barrgroup.com/Embedded-Systems/How-To/Memory-Types-RAM-ROM-Flash

# ESP32




- Inside the chip:
  - SRAM: 520 Kbytes
  - 8 KBytes SRAM RTC SLOW Memory (accessible by co-processor in deep-sleep)
  - 8 KBytes SRAM RTC FAST Memory (accessible by the main CPU during boot from the deep-sleep)
  - ROM : 448 Kbytes (booting and core software for e.g. WiFi)
  - 1 Kbit of EFUSE (write only once, e.g. MAC address)
- External (our board):
  - Flash ROM: 4 Mbytes (stores application code + persistent data)

# Memory mapping (in the ESP32)

- Since the processor architecture is 32 bit, memory has 2^32 (4GB) address space.
- This includes all RAM and ROM (of different types) and peripherals (more on this later).

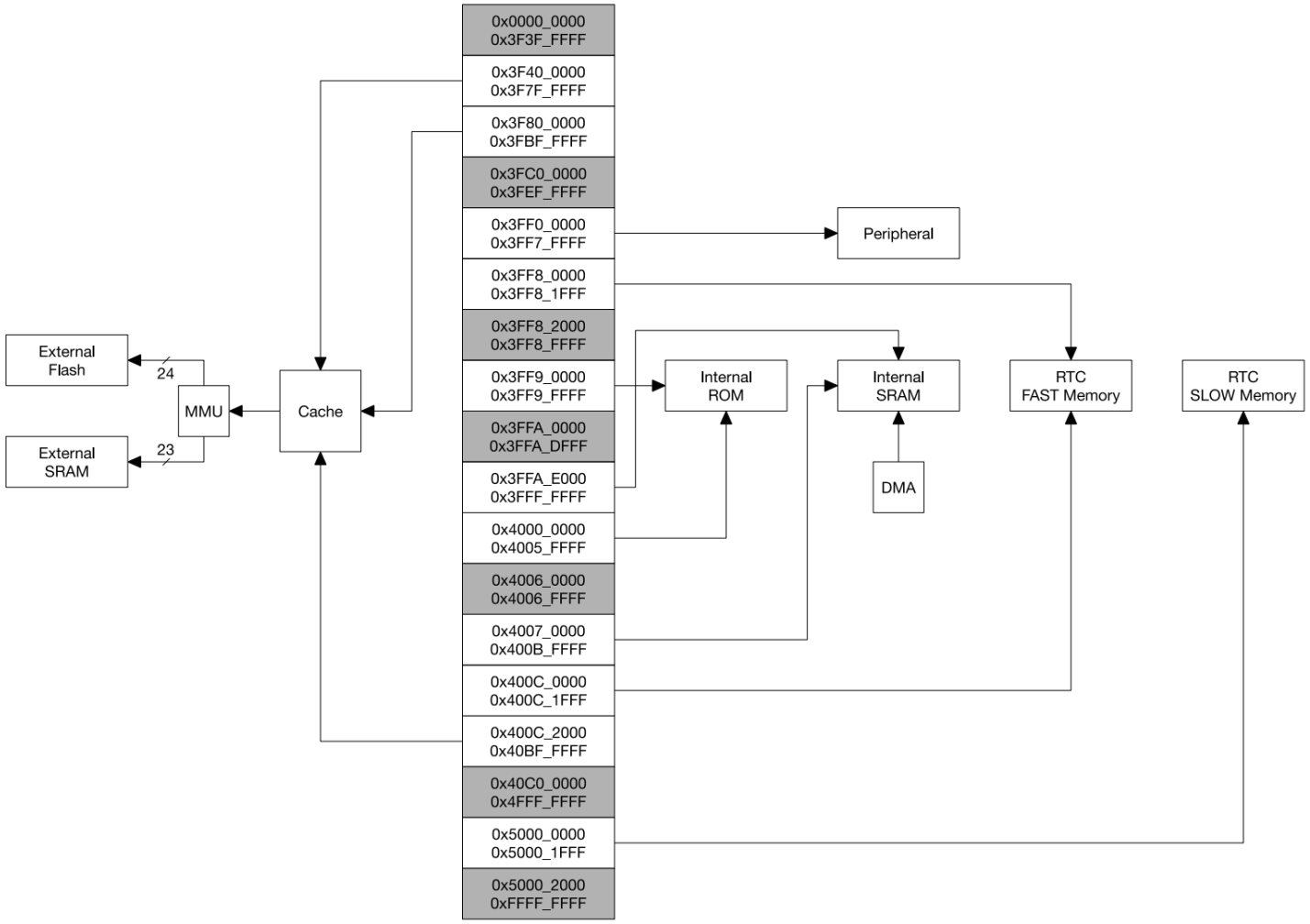


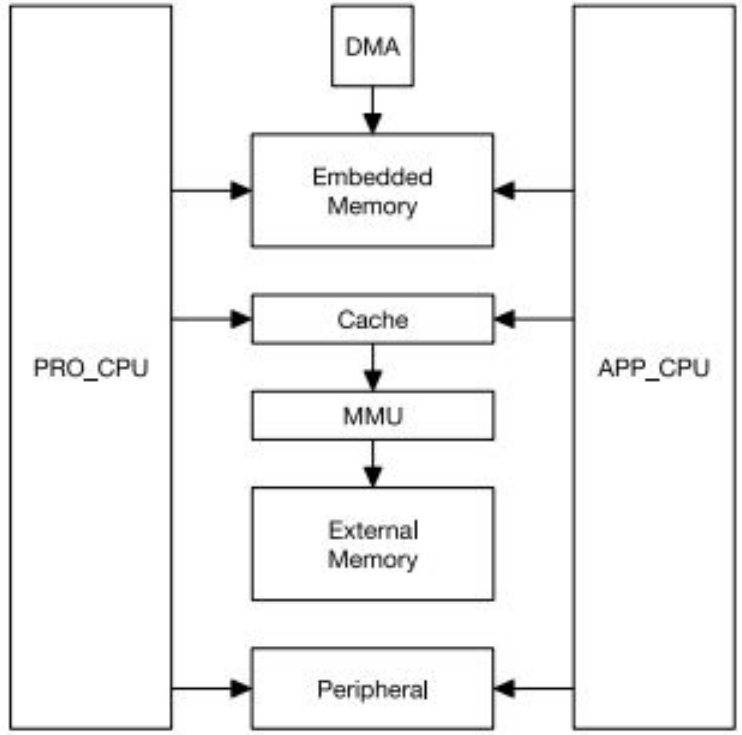
Table 4: Embedded Memory Address Mapping

| Bus Type | Boundary Address | | Size | Target | Comment |
|---|---|---|---|---|---|
| | Low Address | High Address | | | |
| Data | 0x3FF8_0000 | 0x3FF8_1FFF | 8 KB | RTC FAST Memory | PRO_CPU Only |
| | 0x3FF8_2000 | 0x3FF8_FFFF | 56 KB | Reserved | - |
| Data | 0x3FF9_0000 | 0x3FF9_FFFF | 64 KB | Internal ROM 1 | - |
| | 0x3FFA_0000 | 0x3FFA_DFFF | 56 KB | Reserved | - |
| Data | 0x3FFA_E000 | 0x3FFD_FFFF | 200 KB | Internal SRAM 2 | DMA |
| Data | 0x3FFE_0000 | 0x3FFF_FFFF | 128 KB | Internal SRAM 1 | DMA |
| Bus Type | Boundary Address | | Size | Target | Comment |
| | Low Address | High Address | | | |
| Instruction | 0x4000_0000 | 0x4000_7FFF | 32 KB | Internal ROM 0 | Remap |
| Instruction | 0x4000_8000 | 0x4005_FFFF | 352 KB | Internal ROM 0 | - |
| | 0x4006_0000 | 0x4006_FFFF | 64 KB | Reserved | - |
| Instruction | 0x4007_0000 | 0x4007_FFFF | 64 KB | Internal SRAM 0 | Cache |
| Instruction | 0x4008_0000 | 0x4009_FFFF | 128 KB | Internal SRAM 0 | - |
| Instruction | 0x400A_0000 | 0x400A_FFFF | 64 KB | Internal SRAM 1 | - |
| Instruction | 0x400B_0000 | 0x400B_7FFF | 32 KB | Internal SRAM 1 | Remap |
| Instruction | 0x400B_8000 | 0x400B_FFFF | 32 KB | Internal SRAM 1 | - |
| Instruction | 0x400C_0000 | 0x400C_1FFF | 8 KB | RTC FAST Memory | PRO_CPU Only |
| Bus Type | Boundary Address | | Size | Target | Comment |
| | Low Address | High Address | | | |
| Data Instruction | 0x5000_0000 | 0x5000_1FFF | 8 KB | RTC SLOW Memory | - |

# ESP32 memory management

- Both CPUs have DMA, caches and MMU.
- They both see the same memory space (by default).

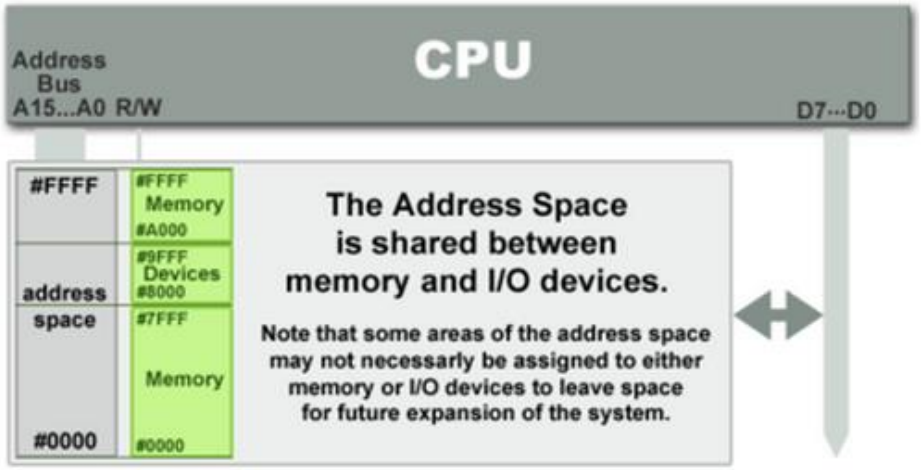
- A **memory management unit** (MMU) is a hardware unit having all memory references passed through itself, primarily performing the translation of virtual memory addresses to physical addresses.

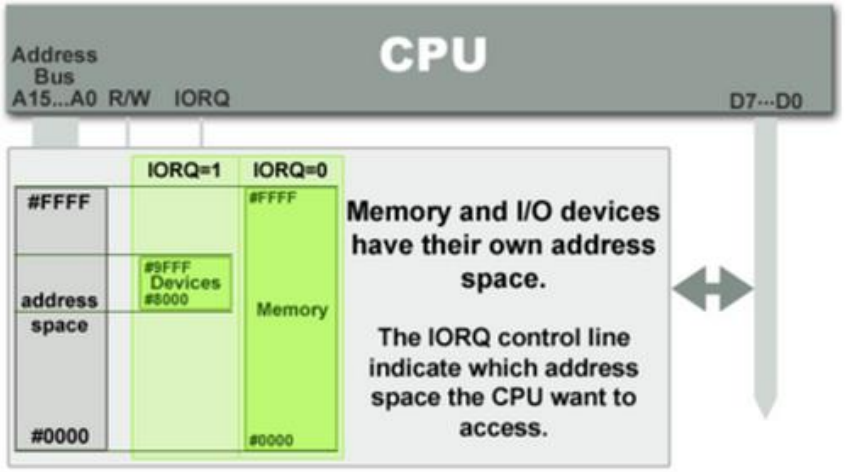



https://developer.arm.com/architectures/learn-the-architecture/memory-management/the-memory-management-unit-mmu

# IO on MCUs

- Microprocessors normally use two methods to connect external devices: **memory mapped** or **port mapped** I/O.
  - **Memory mapped** I/O is mapped into the same address space as program memory and/or user memory, and is accessed in the same way.
  - **Port mapped** I/O uses a separate, dedicated address space and is accessed via a dedicated set of microprocessor instructions.



# Peripherals on the ESP32

- The ESP32 has 41 memory mapped peripherals.

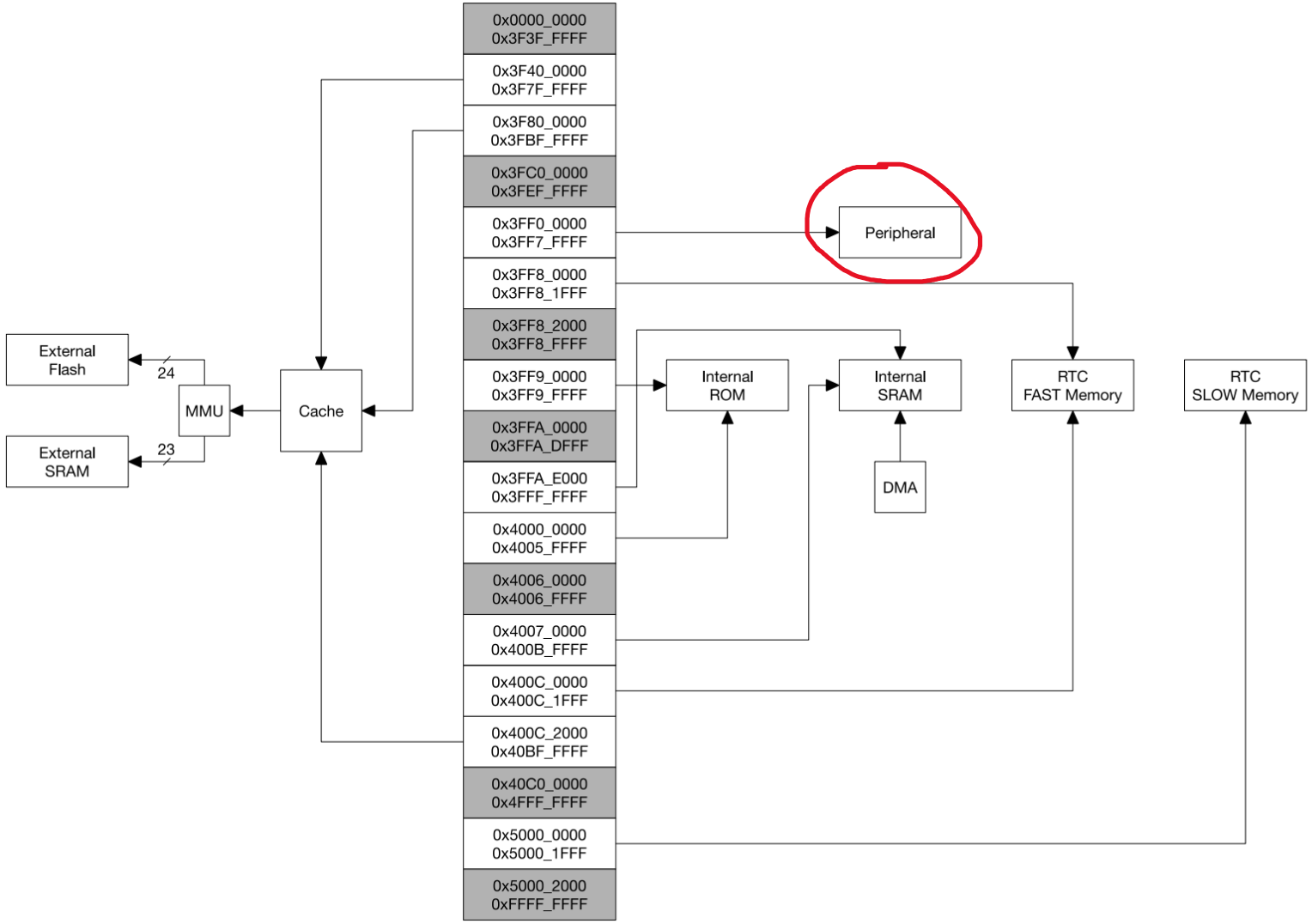


**Table 8: Peripheral Address Mapping**

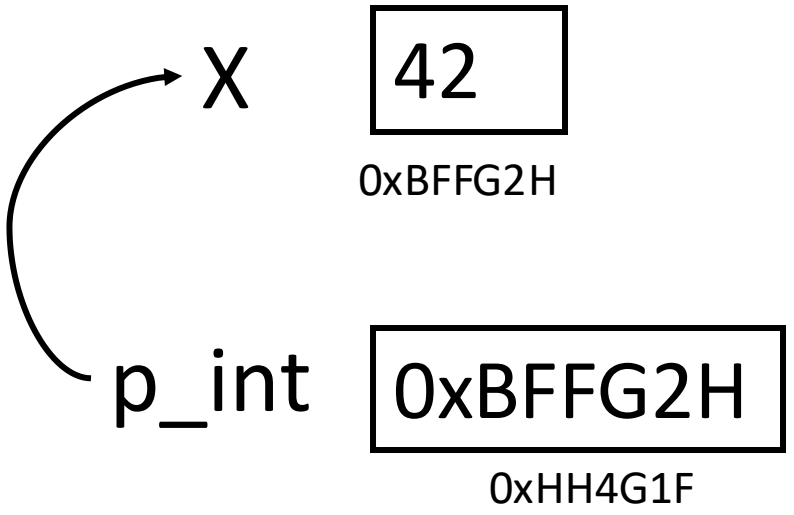
| Bus Type | Boundary Address | | Size | Target | Comment |
|---|---|---|---|---|---|
| | Low Address | High Address | | | |
| Data | 0x3FF0_0000 | 0x3FF0_0FFF | 4 KB | DPort Register | |
| Data | 0x3FF0_1000 | 0x3FF0_1FFF | 4 KB | AES Accelerator | |
| Data | 0x3FF0_2000 | 0x3FF0_2FFF | 4 KB | RSA Accelerator | |
| Data | 0x3FF0_3000 | 0x3FF0_3FFF | 4 KB | SHA Accelerator | |
| Data | 0x3FF0_4000 | 0x3FF0_4FFF | 4 KB | Secure Boot | |
| | 0x3FF0_5000 | 0x3FF0_FFFF | 44 KB | Reserved | |
| Data | 0x3FF1_0000 | 0x3FF1_3FFF | 16 KB | Cache MMU Table | |
| | 0x3FF1_4000 | 0x3FF1_EFFF | 44 KB | Reserved | |
| Data | 0x3FF1_F000 | 0x3FF1_FFFF | 4 KB | PID Controller | Per-CPU peripheral |
| | 0x3FF2_0000 | 0x3FF3_FFFF | 128 KB | Reserved | |
| Data | 0x3FF4_0000 | 0x3FF4_0FFF | 4 KB | UART0 | |
| | 0x3FF4_1000 | 0x3FF4_1FFF | 4 KB | Reserved | |
| Data | 0x3FF4_2000 | 0x3FF4_2FFF | 4 KB | SPI1 | |
| Data | 0x3FF4_3000 | 0x3FF4_3FFF | 4 KB | SPI0 | |
| Data | 0x3FF4_4000 | 0x3FF4_4FFF | 4 KB | GPIO | |
| | 0x3FF4_5000 | 0x3FF4_7FFF | 12 KB | Reserved | |
| Data | 0x3FF4_8000 | 0x3FF4_8FFF | 4 KB | RTC | |
| Data | 0x3FF4_9000 | 0x3FF4_9FFF | 4 KB | IO MUX | |
| | 0x3FF4_A000 | 0x3FF4_AFFF | 4 KB | Reserved | |
| Data | 0x3FF4_B000 | 0x3FF4_BFFF | 4 KB | SDIO Slave | One of three parts |
| Data | 0x3FF4_C000 | 0x3FF4_CFFF | 4 KB | UDMA1 | |
| | 0x3FF4_D000 | 0x3FF4_EFFF | 8 KB | Reserved | |
| Data | 0x3FF4_F000 | 0x3FF4_FFFF | 4 KB | I2S0 | |
| Data | 0x3FF5_0000 | 0x3FF5_0FFF | 4 KB | UART1 | |

| Bus Type | Boundary Address | | Size | Target | Comment |
|---|---|---|---|---|---|
| | Low Address | High Address | | | |
| | 0x3FF5_1000 | 0x3FF5_2FFF | 8 KB | Reserved | |
| Data | 0x3FF5_3000 | 0x3FF5_3FFF | 4 KB | I2C0 | |
| Data | 0x3FF5_4000 | 0x3FF5_4FFF | 4 KB | UDMA0 | |
| Data | 0x3FF5_5000 | 0x3FF5_5FFF | 4 KB | SDIO Slave | One of three parts |
| Data | 0x3FF5_6000 | 0x3FF5_6FFF | 4 KB | RMT | |
| Data | 0x3FF5_7000 | 0x3FF5_7FFF | 4 KB | PCNT | |
| Data | 0x3FF5_8000 | 0x3FF5_8FFF | 4 KB | SDIO Slave | One of three parts |
| Data | 0x3FF5_9000 | 0x3FF5_9FFF | 4 KB | LED PWM | |
| Data | 0x3FF5_A000 | 0x3FF5_AFFF | 4 KB | Efuse Controller | |
| Data | 0x3FF5_B000 | 0x3FF5_BFFF | 4 KB | Flash Encryption | |
| | 0x3FF5_C000 | 0x3FF5_DFFF | 8 KB | Reserved | |
| Data | 0x3FF5_E000 | 0x3FF5_EFFF | 4 KB | PWM0 | |
| Data | 0x3FF5_F000 | 0x3FF5_FFFF | 4 KB | TIMG0 | |
| Data | 0x3FF6_0000 | 0x3FF6_0FFF | 4 KB | TIMG1 | |
| | 0x3FF6_1000 | 0x3FF6_3FFF | 12 KB | Reserved | |
| Data | 0x3FF6_4000 | 0x3FF6_4FFF | 4 KB | SPI2 | |
| Data | 0x3FF6_5000 | 0x3FF6_5FFF | 4 KB | SPI3 | |
| Data | 0x3FF6_6000 | 0x3FF6_6FFF | 4 KB | SYSCON | |
| Data | 0x3FF6_7000 | 0x3FF6_7FFF | 4 KB | I2C1 | |
| Data | 0x3FF6_8000 | 0x3FF6_8FFF | 4 KB | SDMMC | |
| Data | 0x3FF6_9000 | 0x3FF6_AFFF | 8 KB | EMAC | |
| | 0x3FF6_B000 | 0x3FF6_BFFF | 4 KB | Reserved | |
| Data | 0x3FF6_C000 | 0x3FF6_CFFF | 4 KB | PWM1 | |
| Data | 0x3FF6_D000 | 0x3FF6_DFFF | 4 KB | I2S1 | |
| Data | 0x3FF6_E000 | 0x3FF6_EFFF | 4 KB | UART2 | |
| Data | 0x3FF6_F000 | 0x3FF6_FFFF | 4 KB | PWM2 | |
| Data | 0x3FF7_0000 | 0x3FF7_0FFF | 4 KB | PWM3 | |
| | 0x3FF7_1000 | 0x3FF7_4FFF | 16 KB | Reserved | |
| Data | 0x3FF7_5000 | 0x3FF7_5FFF | 4 KB | RNG | |
| | 0x3FF7_6000 | 0x3FF7_FFFF | 40 KB | Reserved | |

# How do you access memory in C?

- Pointers in C

```
int main(void) {
   int x;
   int *p_int;
   x = 42;
   p_int = &x;
}
```




What happens if we do *p_int = 43* ?

# Pointers are associated to memory locations

- Example: the register that controls the ADC is mapped to memory location 0x400C0000

- A pointer to that location can be declared:
  `uint32_t *const p_ADC_CR = (uint32_t *) 0x400C0000U;`

  Type is a pointer to an uint32 and is supposed to not change

  Name of the pointer

  Type casting

  Unsigned

https://www.tutorialspoint.com/cprogramming/c_constants

# Setting registers with pointers

- Example: to start the analog-to-digital conversion you need to set the second bit (from the last one) to 1

**43.7.1 ADC Control Register**

**Name:** ADC_CR

**Address:** 0x400C0000

**Access:** Write-only

| 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 |
| – | – | – | – | – | – | START | SWRST |

- **SWRST: Software Reset**

0 = No effect.

1 = Resets the ADC simulating a hardware reset.

- **START: Start Conversion**

0 = No effect.

1 = Begins analog-to-digital conversion.

```
/* 0000…000010 */
#define STARTADC 0x0002U

/* set the register */
*p_ADC_CR = STARTADC;
```

# Live coding

- Allocating a pointer to a memory position.
- Try this at home! See example code on Canvas.

# Bitwise operators

- & AND
  - The & operator compares each binary digit of two integers and returns a new integer, with a 1 wherever both numbers had a 1 and a 0 anywhere else.
  - 10101010 & 00001111 -> 00001010
- | (inclusive) OR
  - Returns a new value with 1 wherever 1 exists in either one or the other operand.
  - 10101010 | 00001111 -> 10101111
- ^ (exclusive) XOR
  - Copies the bit if it is set in one operand but **not** both.
  - 10101010 ^ 00001111 -> 10100101

https://www.tutorialspoint.com/cprogramming/c_bitwise_operators

# Bitwise operators

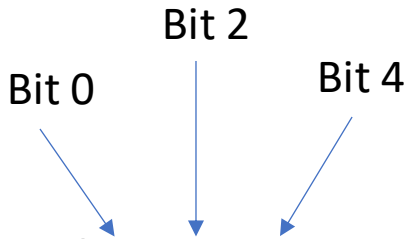
- ~ NOT (One’s complement)
  - Has the effect of 'flipping' bits.
  - ~ 10101010 -> 01010101
- << Left shift
  - The left operands value is moved left by the number of bits specified by the right operand.
  - 10101010 ^ 1 -> 01010100 (bits are moved 1 position to the left and a 0 is added at the end)
- >> Right shift
  - The left operands value is moved right by the number of bits specified by the right operand.
  - 00001111 ^ 2 -> 00000011 (bits are moved 2 positions to the right and two 0s are added at the beginning)

# Masking

- A **mask** defines which bits you want to keep or change, and which bits you want to ignore.
- **Masking** is the act of applying a mask to a value. Using a mask and a bitwise operator you can:
  - **Extract** a subset of the bits in the value with AND
  - **Set** a subset of the bits in the value with OR
  - **Toggle** a subset of the bits in the value with XOR

https://stackoverflow.com/questions/10493411/what-is-bit-masking

# Masking: extract




- Suppose you want to know if bits 2 and 4 of 00**0**0**1**111 are set (=1)
  - Note: we call the first bit, bit zero
  - Mask: 00**1**0**1**000
  - Operation: 00**0**0**1**111 & 00**1**0**1**000 -> 00001000
  - if (00**0**0**1**111 & 00**1**0**1**000) …
    is true if ANY of the bits are set
  - if (00**0**0**1**111 & 00**1**0**1**000 == 00**1**0**1**000) …
    is true if ALL the bits are set

- Question: what mask do we need if we want to know if bit 0 is set?

# Masking: extract (2)

- Suppose you want to know if bit 4 of 00001111 is 1
  - 00001111 >> 4 -> 00000001
    We “push” the 4th bit to the end of the byte
  - 00000001 & 00000001
    Then we compare it against a mask with one bit only at the end
  - (00001111 >> 4) & 00000001
  - Note: you can substitute 4 with any other number

# Masking: set

- Suppose you want to set bits 2 and 4 to 1 of 00**0**0**1**111 (keeping all the others as they are)
  - Mask: 00**1**0**1**000
  - 00**0**0**1**111 | 00**1**0**1**000 -> 00101111
    all ones of the original value are copied and bits 2 and 4 set to 1 regardless of what was set in the original value

- Now let's set those bits to 0!
  - 00001111 & 11**0**1**0**111 -> 00000111
  - Which is equivalent to: 00001111 & ~00101000 (the one's complement of the mask)

# Masking: set (2)

- Alternatively, we can use the OR with a simple mask left shifted N positions.
- Supposing we want to set the bit 4 to 1
  - 00001111 | (00000001 << 3)

# Masking: toggle

- Suppose you want to toggle bits 2 and 4 to 1 of 00**0**0**1**111 (keeping all the others as they are)
  - Mask: 00101000
  - 00001111 ^ 00101000 -> 00**1**0**0**111
    all bits except the bits 1 and 4 are copied as they are, but bit 1 and 4 are flipped

- You can toggle bit 4 with a shift plus a XOR
  - 00001111 ^ (00000001 << 4) -> 00101111

# Live coding

- Examples of bit masking.
- Try this at home! See example code on Canvas.

# Bit fields

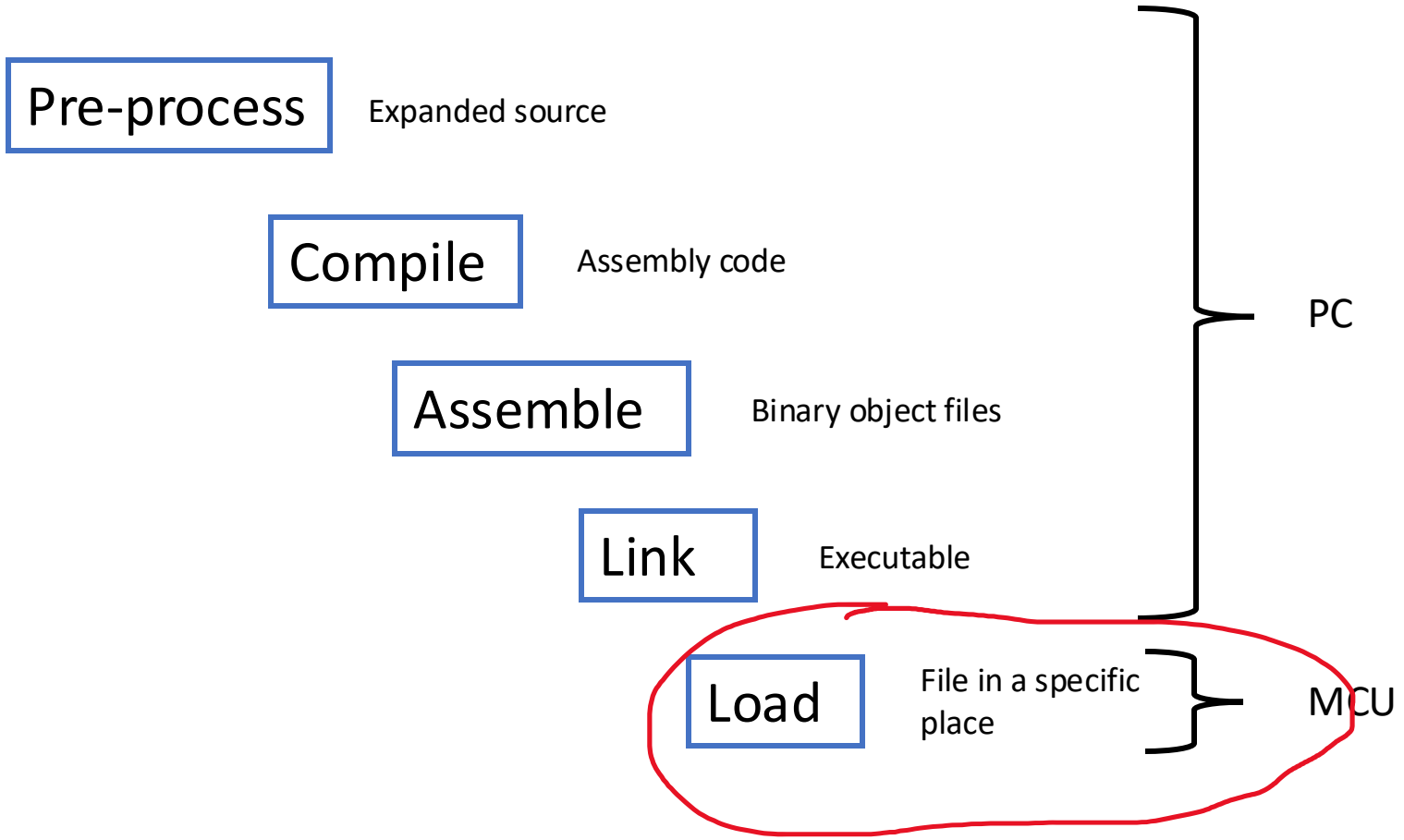
- C allows specifying the number of bits a field in a structure takes. Example:

```
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
    } status;
```

# Live coding

- Try this at home! See example code on Canvas.

# Compilation process in C



# Loading the code on a MCU

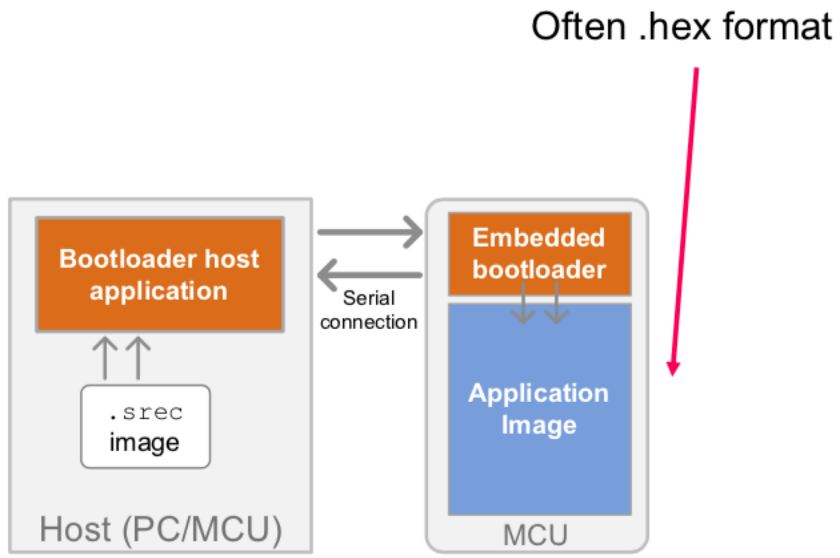
- During development you can usually program the microcontroller with a **dedicated hardware** (e.g. JTAG).
  - This can stop the processor and write the code into some rewriteable ROM.
  - The method for doing that is chip-dependent, but it usually requires some special debugging widget.

- If you cannot write directly into the ROM, you need **a piece of code** that reads the code from a communication mean (e.g. serial port) and loads into the ROM (the *bootloader*).
  - Some processor have a built-in bootloader.
  - In other cases, you need to write your own bootloader.

- Code can be executed from ROM directly, or may be copied onto RAM first.

# Bootloader

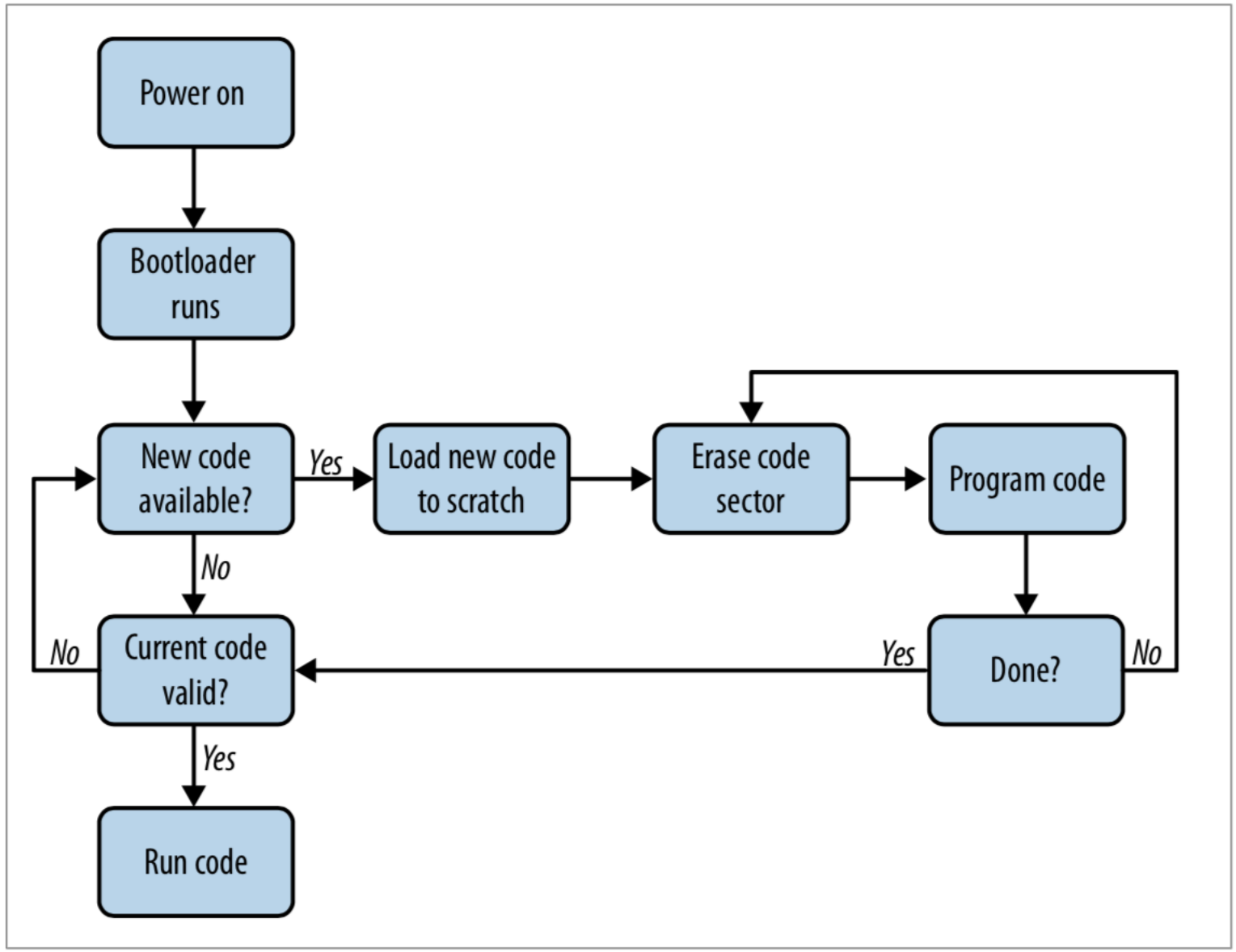
- A ”bootloader” serves the only purpose of loading an application image (firmware) intc the MCU’s internal flash memory or EEPROM and start the execution.
- Bootloaders should occupy little memory and should use simple communication protocols (e.g. serial).
- More sophisticated approaches can make use of wired and wireless networks (over-the-air updates).



# Things can go wrong

- New code can be corrupted
  - -> it is not a good idea to just load the new code onto ROM as it comes.

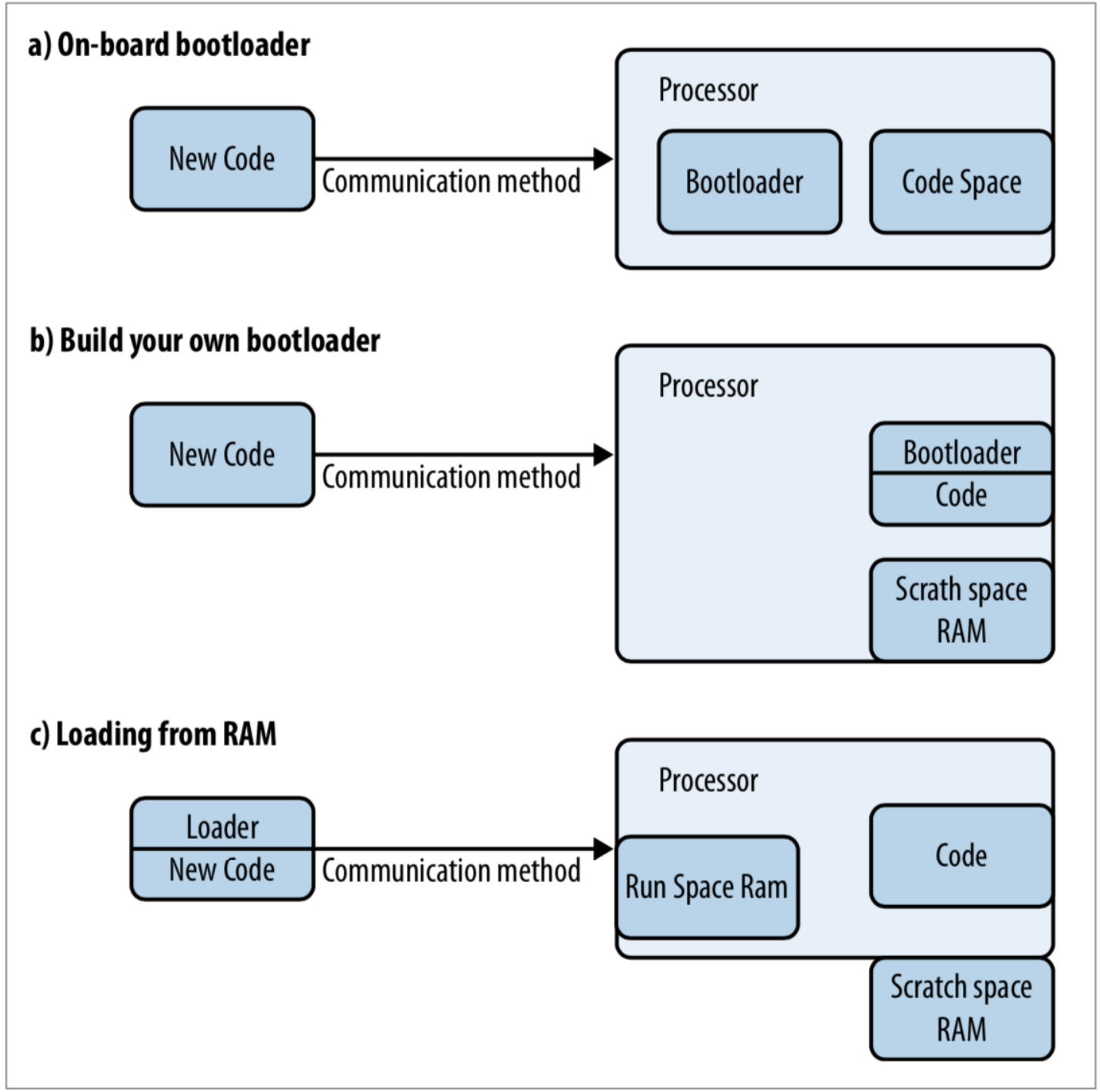
- It may be better to first verify it (e.g. computing a checksum).
- You need a “scratch space” for loading the code and checking.
  - Scratch space space must be > code size.
    - Can be in RAM (ideally) or ROM.
  - If code is OK -> then overwrite old code on ROM.
  - If code is not OK -> keep old code.


Power on
Bootloader runs
New code available?
Yes
Load new code to scratch
Erase code sector
Program code
No
Current code valid?
No
Yes
Done?
No
Yes
Run code

# Updating the bootloader

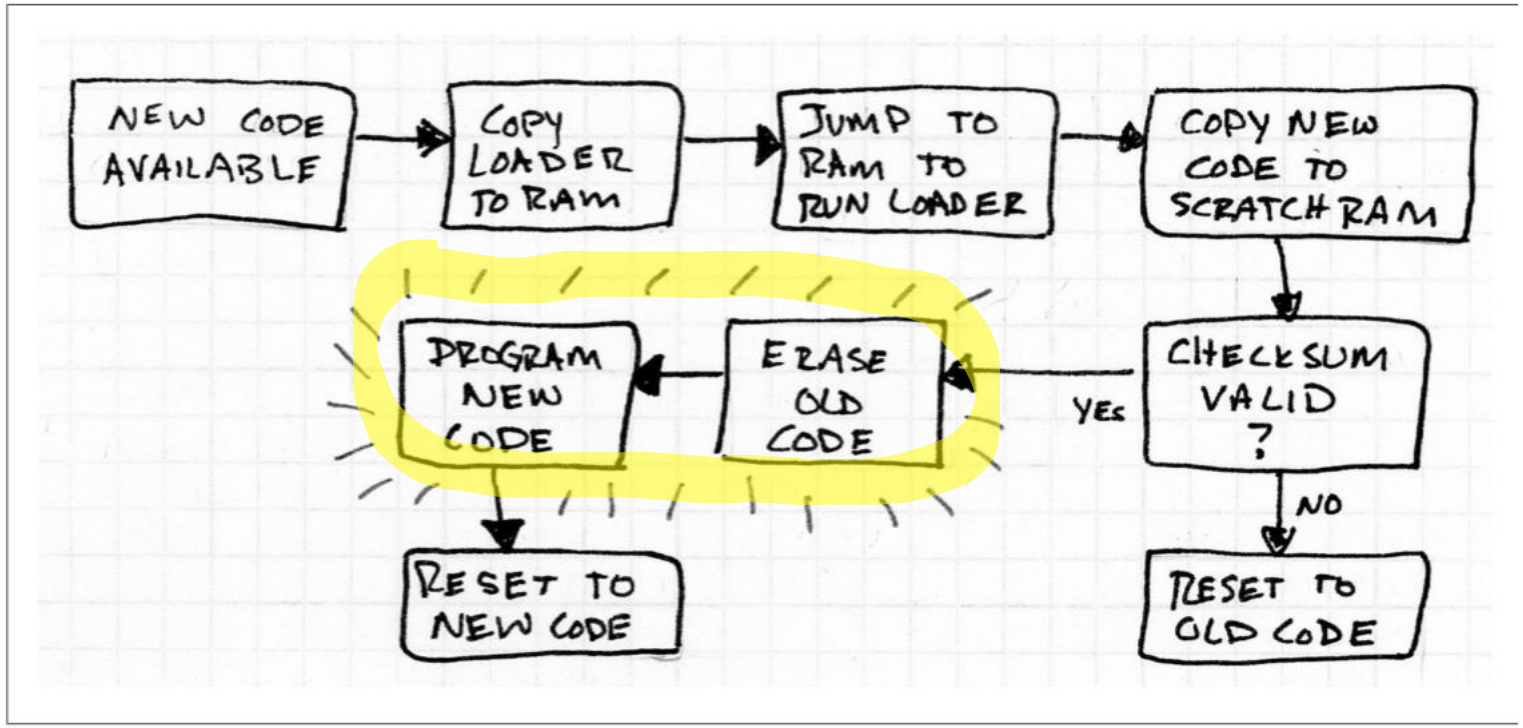
- If your bootloader deletes the ROM it is executed from, it will delete itself and be corrupted!
  - -> You need to **run the bootloader from RAM** instead.
  - But do not corrupt currently used RAM! (at least not the part that is needed to load the new bootloader)
  - The linker can be configured to specify what portions go onto RAM and what onto ROM.

1. Copy the loader from the code storage to RAM.
2. Run the loader from RAM.
3. Copy the new code (with loader) to a scratch area.
4. Verify new code.
5. Erase the old code and program the new code.
6. Reset the processor to run the new code.


a) On-board bootloader
New Code
Communication method
Processor
Bootloader
Code Space
b) Build your own bootloader
New Code
Communication method
Processor
Bootloader
Code
Scrath space
RAM
c) Loading from RAM
Loader
New Code
Communication method
Processor
Run Space Ram
Code
Scratch space
RAM

# Weak points

- If the the power goes down during updating of the bootloader section -> system corrupted forever (bricked).
  - -> better having small bootloaders!



# Security

- Many chips offer **code read protection**. Once it is turned on, on-chip memory becomes unreadable from debuggers (or code itself).
- The new code should be encrypted when sent and decrypted by the bootloader to verify authenticity.
- The bootloader can decrypt the code and place it onto RAM.
- If you don’t have space, you can encrypt only parts of the new code.

- No system is perfect! Most MCUs are sensitive to glitched power supplies that produce behaviours the manufacturer did not properly consider.

https://limitedresults.com/2019/11/pwn-the-esp32-forever-flash-encryption-and-sec-boot-keys-extraction/

# To think about…

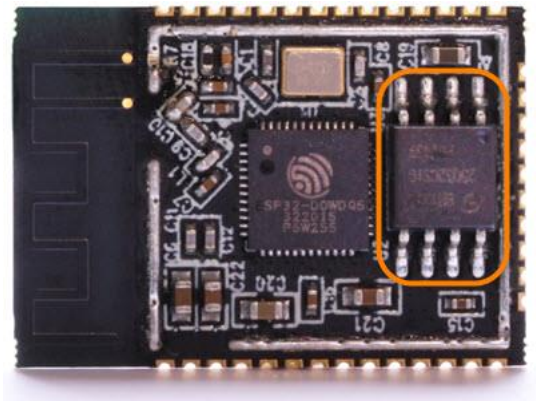
- How can you load a firmware on a connected device, e.g., through Wifi or Bluetooth? Imagine the required sequence of steps.
- Where would you put the code that fetches the new firmware from network, in the application or the bootloader?

# Bootloader on the ESP32

- In the esp32 ROM memory there's a small program, named *first-stage bootloader*.
- This program is executed at each reset of the chip: it configures the access to the external flash memory and, if required, stores on it new data coming from the serial/USB port (*flash* process).
- Once finished, it accesses the flash memory (at address 0x1000) and loads and executes the *second-stage bootloader*.

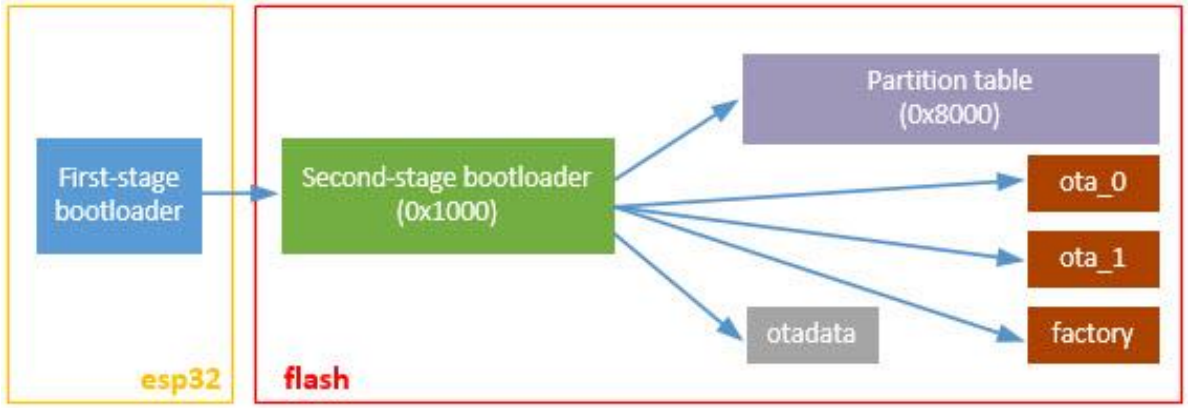
# Flash memory partitions on ESP32

- The esp32 chip requires an **external** flash memory to store programs, data, configuration parameters…
- The external memory is connected to the chip via the **SPI** bus and the supported capacity is up to **16Mb**.
- The flash memory can store different elements: programs, data… hence it's divided into **sections** (*partitions*).
- The list of partitions, their size and position within the flash memory is stored in the memory itself (at address 0x8000) and it's called **partition table**.
- Two partition types are defined by *default*:
  - **app** (type 0) – partition that contains an application (program)
  - **data** (type 1) – partition that contains data



# Bootloader on the ESP32

- The *second-stage bootloader* reads the *partition table* at address 0x8000 and searches for **app** partitions.
- It decides which application has to be executed based on the content of the **otadata** partition: if this partition is empty or doesn't exist, the bootloaded executes the application stored in the **factory** partition.
- This allows to implement an **over-the-air** (OTA) application update process: your code downloads the new version of your application (e.g. from web) and stores it in a new app partition
- Once the upload is completed, the id of the partition is saved in otadata and the chip is rebooted; the bootloader will execute the new version.



# Example exam questions

- What is the difference between *declaring* and *defining* a variable in C?
  - And a function?
- What is a header file used for in C?
- What are the steps needed to compile a piece of code from source to executable?
- What is the role of the C pre-processor?
- What are typical commands used in a C pre-processor?
- What are *include guards* and what are they used for?
- How can one access IO peripherals from C code?
- Can one access a memory location directly in C code?
- What bitwise operations are useful in C to manipulate the content of a register?
- How do you set the 2$^{nd}$ bit from the right to 1 of a pointer to a u_int8_t ?
- How can you define a struct where the first two fields, named A and B, take 1 bit each and the third, C, takes 6 bits?
- What is a Bootloader? Why is it needed?
- How do you avoid loading a corrupted firmware?
- How does bootloading work on an ESP32?