Embedded Systems
Lecture 3.
Hardware Software Architecture and Software Dev.
Memory, Clock, GPIOs.

© Lothar Thiele

Computer Engineering and Networks Laboratory

Michele Magno

D-ITET center for project based learning

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Source the Slides or adaptation from:
Embedded Systems. P. Marwedel

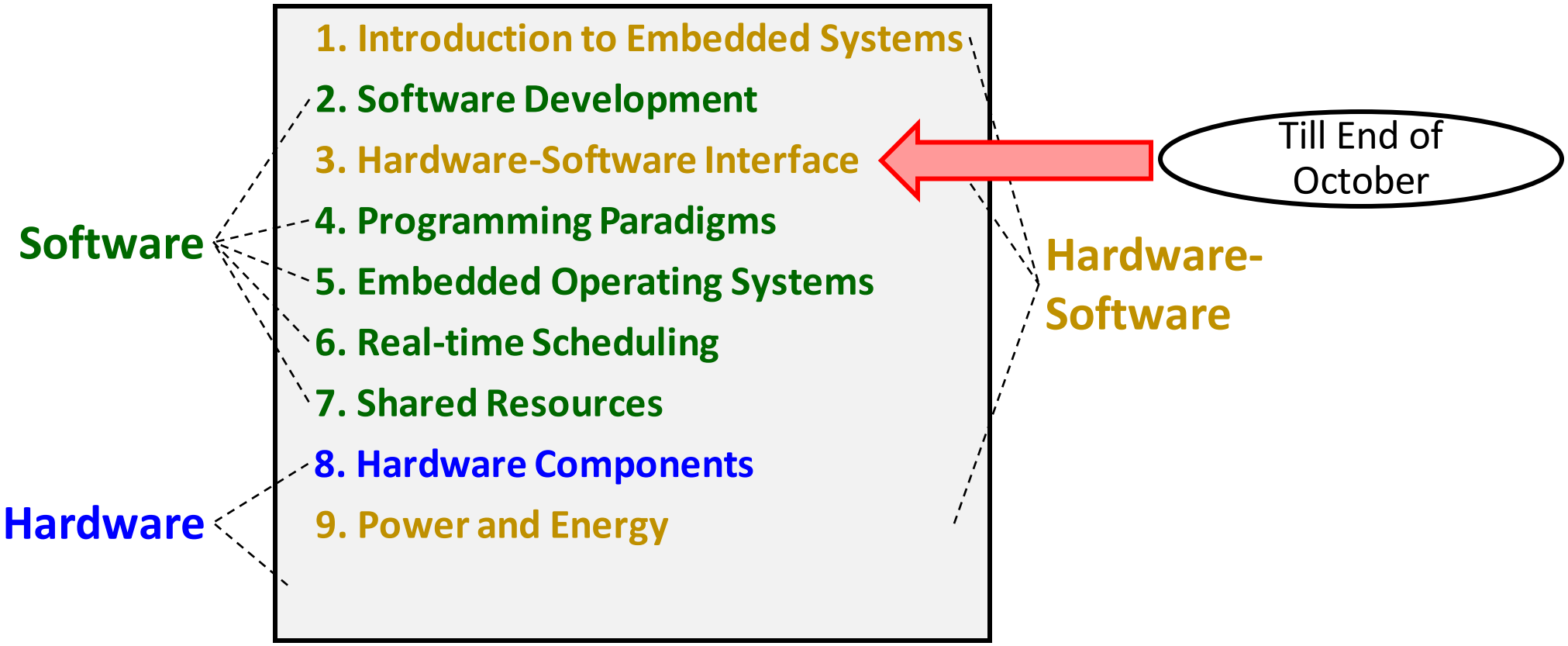
Exams information.

- 2h written exam
  - One in February and one in August.
- They will be 3 main parts.
  - Energy and latency linked with hardware-software interface
  - Real time and scheduling
  - Architecture Synthesis and systems overview.
  - Labs will be not explicitly part of the exam but can help to understand better the questions and perform.
- Exercise
  - By the 31.10 home work with some more exercise will be provided as home work.
  - By end of year more home work will be given for practice before the exam
  - Previous years exams are useful for training
    - Especially last year one.

Do you Remember ?

- Variety of possibility for embedded processing
- CPU has an instruction set.
  - Different architecture according to the bus
- According to the architecture we can run more fast or less fast a task
- **Energy is different than power Possible exam exercise!**
  - **Different architecture for different execution time**
  - **Frequency affect the time and the energy**
  - **Parallel processing can affect both**
- ARM processors
  - What we will use as commodity platform for the lab?

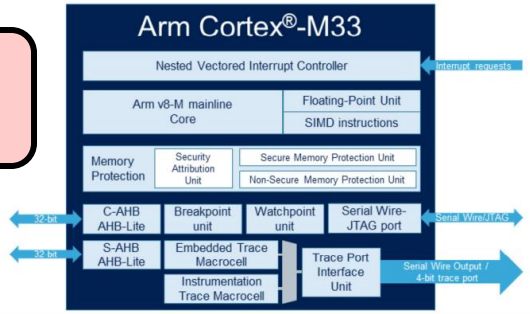
Where we are …


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

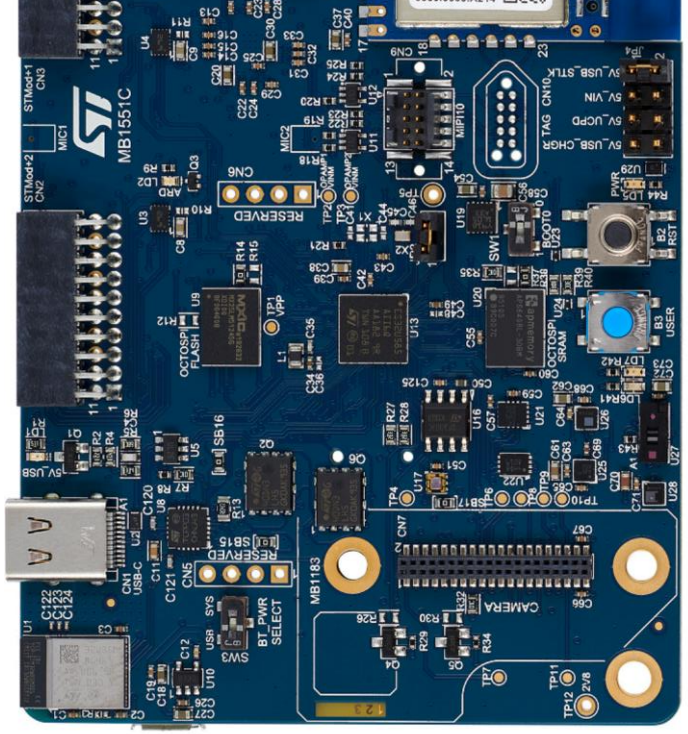
A recent architecture from ARM – CortexM33.

Cortex®-M33 processor overview

- Armv8-M architecture
- Harvard architecture, 3-stage pipeline
  - Floating-Point Unit
  - Single Instruction Multiple Data Instructions
- Memory Protection
  - Security Attribution Unit (SAU)
  - 2x Memory Protection Units (MPUs)
- Integrated Nested Vectored Interrupt Controller (NVIC)
- Invasive and non-invasive debug



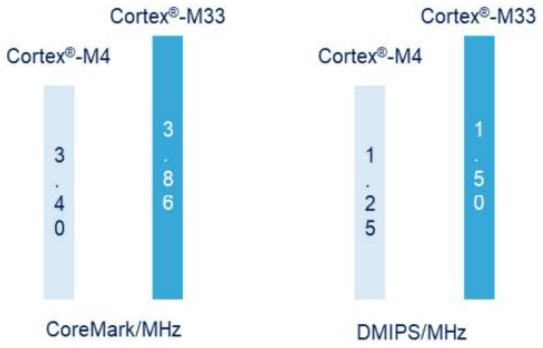
Energy efficient	Software productivity	System security
Most energy efficient 32-bit embedded processor for MCU+DSP requirements	SIMD instructions accelerate data signal processing	TrustZone® for Armv8-M



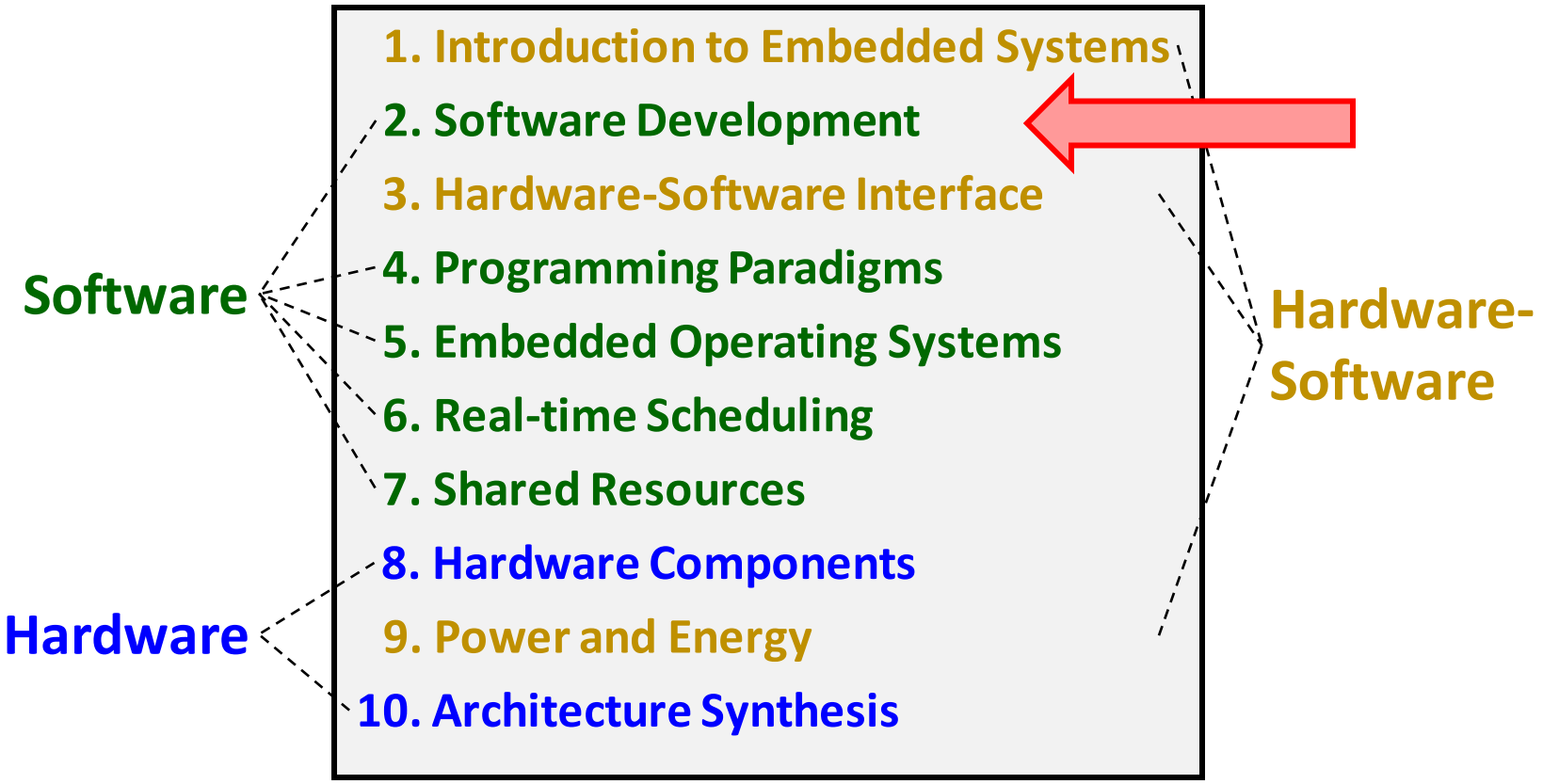
- Seamless architecture across all applications

Cortex-M0 & M0+	Cortex-M3	Cortex-M4	Cortex-M33	Cortex-M7
Ultra low power	First Cortex®-M CPU	High performance		

Binary and tool compatible → Highest Performance

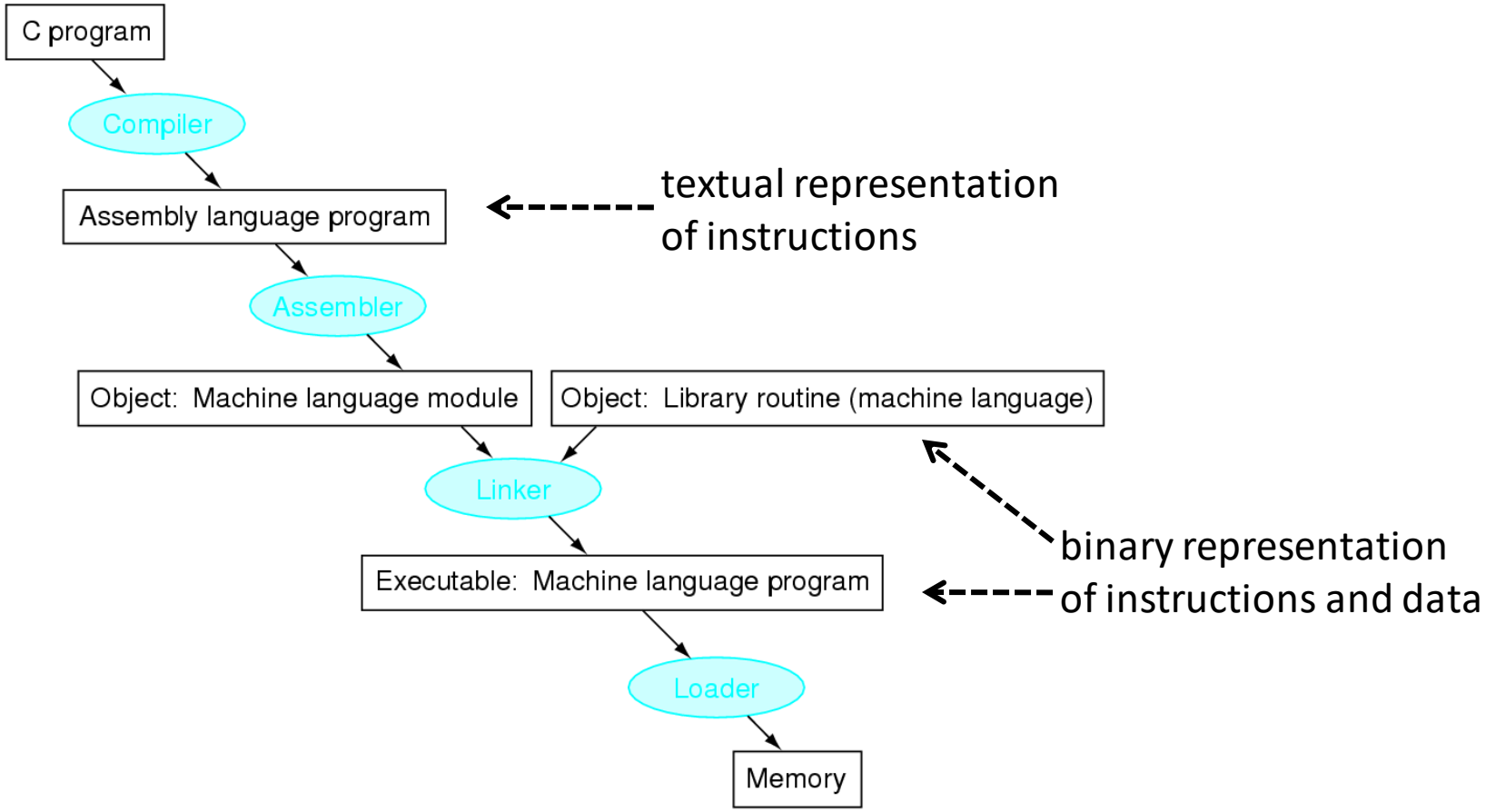


Where we are …

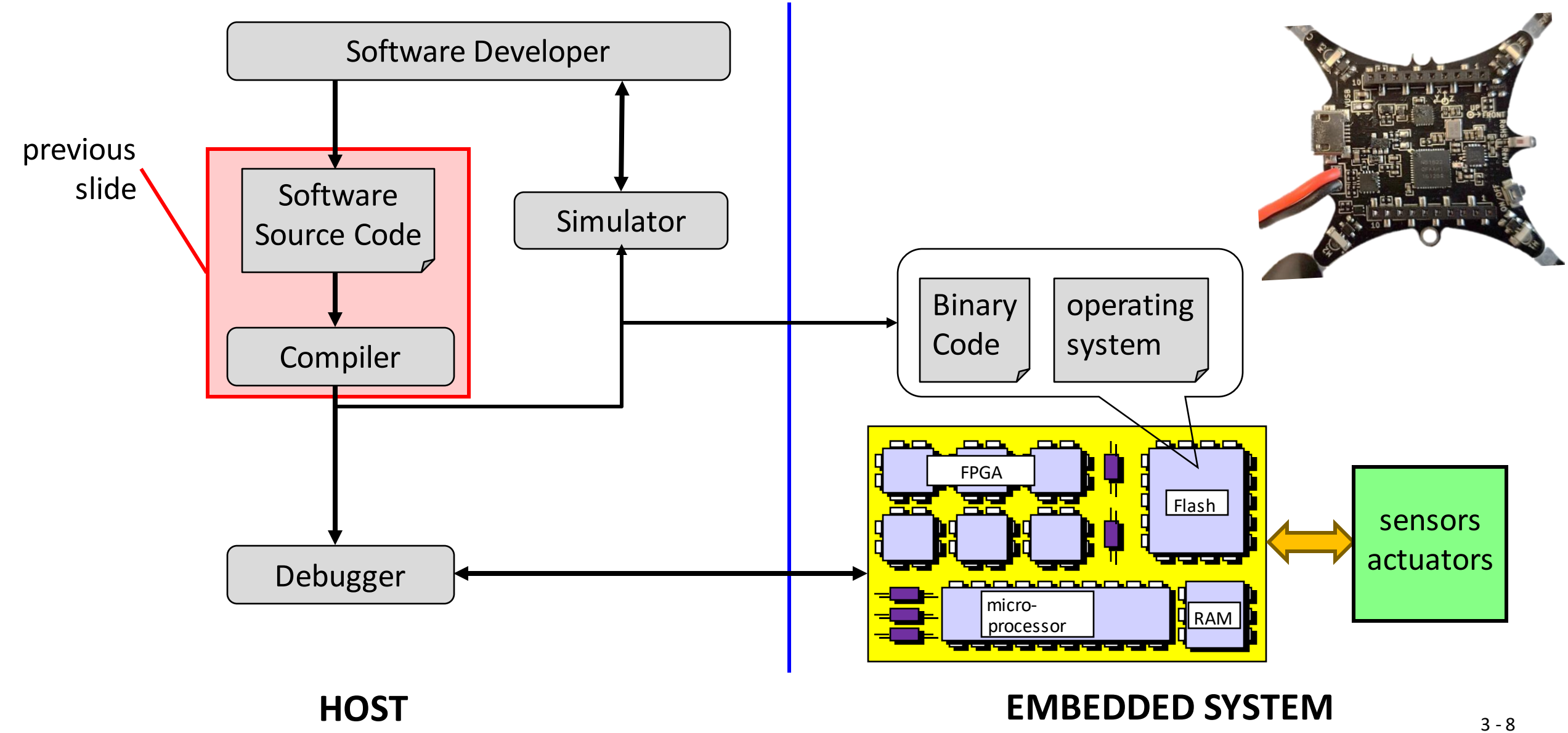

1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-
Software

Remember: Computer Engineering I

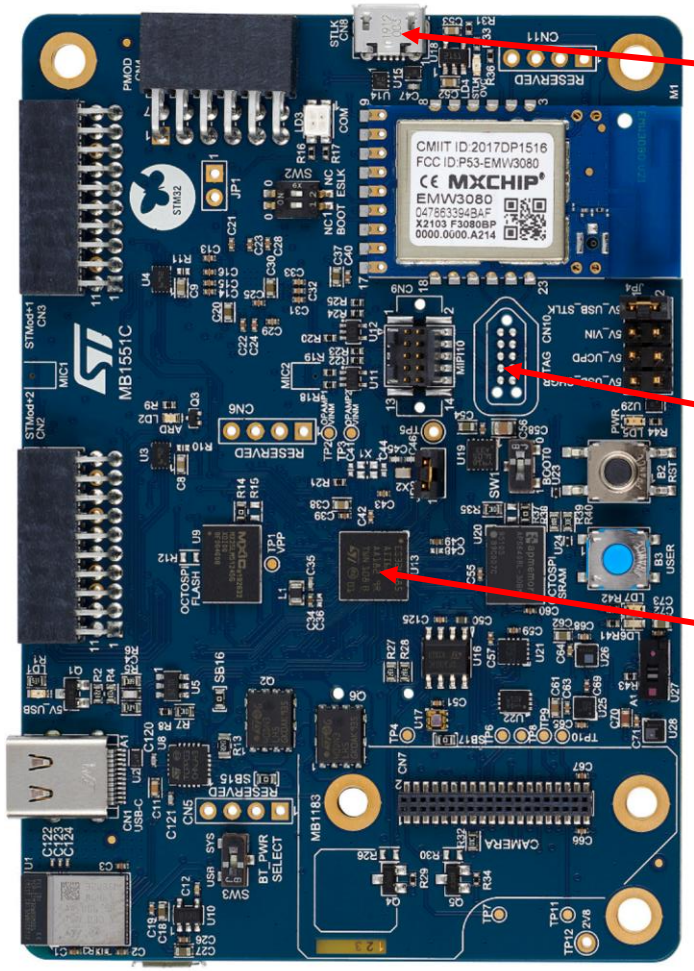
Compilation of a C program to machine language program:



Embedded Software Development



Software Development with MSP432 (ES-Lab)



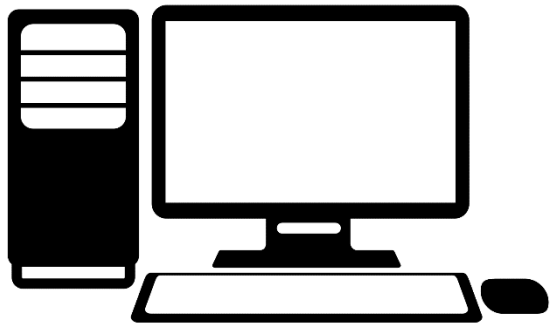
ESD protection

Micro-B USB connector

Status LED RGB

LDO 5 V, 3.3 V, 2.8 V

Debug MCU STLINK-V3E



host PC

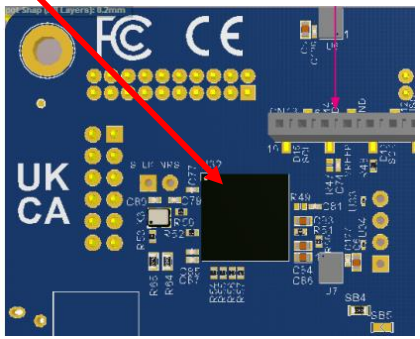
Power, UART, SWD to target

32.768 kHz crystal

Target Device STM32U585

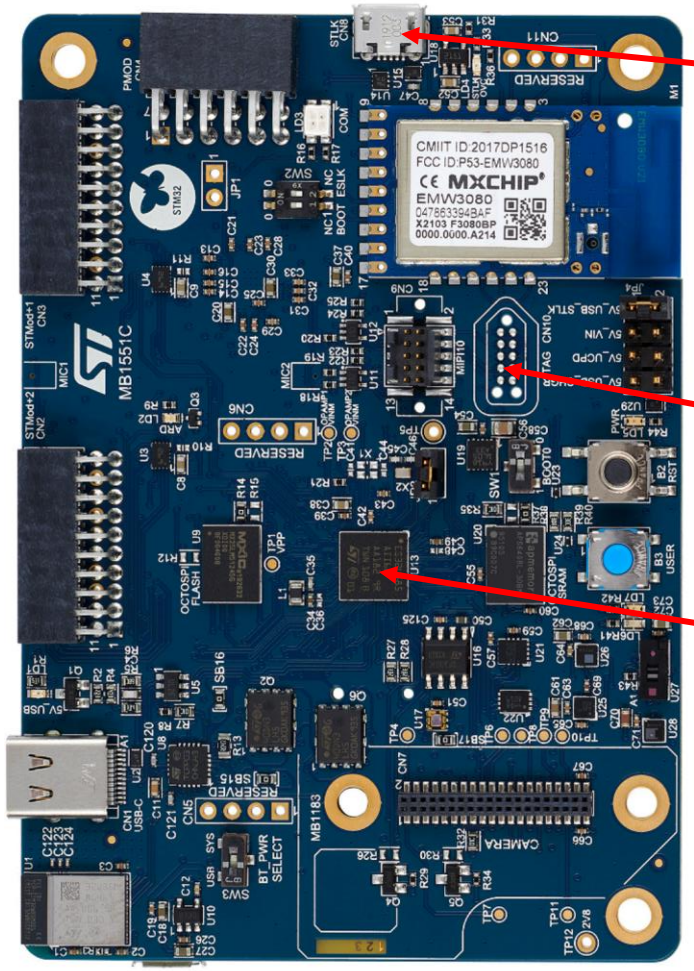
Pmod & Arduino headers

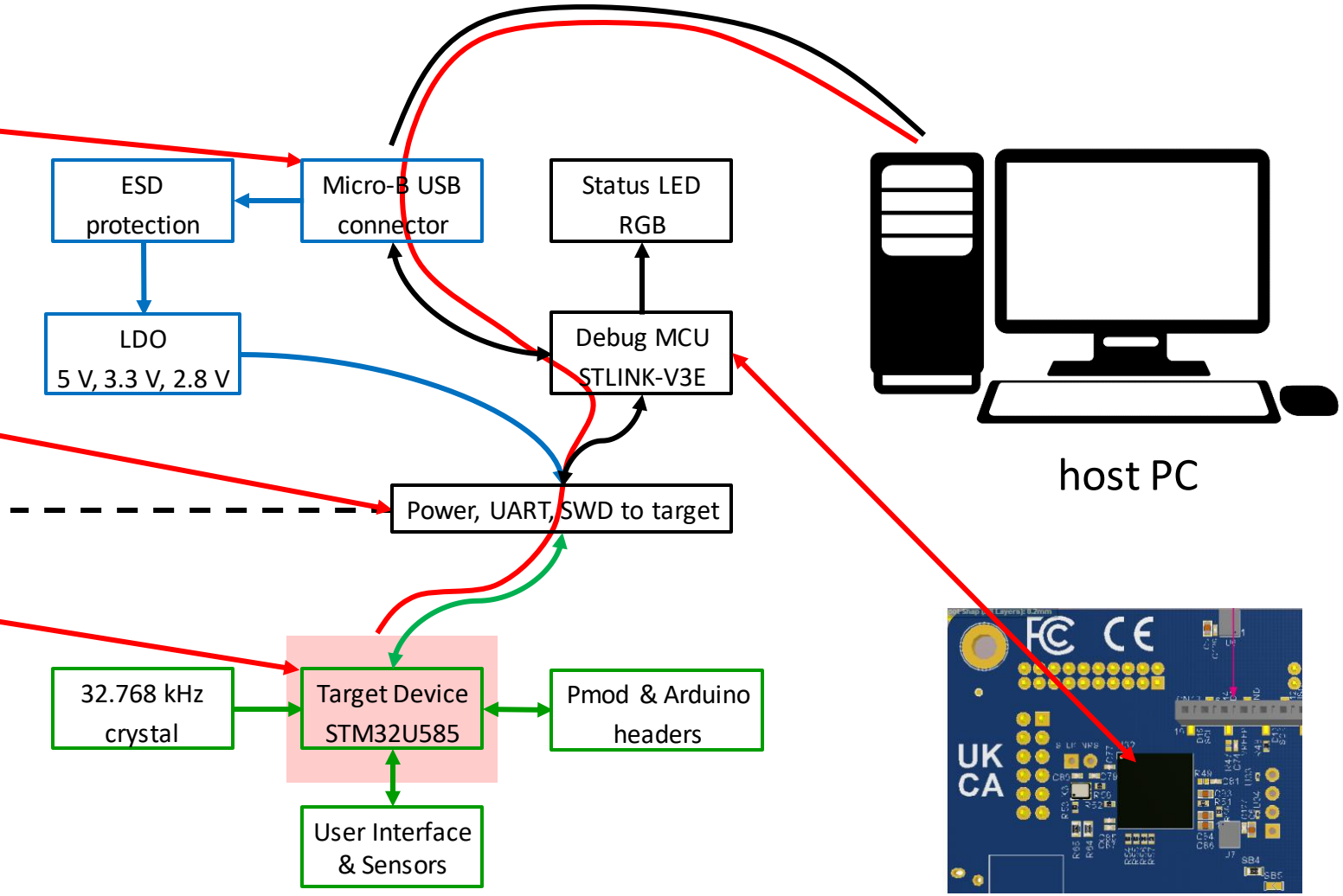
User Interface & Sensors



bottom side

Software Development with MSP432 (ES-Lab)



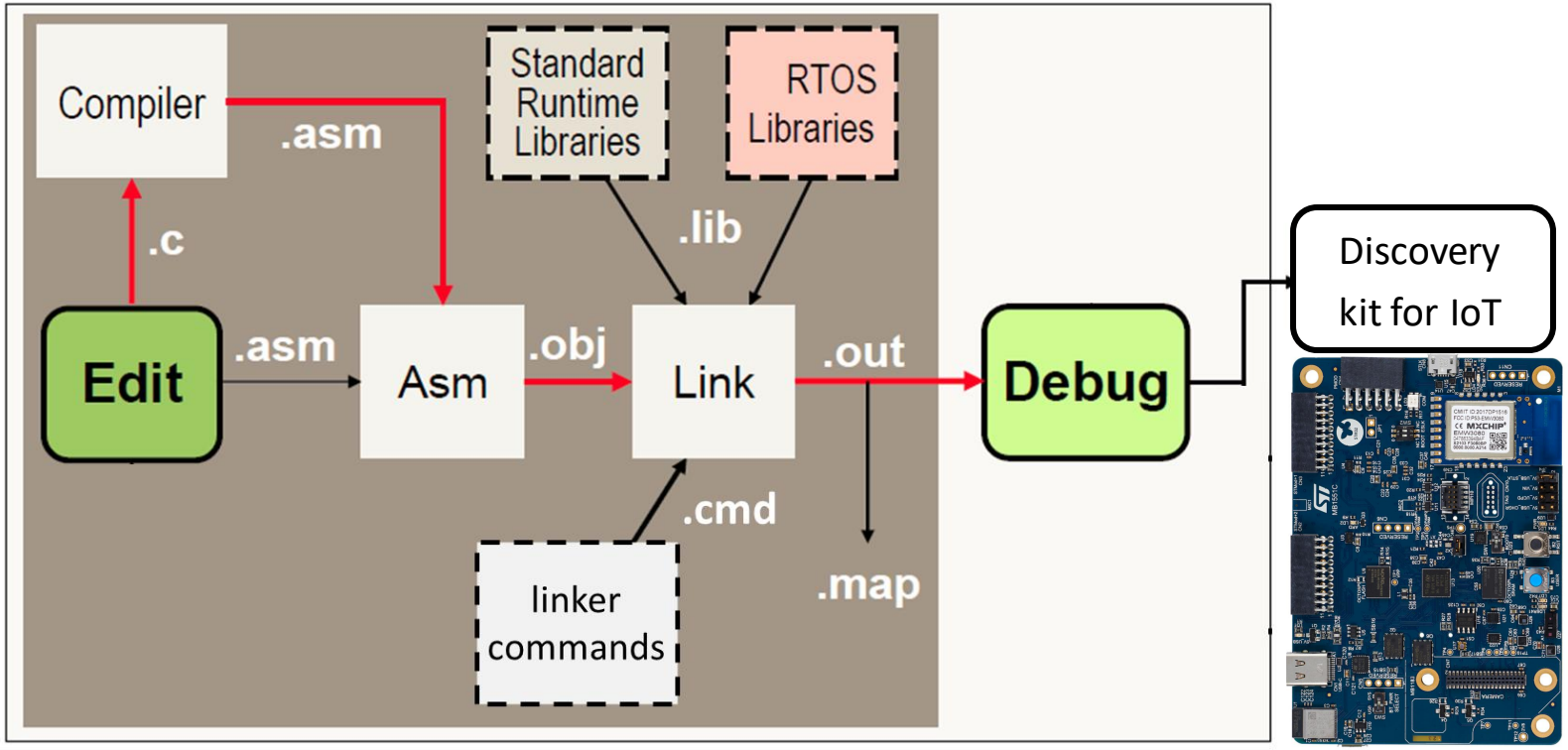


bottom side

Software Development (ES-Lab)

Software development is nowadays usually done with the support of an IDE (Integrated Debugger and Editor / Integrated Development Environment)

- edit and build the code
- debug and validate



Software Development (ES-Lab)

assembly code

object libraries that are referenced in the code

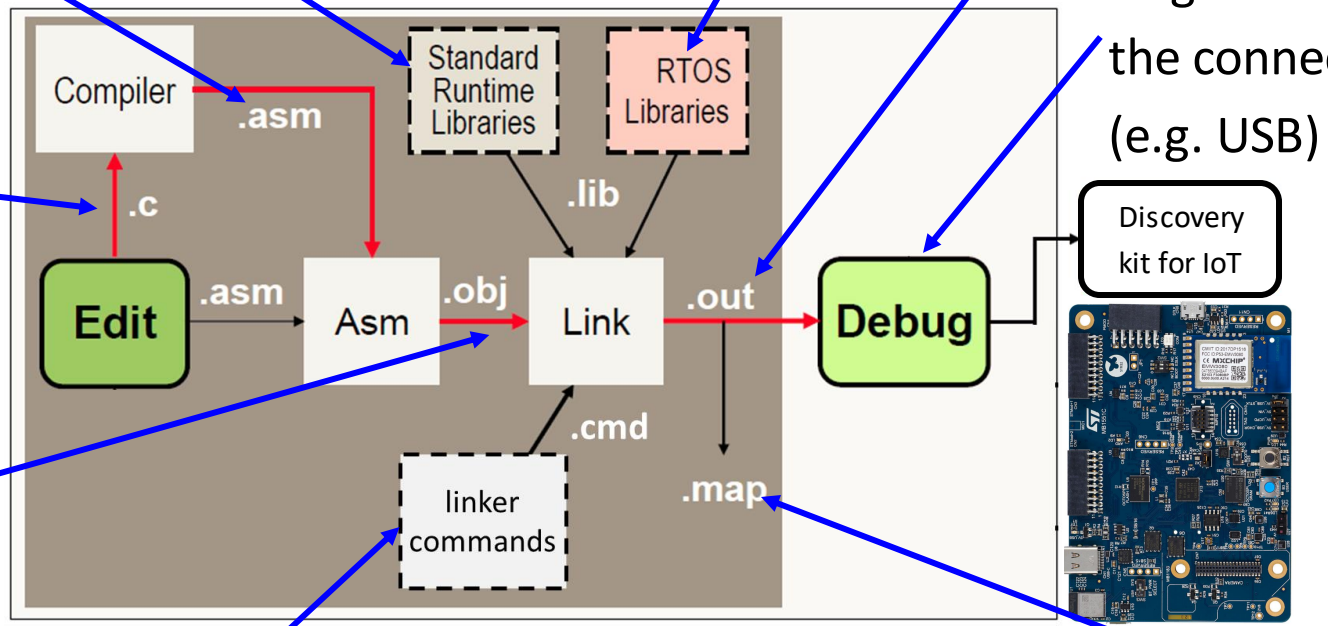
object libraries that contain the operating system (if any)

the executable output file that is loaded into flash memory on the processor

target configuration file specifies the connection to the target (e.g. USB) and the target device

source code file in C

relocatable object file



Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

report created by the linker describing where the program and data sections are located in memory.

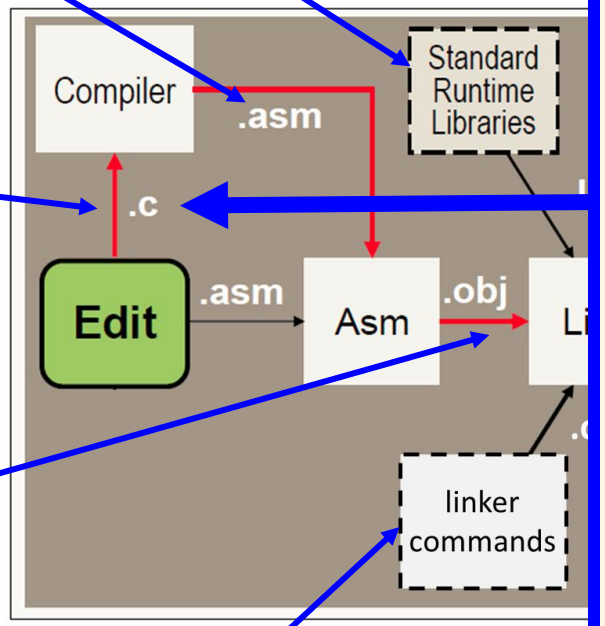
Software Development (ES-Lab)

assembly code

object libraries that are referenced in the code

source code file in C

relocatable object file




```c
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* Configure the System Power */
  SystemPower_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
```

Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

report created by the linker describing where the program and data sections are located in memory.

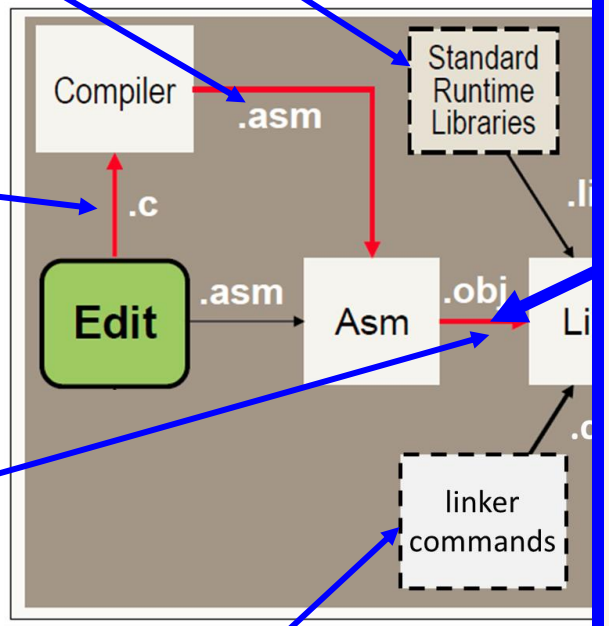
Software Development (ES-Lab)

assembly code

object libraries that are referenced in the code

source code file in C

relocatable object file



Linker command file that tells the lin
how to allocate memory and to stitc
the object files and libraries togethe

```
0x90 <main>                     push    {r7, lr}
0x92 <main+2>                   add     r7, sp, #0
0x94 <main+4>                   bl      0x94 <main+4>
0x98 <main+8>                   bl      0x98 <main+8>
0x9c <main+12>                  bl      0x9c <main+12>
0xa0 <main+16>                  bl      0xa0 <main+16>
0xa4 <main+20>                  bl      0xa4 <main+20>
0xa8 <main+24>                  bl      0xa8 <main+24>
0xac <main+28>                  b.n     0xac <main+28>
0xae <SystemClock_Config>       push    {r7, lr}
0xb0 <SystemClock_Config+2>     sub     sp, #120        ; 0x78
0xb2 <SystemClock_Config+4>     add     r7, sp, #0
0xb4 <SystemClock_Config+6>     add.w   r3, r7, #24
0xb8 <SystemClock_Config+10>    movs    r2, #96 ; 0x60
0xba <SystemClock_Config+12>    movs    r1, #0
0xbc <SystemClock_Config+14>    mov     r0, r3
0xbe <SystemClock_Config+16>    bl      0xbe <SystemClock_Config+16>
0xc2 <SystemClock_Config+20>    mov     r3, r7
0xc4 <SystemClock_Config+22>    movs    r2, #0
0xc6 <SystemClock_Config+24>    str     r2, [r3, #0]
0xc8 <SystemClock_Config+26>    str     r2, [r3, #4]
0xca <SystemClock_Config+28>    str     r2, [r3, #8]
0xcc <SystemClock_Config+30>    str     r2, [r3, #12]
0xce <SystemClock_Config+32>    str     r2, [r3, #16]
0xd0 <SystemClock_Config+34>    str     r2, [r3, #20]
0xd2 <SystemClock_Config+36>    movs    r0, #0
0xd4 <SystemClock_Config+38>    bl      0xd4 <SystemClock_Config+38>
0xd8 <SystemClock_Config+42>    mov     r3, r0
0xda <SystemClock_Config+44>    cmp     r3, #0
0xdc <SystemClock_Config+46>    beq.n   0xe2 <SystemClock_Config+52>
0xde <SystemClock_Config+48>    bl      0xde <SystemClock_Config+48>
```

Software Development (ES-Lab)

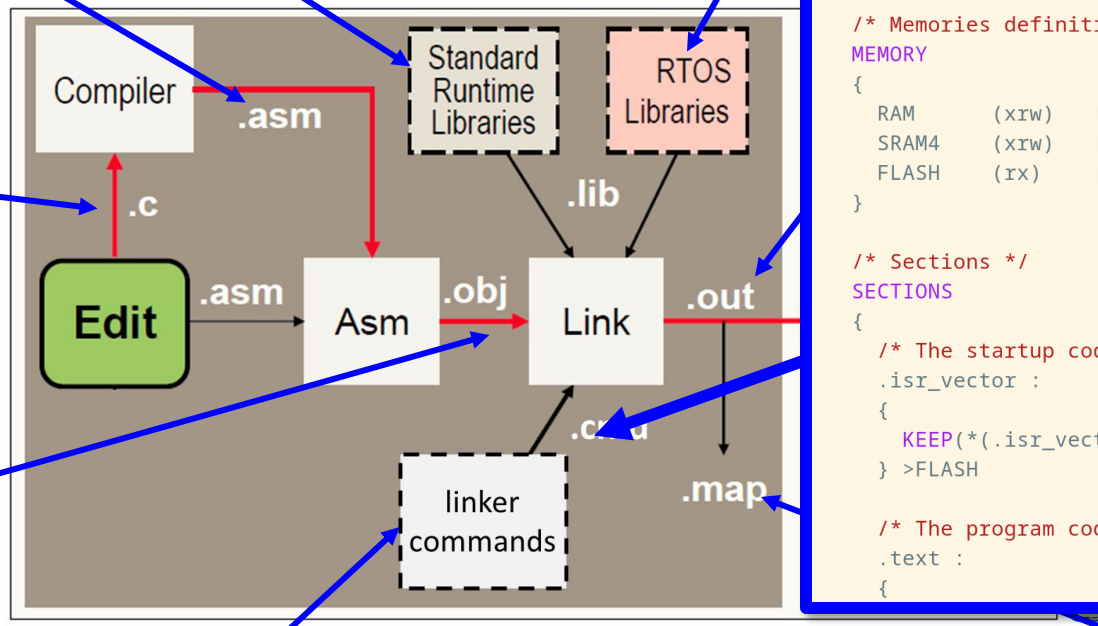
assembly code

source code file in C

relocatable object file

object libraries that are referenced in the code

object librarie
contain the o
system (if any



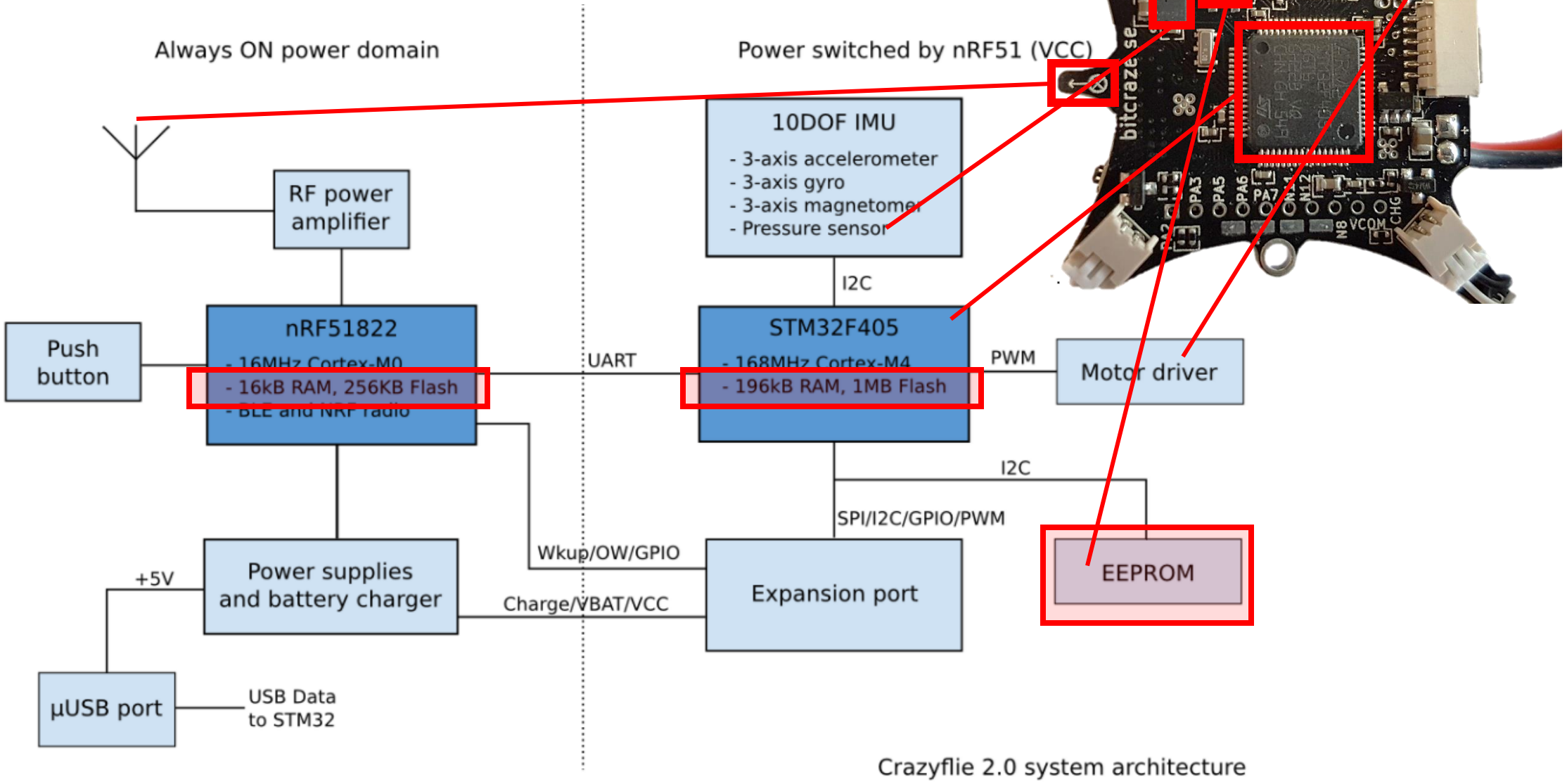

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM);       /* end of "RAM" Ram type memory */

_Min_Heap_Size = 0x200 ;   /* required amount of heap  */
_Min_Stack_Size = 0x400 ;  /* required amount of stack */

/* Memories definition */
MEMORY
{
  RAM      (xrw)    : ORIGIN = 0x20000000,  LENGTH = 768K
  SRAM4    (xrw)    : ORIGIN = 0x28000000,  LENGTH = 16K
  FLASH    (rx)     : ORIGIN = 0x08000000,  LENGTH = 2048K
}

/* Sections */
SECTIONS
{
  /* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    KEEP(*(.isr_vector)) /* Startup code */
  } >FLASH

  /* The program code and other data into "FLASH" Rom type memory */
  .text :
  {
```

Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

report created by the linker describing where the program and data sections are located in memory.

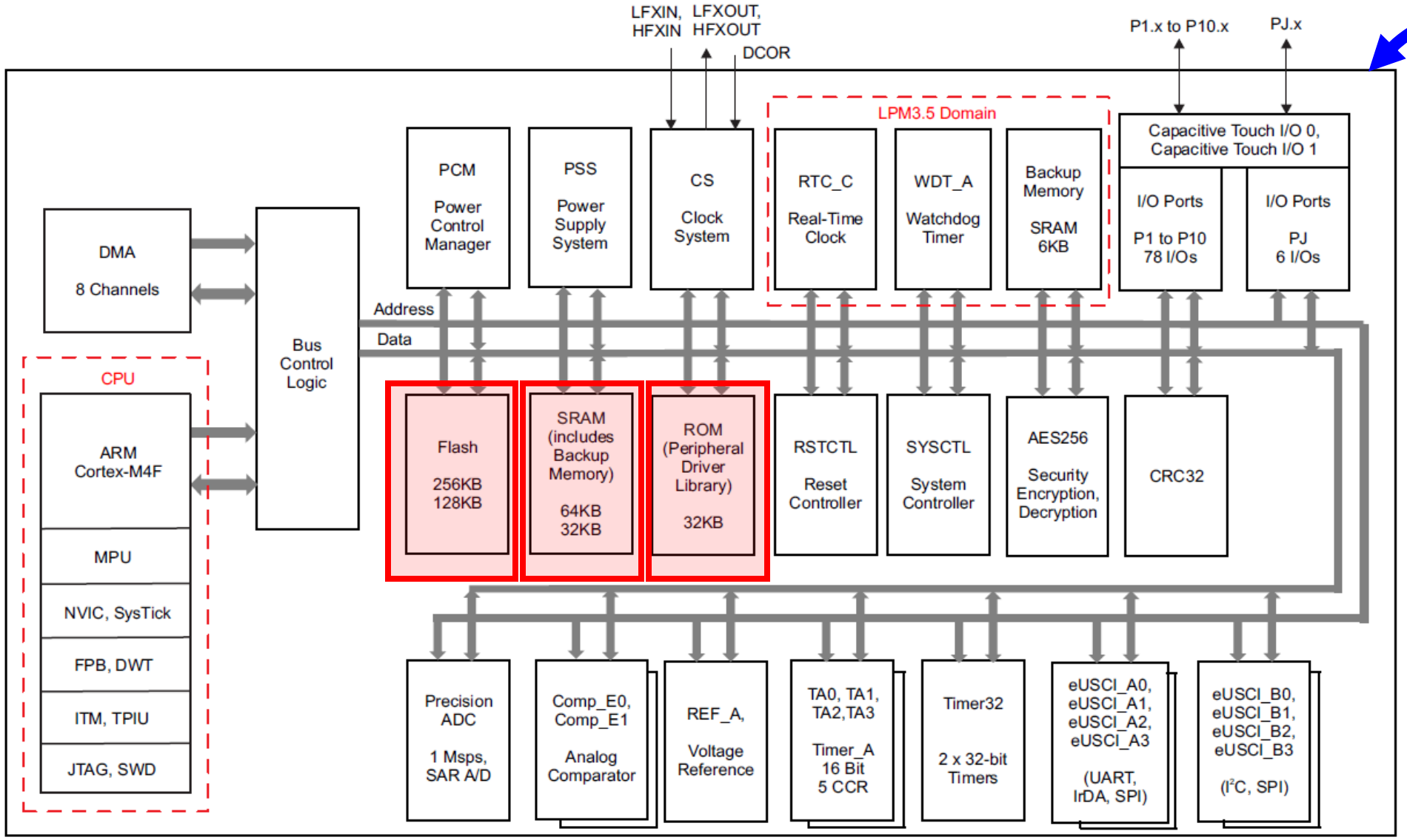
Storage

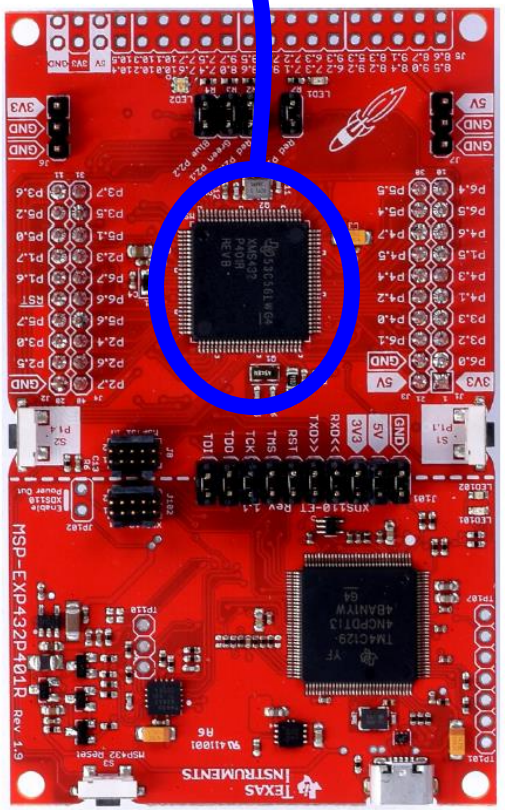
Remember … ?



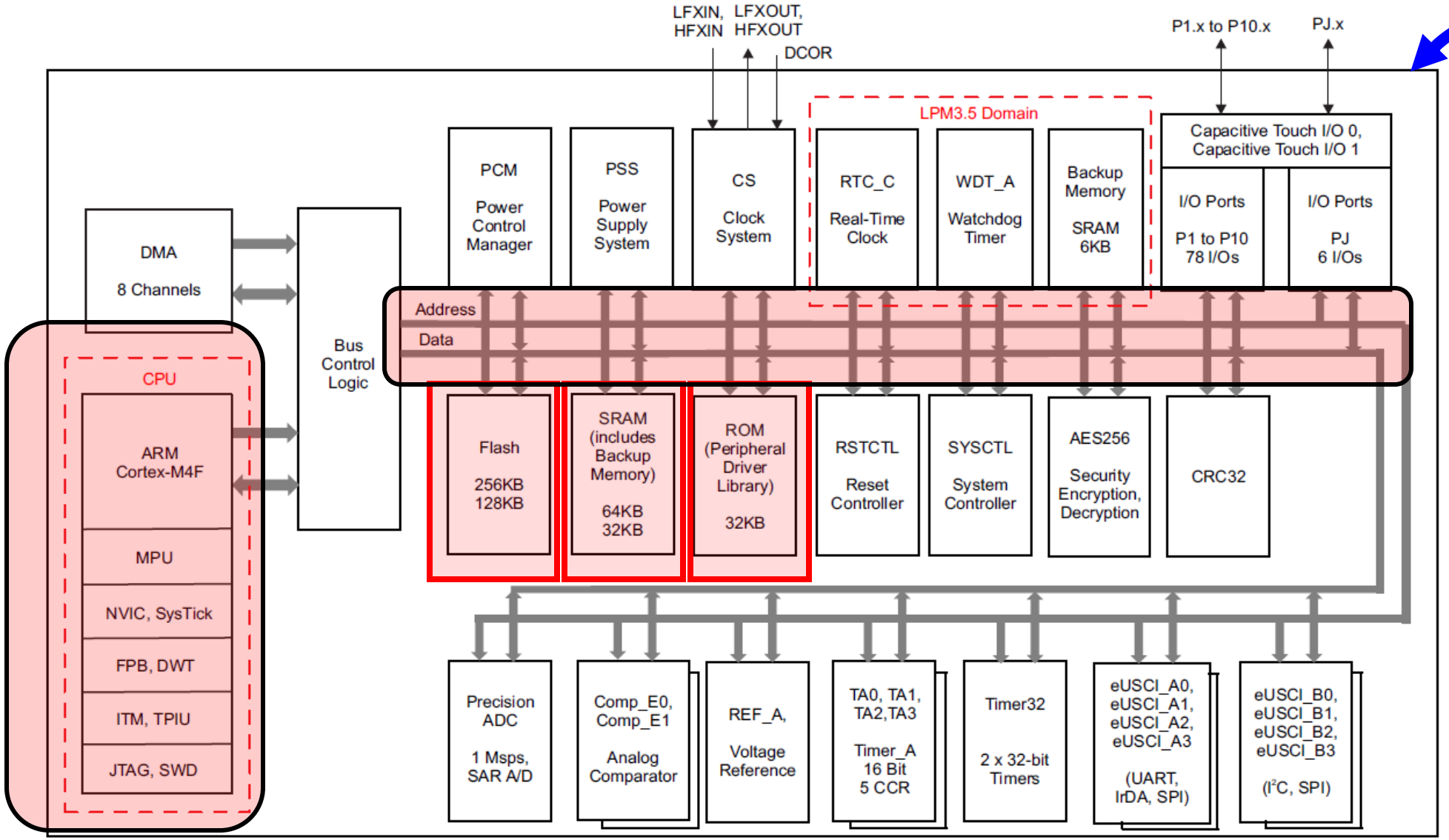
Crazyflie 2.0 system architecture

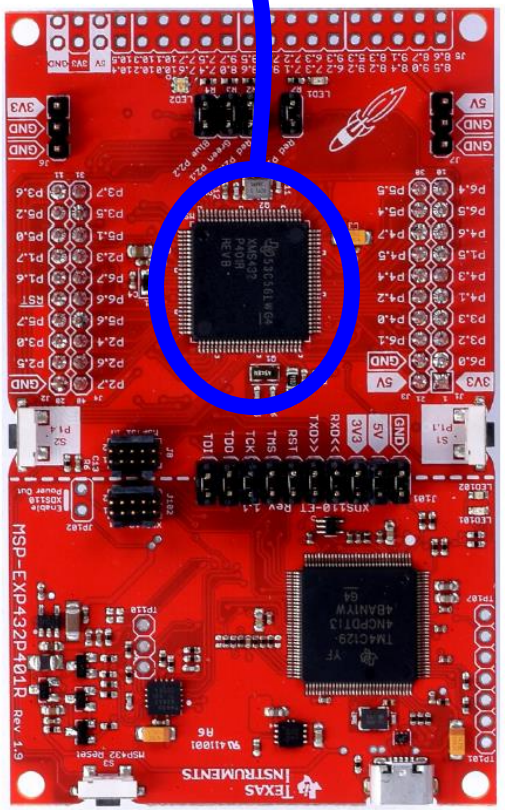
MSP432P401R (Last year lab - ES-Lab)





von Neumann or Harvard ? Why?



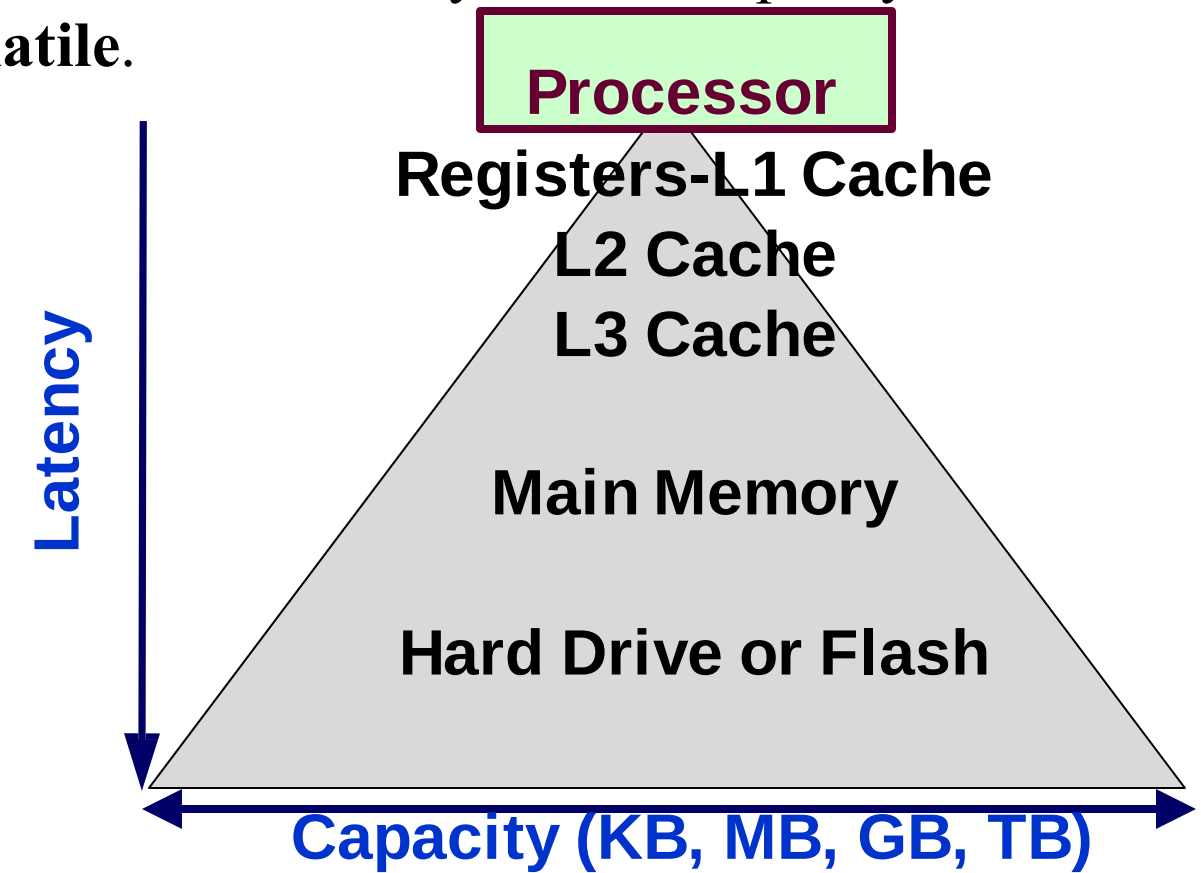


Storage

Registers/SRAM / DRAM / Flash

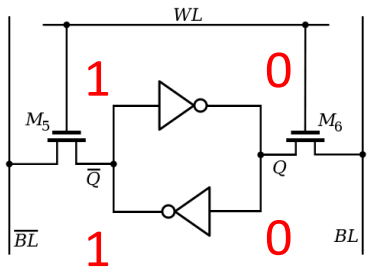
Memory Hierarchies and persistent and volatile memory

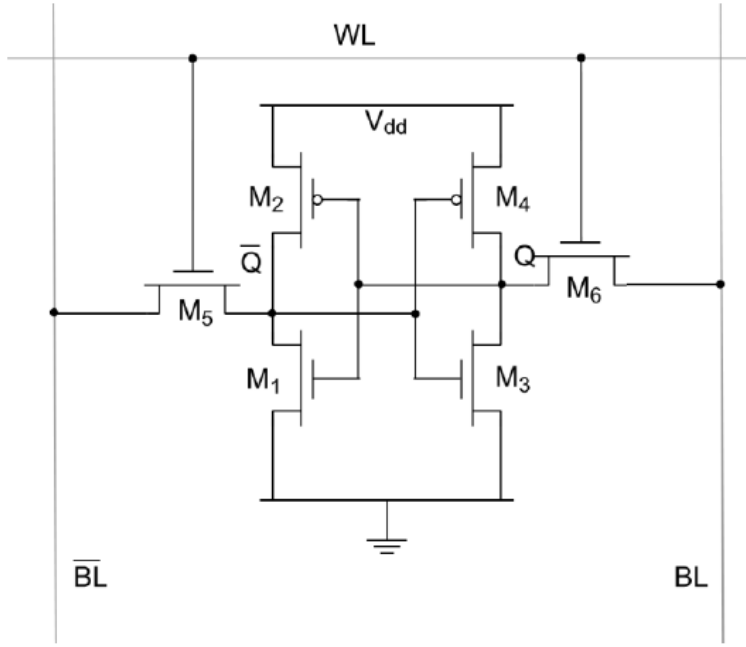
- **Processor registers** can be seen as the fastest level in the memory hierarchy, with only a limited capacity of at most a few hundred words.
- The **working memory** (or **main memory**) of computer systems implements the storage implied by processor memory addresses. Usually it has a capacity between a few megabytes and some gigabytes and is **volatile**.
- Hard drive or Flash
  - Persistent



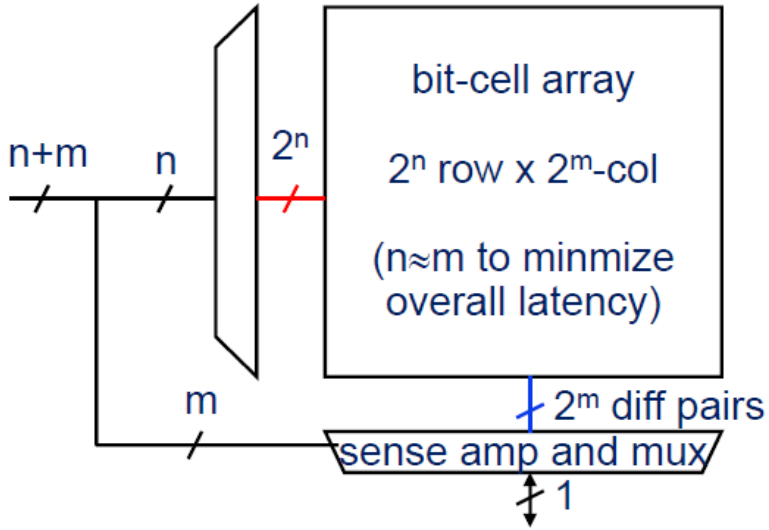
Static Random Access Memory (SRAM)

- *Single bit is stored in a bi-stable circuit*
- *Static Random Access Memory* is used for
  - caches
  - register file within the processor core
  - small but fast memories





- *Read:*
  1. Pre-charge all bit-lines to average voltage
  2. decode address (n+m bits)
  3. select row of cells using 2^n single-bit word lines (WL)
  4. selected bit-cells drive all bit-lines BL (2^m pairs)
  5. sense difference between bit-line pairs and read out
- *Write:*
  - select row and overwrite bit-lines using strong signals



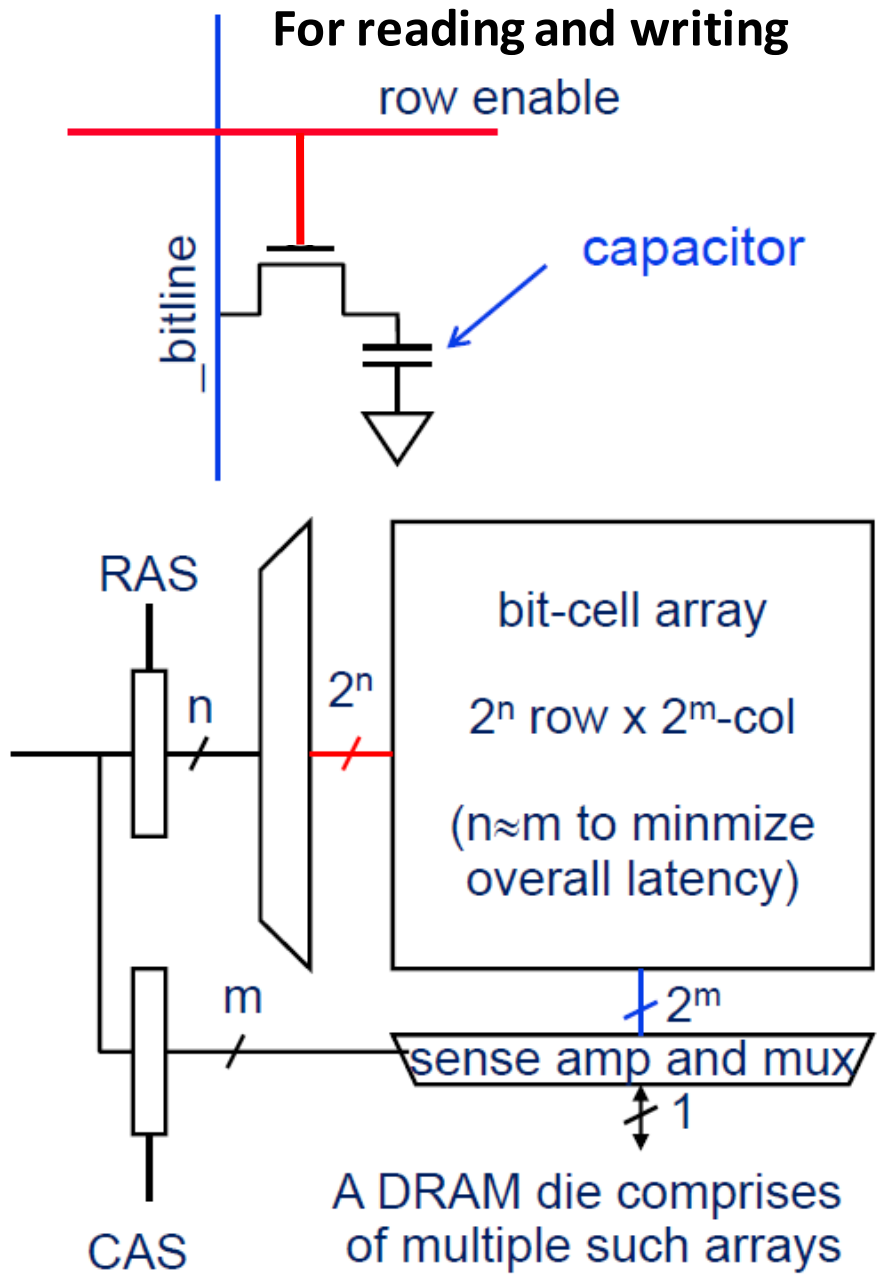
Dynamic Random Access (DRAM)

Single bit is stored as a charge in a capacitor

- Bit cell loses charge when read, bit cell drains over time
- Slower access than with SRAM due to small storage capacity in comparison to capacity of bit-line.
- Higher density than SRAM (1 vs. 6 transistors per bit)

DRAMs require *periodic refresh* of charge

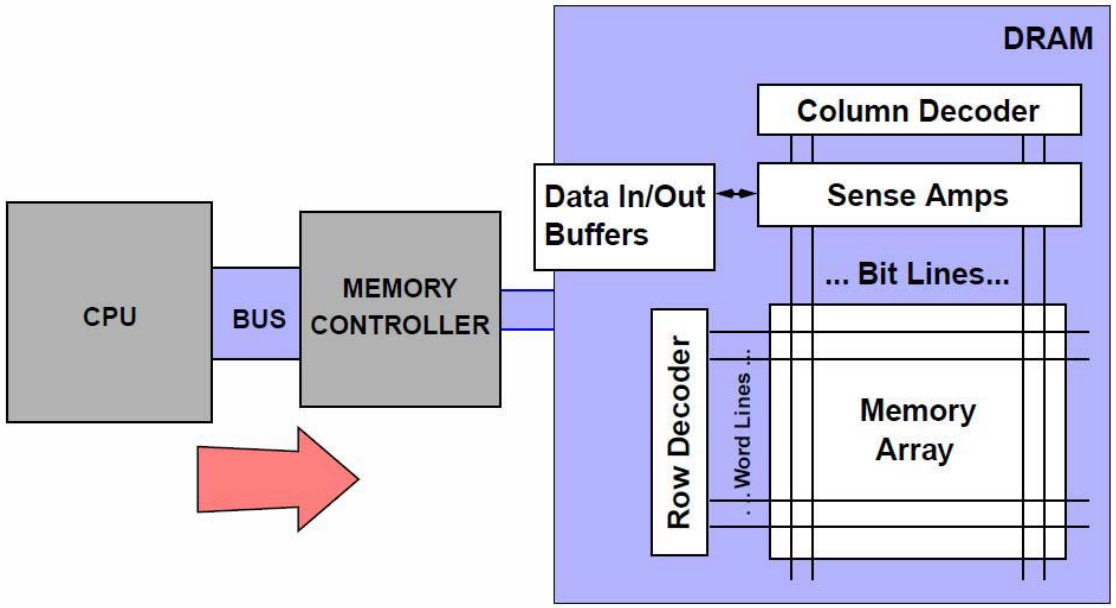
- Performed by the memory controller
- Refresh interval is tens of ms
- DRAM is unavailable during refresh



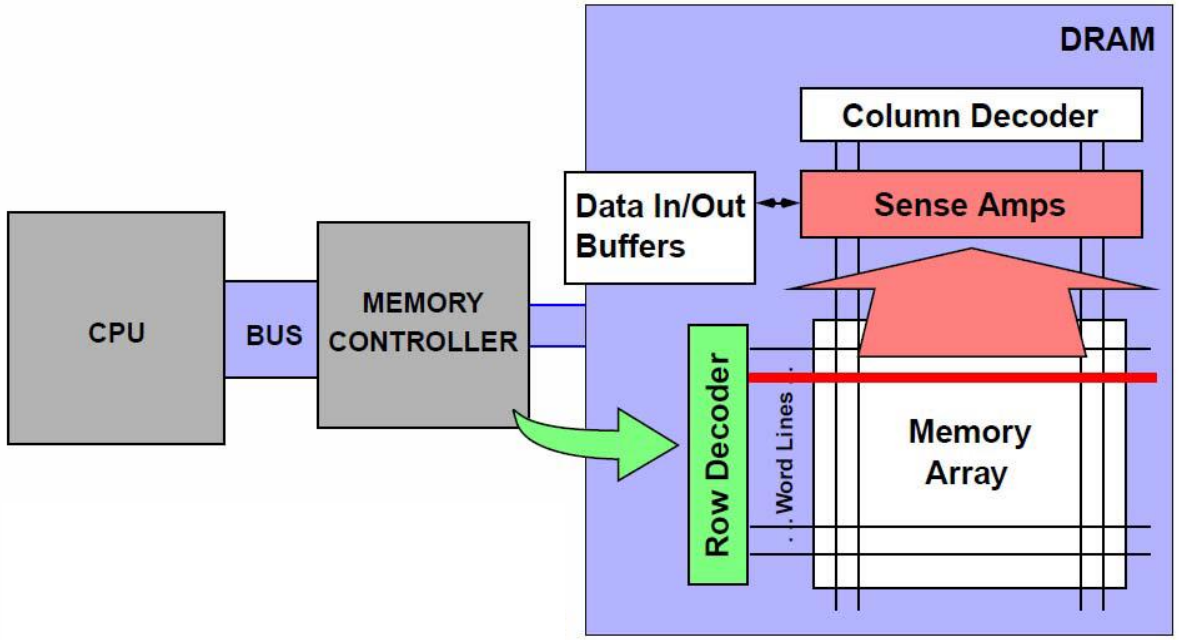
(RAS/CAS = row/column address select)

DRAM – Typical Access Process

1. Bus Transmission

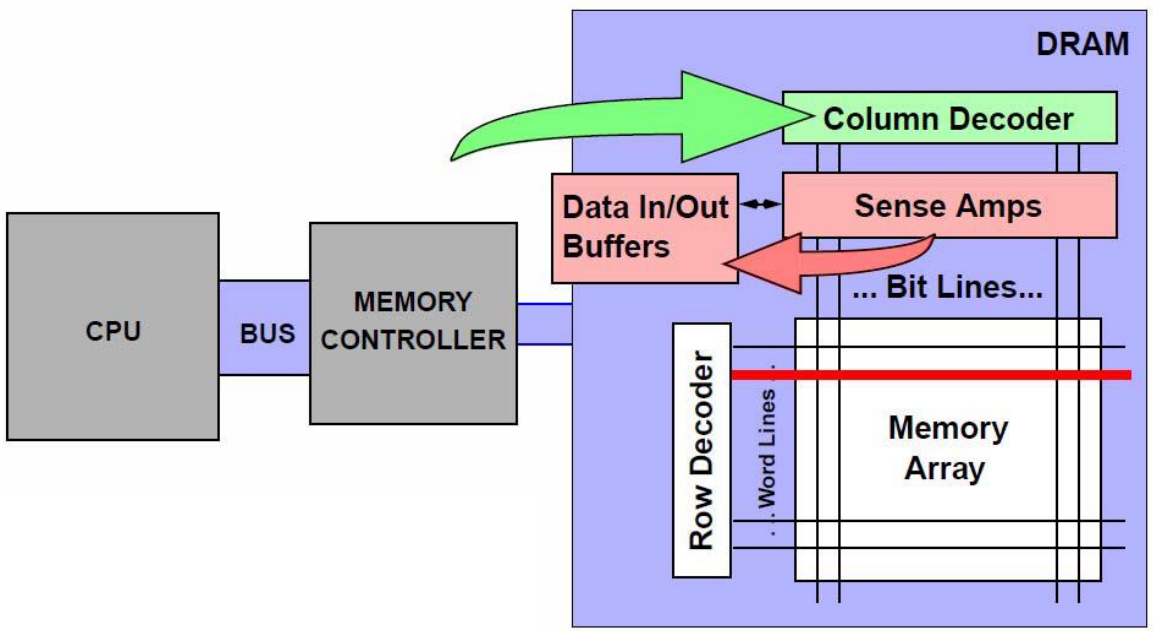


2. Precharge and Row Access

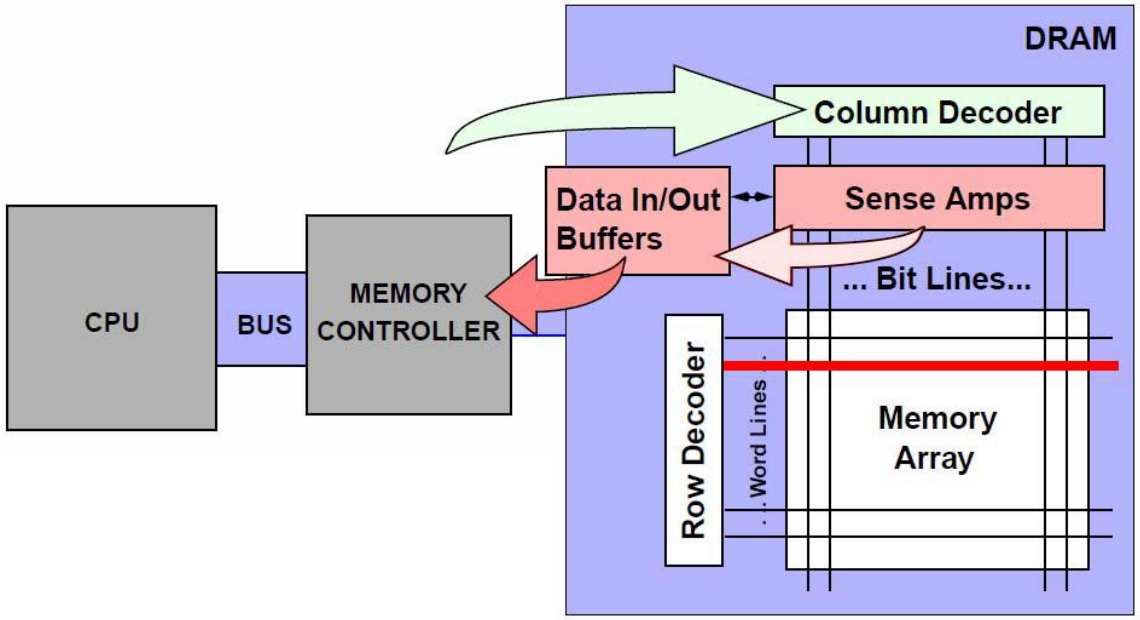


DRAM – Typical Access Process

3. Column Access



4. Data Transfer and Bus Transmission

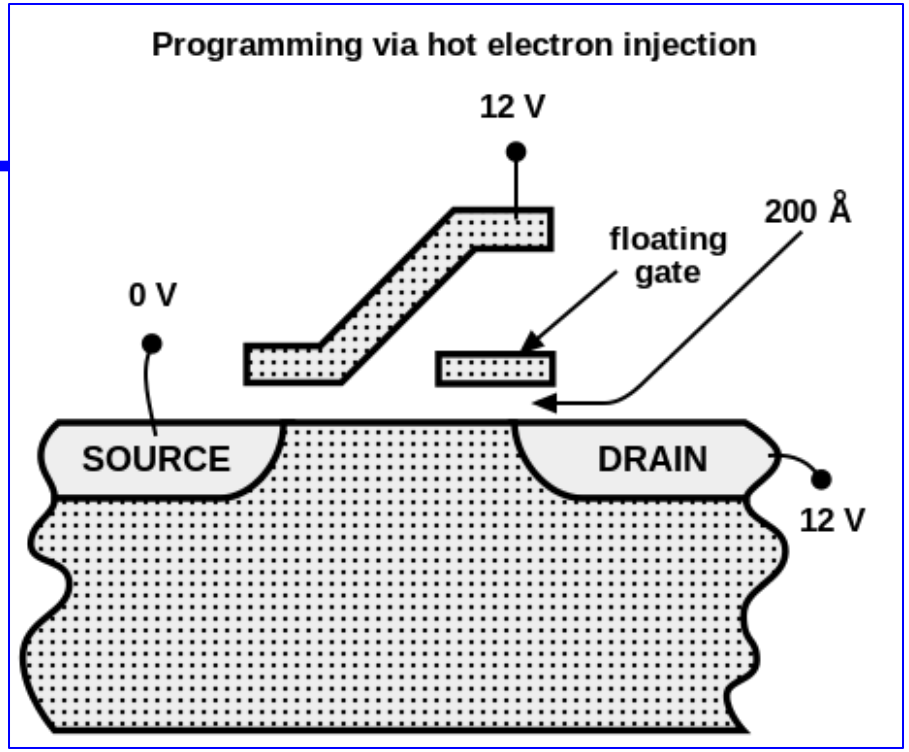


Flash Memory

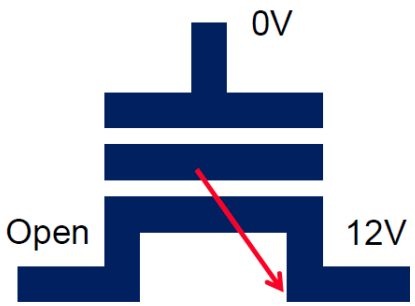
Electrically modifiable, non-volatile storage

Principle of operation:

- Transistor with a second “floating” gate
- Floating gate can trap electrons
- This results in a detectable change in threshold voltage

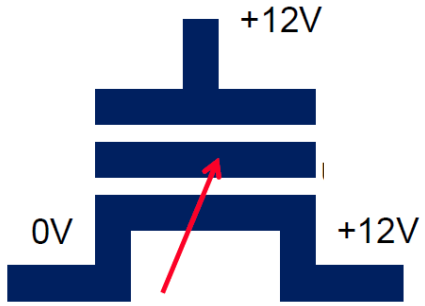


Erasing to logical “1”



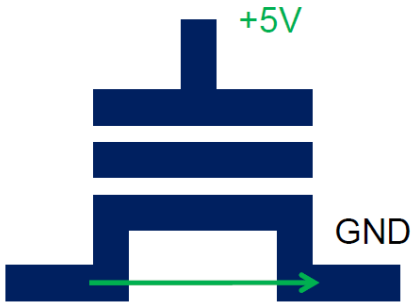
“Quantum tunneling”
Drains charge from FG

Programming (=writing) to logical “0”



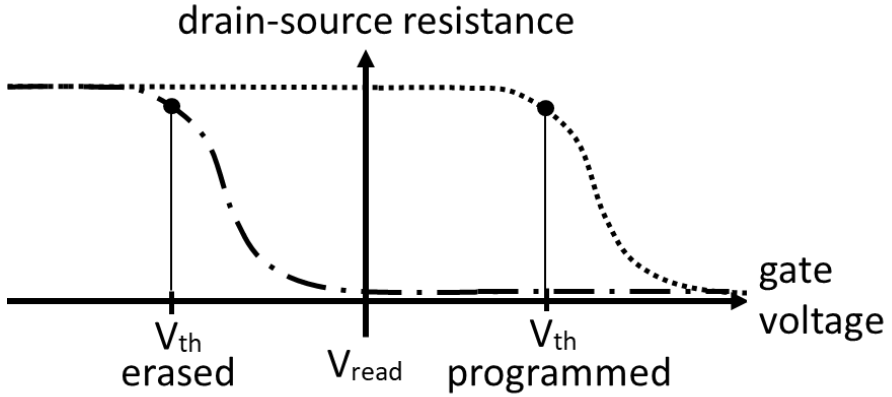
“Hot-electron injection”
traps charge in FG

Reading

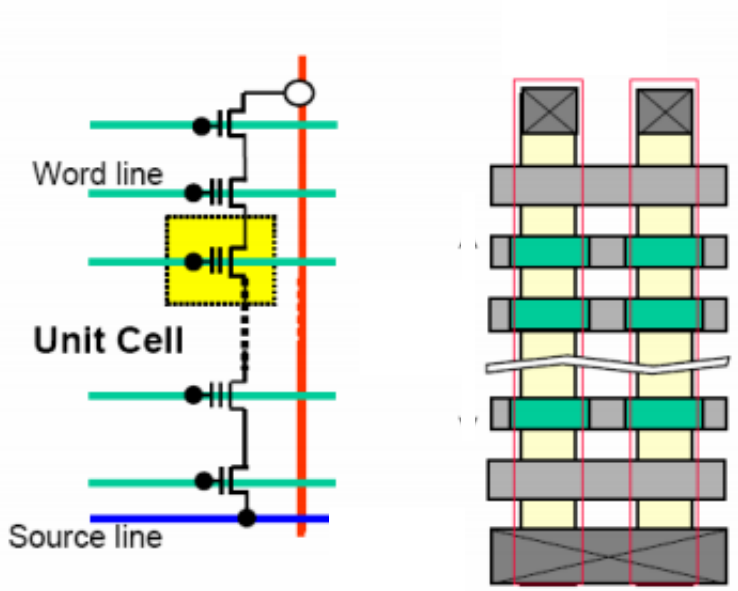
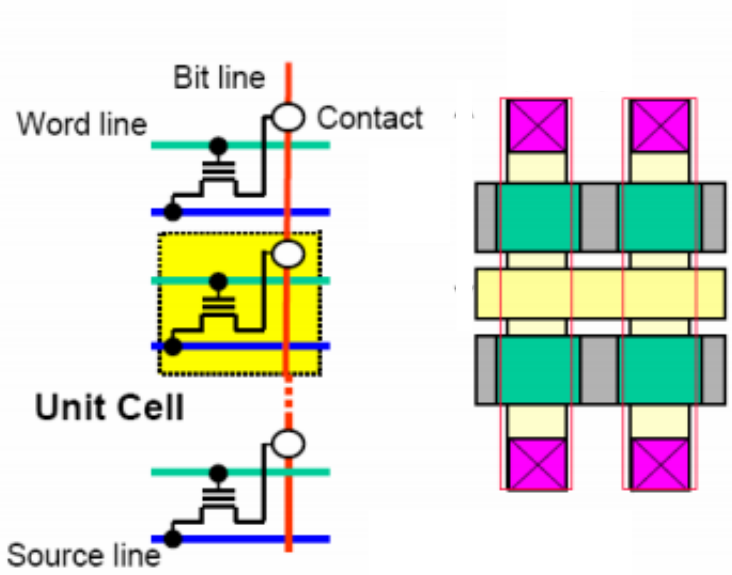


Turn on low Vt or High Vt?

Detect I_{on} to read 0 or 1



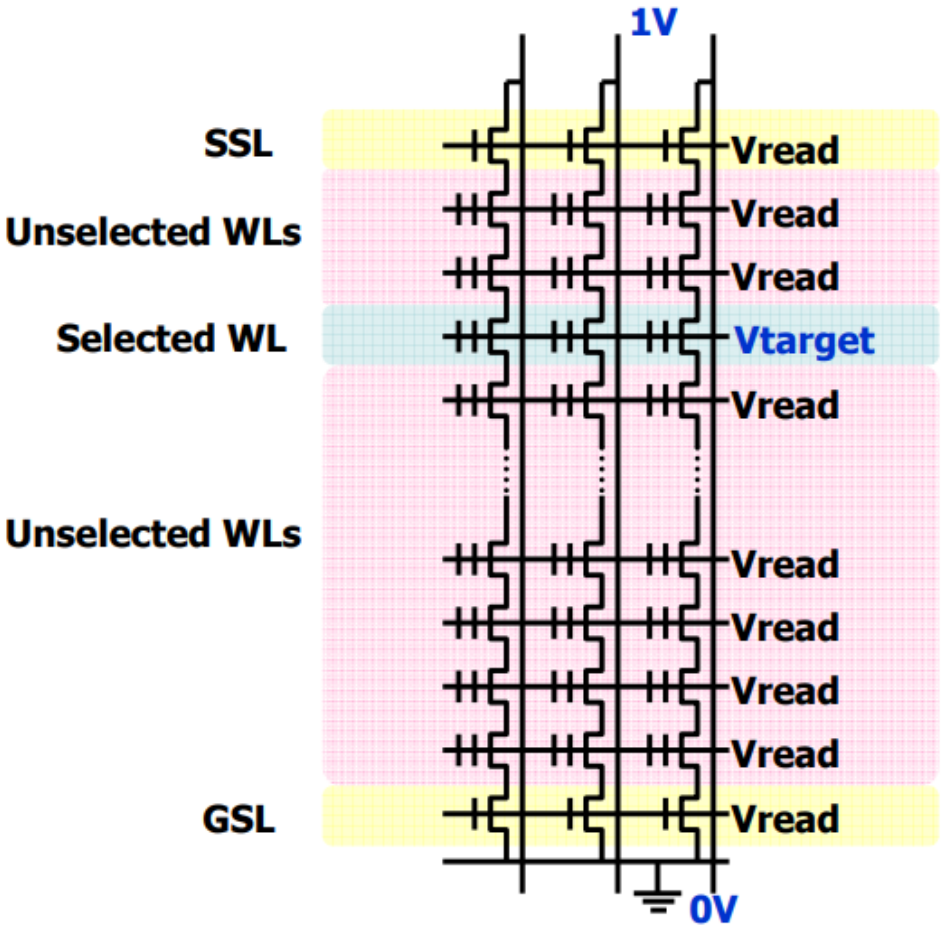
NAND and NOR Flash Memory

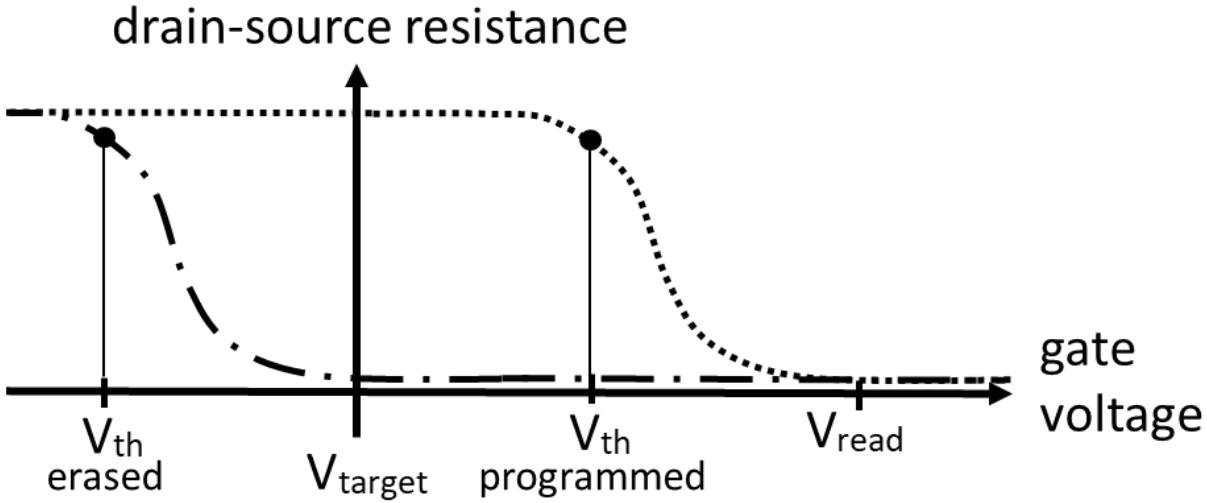
	NAND	NOR
Cell Array & Size		
Cross-section		
Features	Small Cell Size, High Density Low Power ➔ *Mass Storage*	**Fast random access** ➔ *Code Storage*

Example: Reading out NAND Flash

Selected word-line (WL) : Target voltage (V_{target})

Unselected word-lines : V_{read} is high enough to have a low resistance in all transistors in this row





Storage

Memory Map

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Available memory:

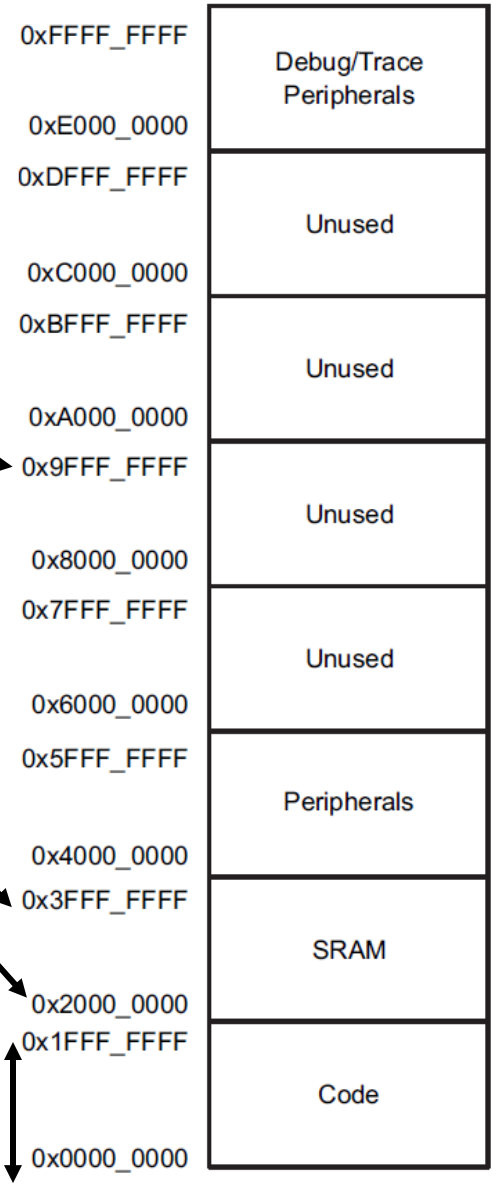
- The processor used in the lab (MSP432P401R) has built in 256kB flash memory, 64kB SRAM and 32kB ROM (Read Only Memory).

Address space:

- The processor uses 32 bit addresses as all the **ARM Cortex-M Microcontrollers**. Therefore, the addressable memory space is 4 GByte (= 2^{32} Byte) as each memory location corresponds to 1 Byte (for other ARM cortex-M4 usually is used 32 bits) .
- The address space is used to address the memories (reading and writing), to address the peripheral units, and to have access to debug and trace information (memory mapped microarchitecture).
- The address space is partitioned into zones, each one with a dedicated use. The following is a simplified description to introduce the basic concepts.

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

Hexa	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111

Hexa	8	9	A	B	C	D	E	F
Binary	1000	1001	1010	1011	1100	1101	1110	1111

0011 1111 …. 1111

0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → 2^{29} different addresses

capacity = 2^{29} Byte = 512 MByte

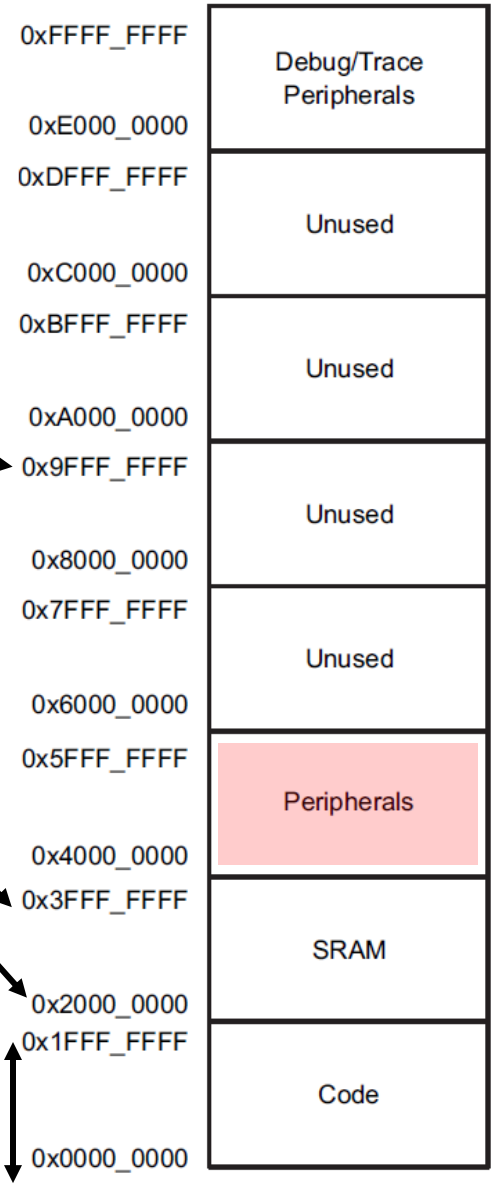
3FFF FFFF -

0x200 000 +

1

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

0011 1111 …. 1111

0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → 2^{29} different addresses

capacity = 2^{29} Byte = 512 MByte

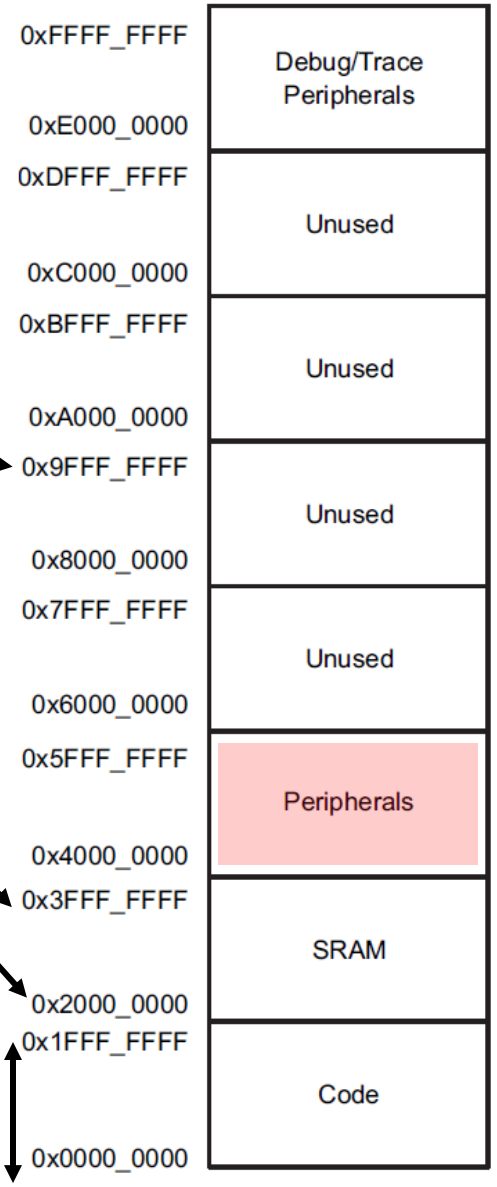
ADDRESS RANGE	PERIPHERAL
0x4000_0000 to 0x4000_03FF	Timer_A0
0x4000_0400 to 0x4000_07FF	Timer_A1
0x4000_0800 to 0x4000_0BFF	Timer_A2
0x4000_0C00 to 0x4000_0FFF	Timer_A3
0x4000_1000 to 0x4000_13FF	eUSCI_A0
0x4000_1400 to 0x4000_17FF	eUSCI_A1
0x4000_1800 to 0x4000_1BFF	eUSCI_A2
0x4000_1C00 to 0x4000_1FFF	eUSCI_A3
…	
0x4000_4400 to 0x4000_47FF	RTC_C
0x4000_4800 to 0x4000_4BFF	WDT_A
0x4000_4C00 to 0x4000_4FFF	Port Module
…	

Table 6-21. Port Registers (Base Address: 0x4000_4C00)

REGISTER NAME	ACRONYM	OFFSET from base address
Port 1 Input	P1IN	000h
Port 2 Input	P2IN	001h
Port 1 Output	P1OUT	002h
Port 2 Output	P2OUT	003h

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

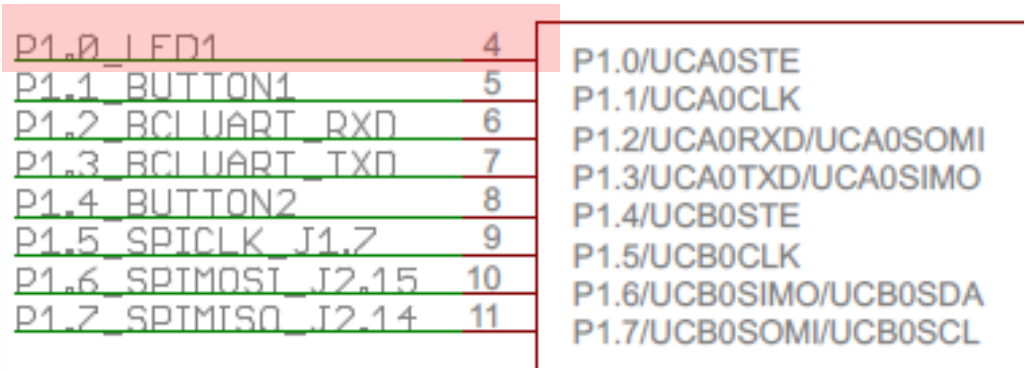
0011 1111 …. 1111
0010 0000 …. 0000

diff. = 0001 1111 …. 1111 →
2^{29} different addresses
capacity = 2^{29} Byte = 512 MByte

Table 6-21. Port Registers (Base Address: 0x4000_4C00)

REGISTER NAME	ACRONYM	OFFSET
Port 1 Input	P1IN	000h
Port 2 Input	P2IN	001h
Port 1 Output	P1OUT	002h
Port 2 Output	P2OUT	003h

Schematic of LaunchPad as used in the Lab:

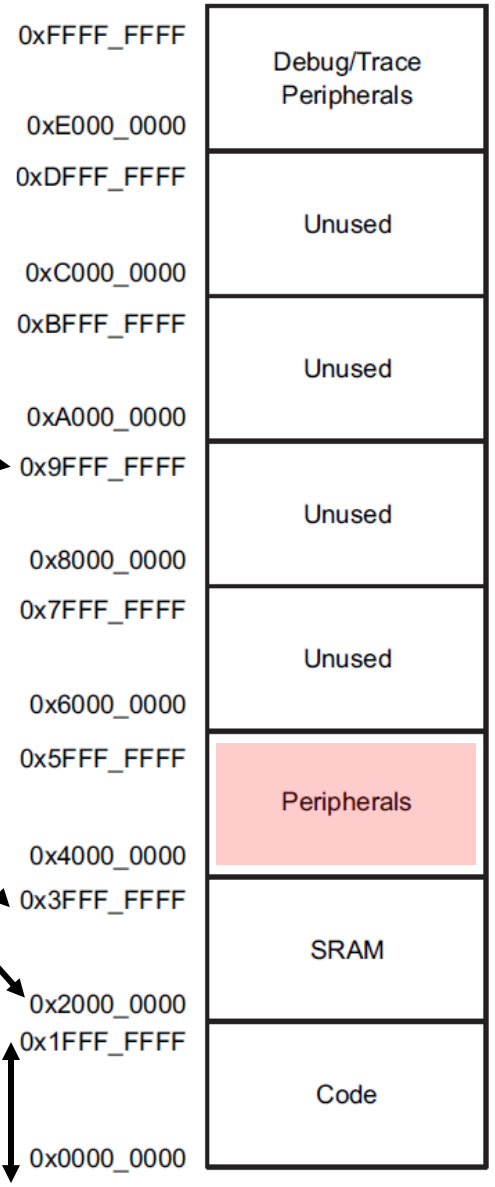


LED1 is connected to Port 1, Pin 0

How do we toggle LED1 in a C program?

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

0011 1111 …. 1111
0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → 2^{29} different addresses

capacity = 2^{29} Byte = 512 MByte

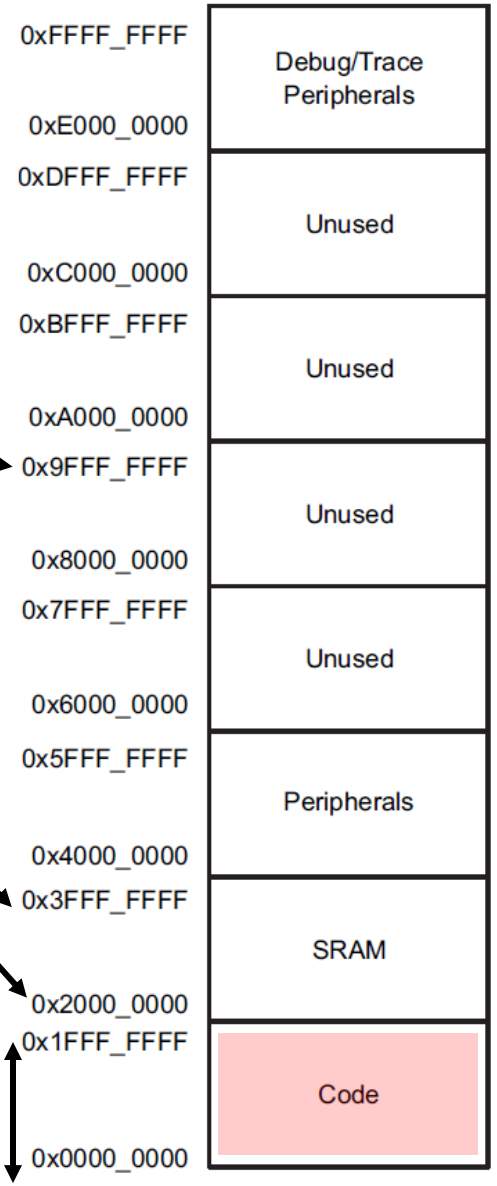
Many necessary elements are missing in the sketch below, in particular the configuration of the port (input or output, pull up or pull down resistors for input, drive strength for output). See lab session.

```
…
//declare p1out as a pointer to an 8Bit integer
volatile uint8_t*  p1out;

//P1OUT should point to Port 1 where LED1 is connected
p1out = (uint8_t*) 0x40004C02;

//Toggle Bit 0 (Signal to which LED1 is connected)
*p1out = *p1out ^ 0x01;
```

^ : XOR

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



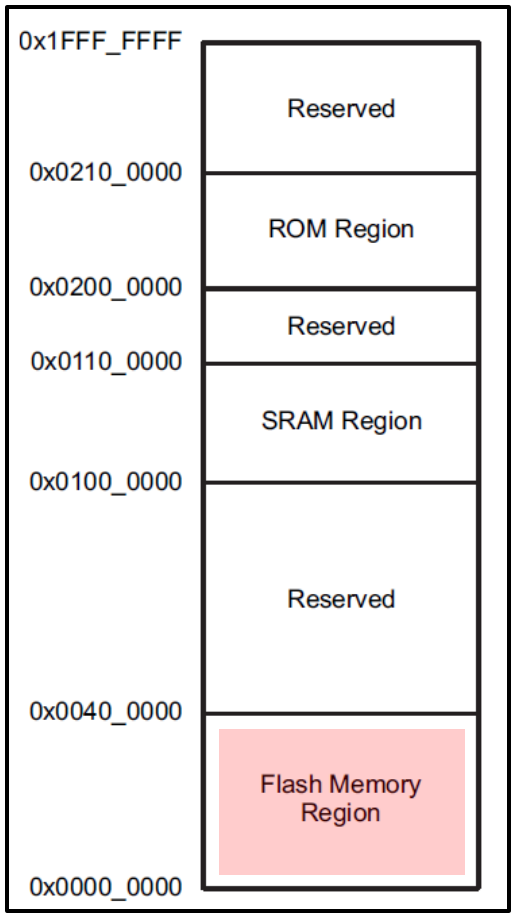
hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

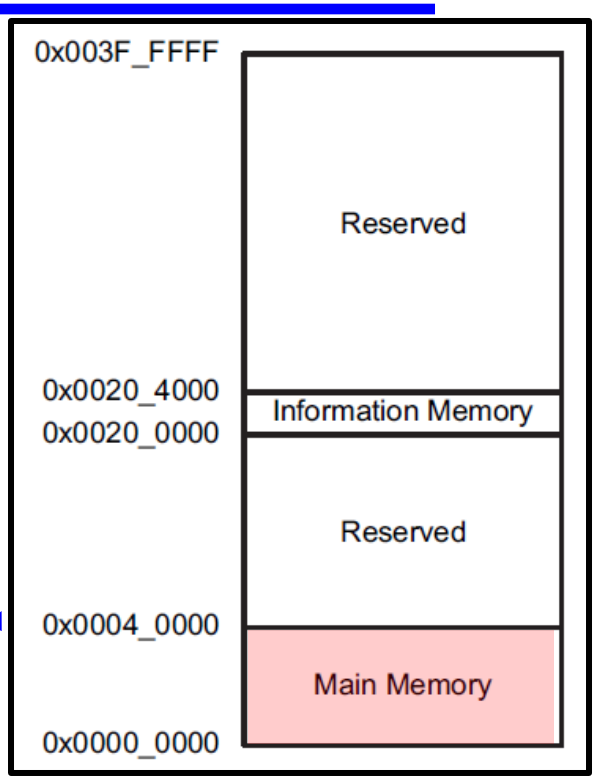
0011 1111 …. 1111

0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → 2^{29} different addresses

capacity = 2^{29} Byte = 512 MByte

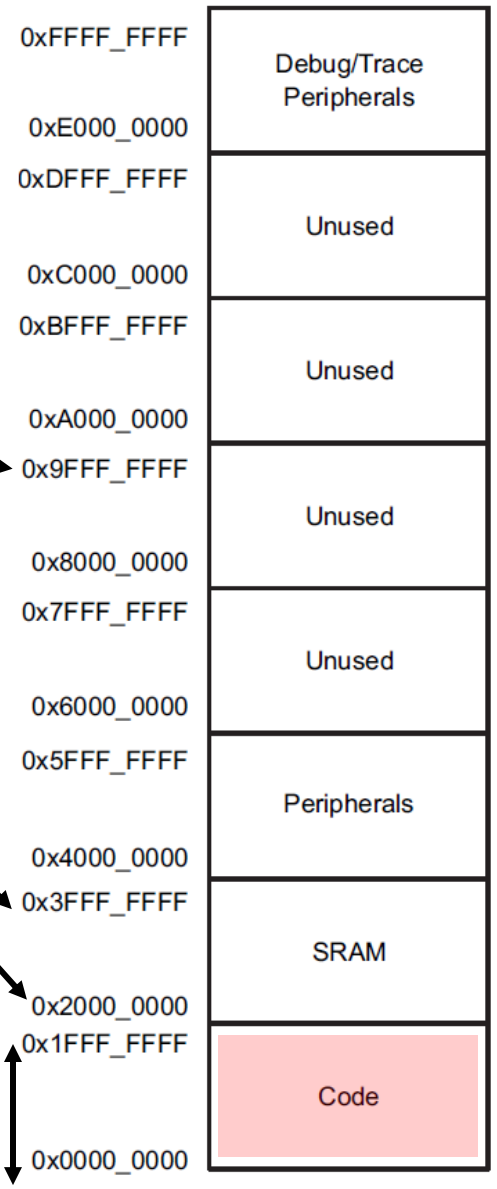




- 0x3FFF address difference = 4 * 2^{16} different addresses → 256 kByte maximal data capacity for Flash Main Memory
- Used for program, data and non-volatile configuration.

Example: Memory Map in MSP432 (Example MCU last year ES-Lab)

Memory map:



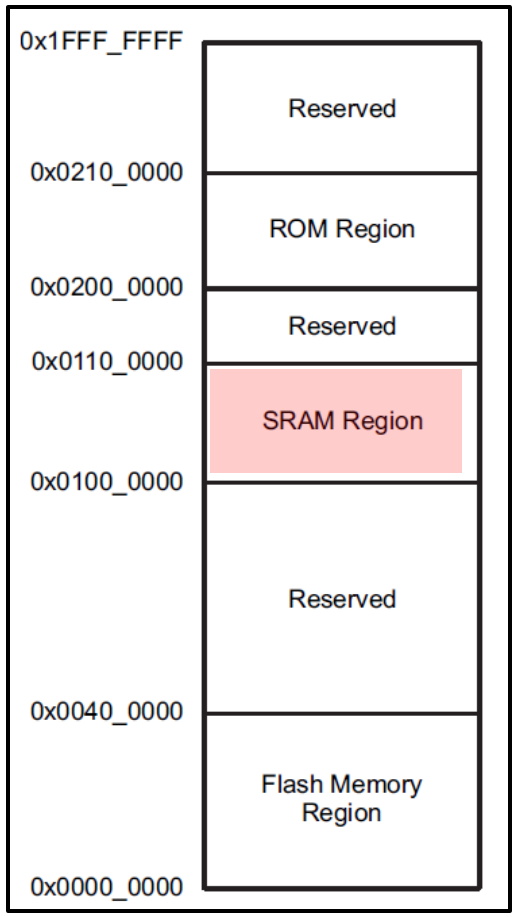
hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

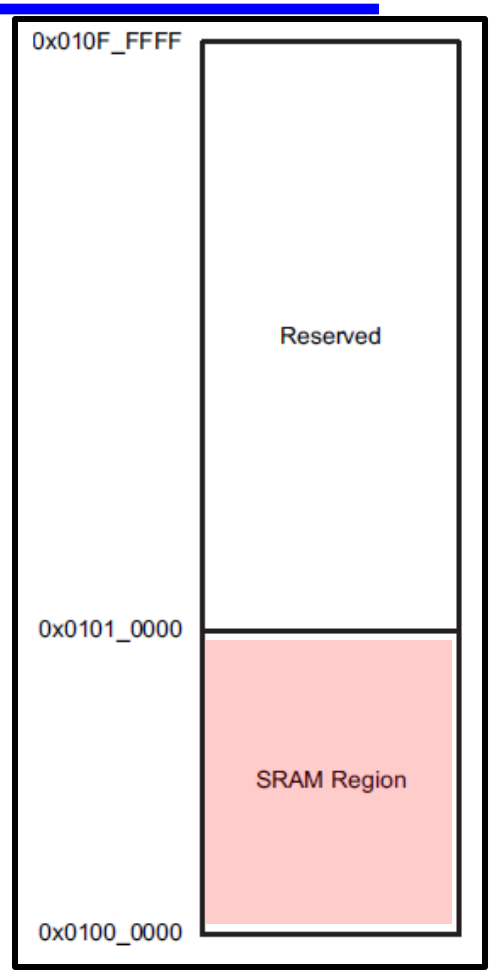
0011 1111 …. 1111

0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → 2^{29} different addresses

capacity = 2^{29} Byte = 512 MByte





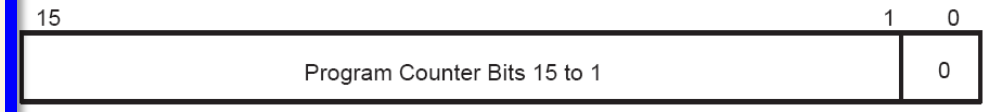
- 0x FFFF address difference = 2^{16} different addresses → 64 kByte maximal data capacity for SRAM Region
- Used for program and data.

CPU Internal Registers

```
0x90 <main>                    push    {r7, lr}
0x92 <main+2>                  add     r7, sp, #0
0x94 <main+4>                  bl      0x94 <main+4>
0x98 <main+8>                  bl      0x98 <main+8>
0x9c <main+12>                 bl      0x9c <main+12>
0xa0 <main+16>                 bl      0xa0 <main+16>
0xa4 <main+20>                 bl      0xa4 <main+20>
0xa8 <main+24>                 bl      0xa8 <main+24>
0xac <main+28>                 b.n     0xac <main+28>
0xae <SystemClock_Config>      push    {r7, lr}
0xb0 <SystemClock_Config+2>    sub     sp, #120        ; 0x78
0xb2 <SystemClock_Config+4>    add     r7, sp, #0
0xb4 <SystemClock_Config+6>    add.w   r3, r7, #24
0xb8 <SystemClock_Config+10>   movs    r2, #96 ; 0x60
0xba <SystemClock_Config+12>   movs    r1, #0
0xbc <SystemClock_Config+14>   mov     r0, r3
0xbe <SystemClock_Config+16>   bl      0xbe <SystemClock_Config+16>
0xc2 <SystemClock_Config+20>   mov     r3, r7
0xc4 <SystemClock_Config+22>   movs    r2, #0
0xc6 <SystemClock_Config+24>   str     r2, [r3, #0]
0xc8 <SystemClock_Config+26>   str     r2, [r3, #4]
0xca <SystemClock_Config+28>   str     r2, [r3, #8]
0xcc <SystemClock_Config+30>   str     r2, [r3, #12]
0xce <SystemClock_Config+32>   str     r2, [r3, #16]
0xd0 <SystemClock_Config+34>   str     r2, [r3, #20]
0xd2 <SystemClock_Config+36>   movs    r0, #0
0xd4 <SystemClock_Config+38>   bl      0xd4 <SystemClock_Config+38>
0xd8 <SystemClock_Config+42>   mov     r3, r0
0xda <SystemClock_Config+44>   cmp     r3, #0
0xdc <SystemClock_Config+46>   beq.n   0xe2 <SystemClock_Config+52>
0xde <SystemClock_Config+48>   bl      0xde <SystemClock_Config+48>
```

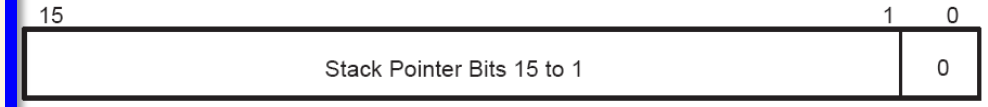
Program Counter (PC)

Points to the next instruction to be executed



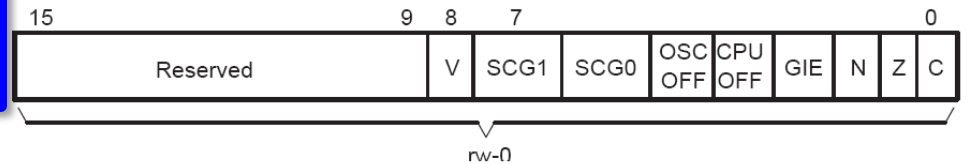
Stack Pointer (SP)

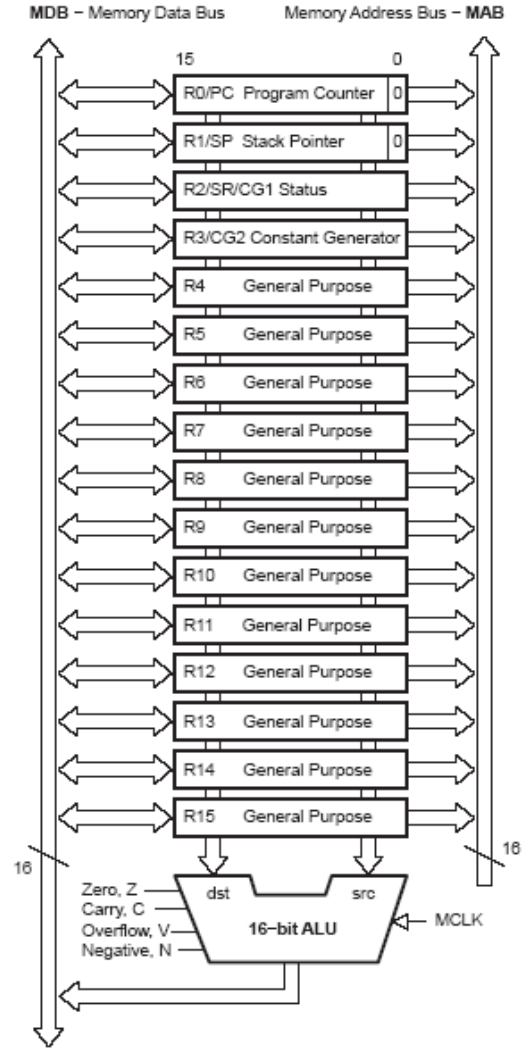
Store the return addresses of subroutine calls and interrupts



Status Register (SR)

Source or destination register





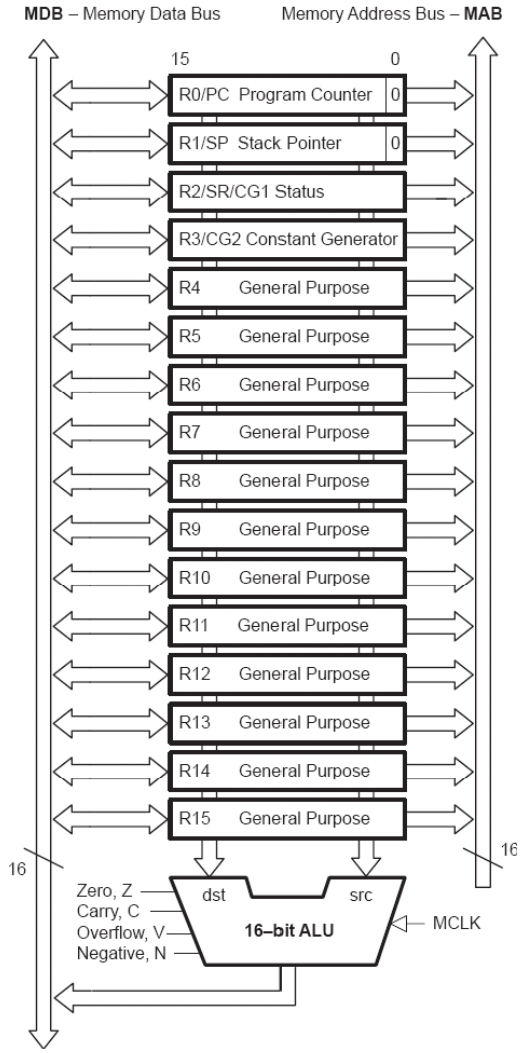
Dedicated Registers

Dedicated Registers:

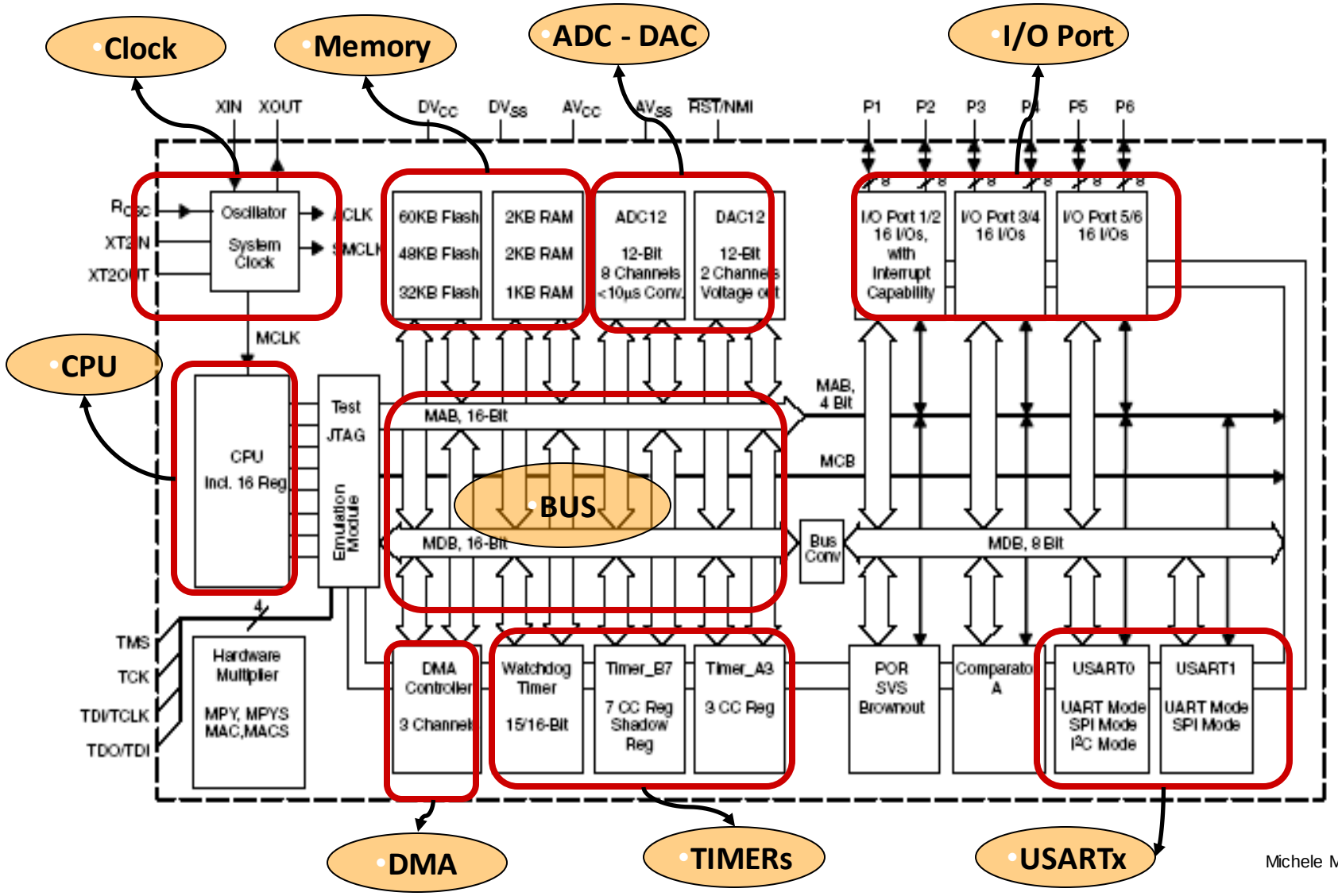
- Program Counter (PC):
  - Points to the next instruction to be read from memory and executed by the CPU.
- Stack Pointer (SP):
  - stack can be used by user to store data for later use (instructions: store by PUSH, retrieve by POP);
  - can be used by user or by compiler for subroutine parameters (PUSH, POP in calling routine; addressed via offset calculation on stack pointer (SP) in called subroutine);
  - used by subroutine calls to store the program counter value for return at subroutine's end (RET).
- Status Register (SR):
  - Stores status, control bits and system flags, updated automatically by the CPU.

General Purpose Registers

- General–Purpose Registers:
  - The general-purpose registers are adequate to store data registers, address pointers, or index values and can be accessed with byte or word instructions.
  - Used to execute arithmetical operations or to read/write memory.



Example of MCU Architecture

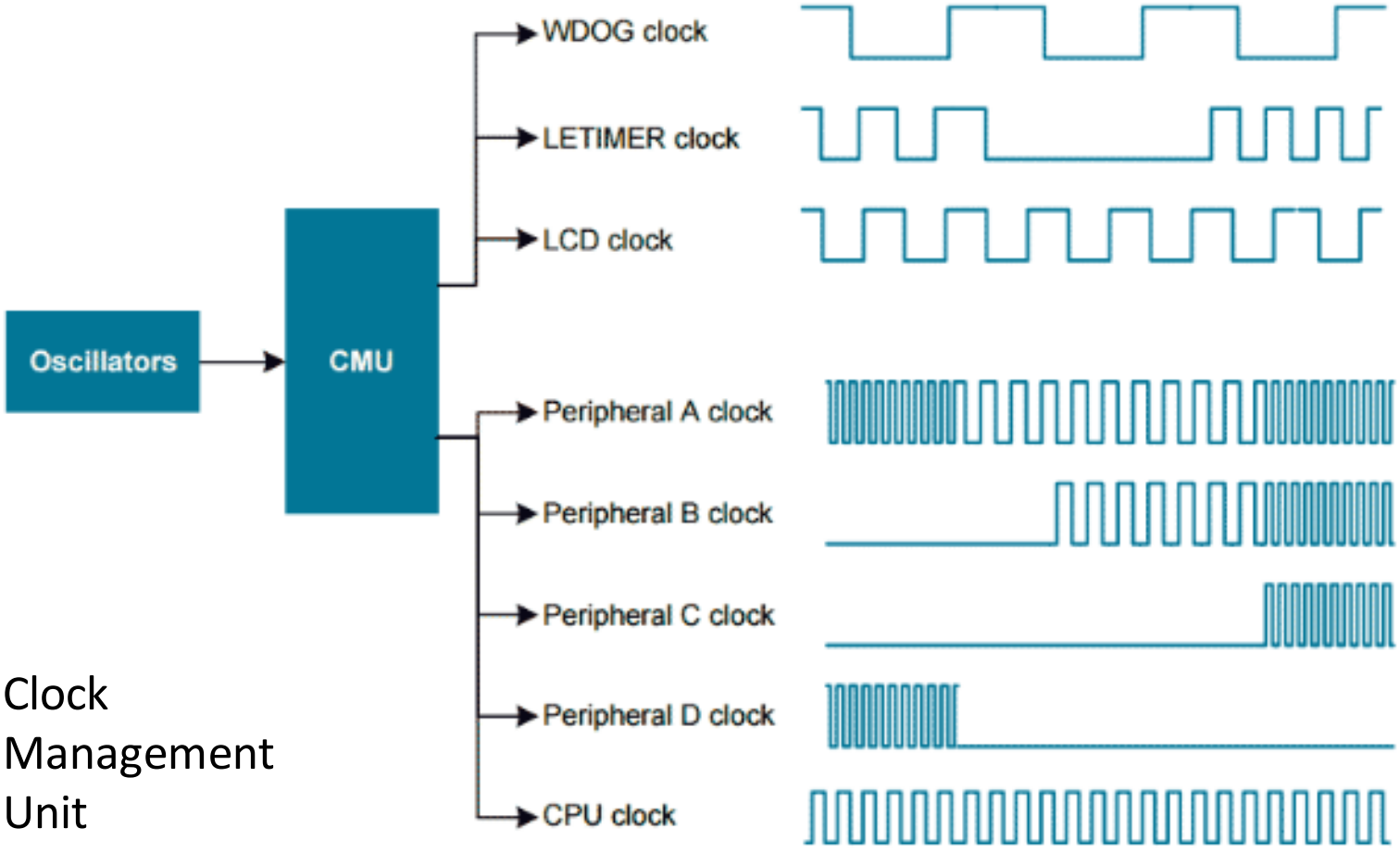


Clocks

- **Fast Clocks** CPU, Communications, Burst Processing
- **Low-power** RTC, Remote, Battery, Energy Harvesting
- **Accurate** Stable over °/V, Communications, RTC, Sensors
- **Failsafe** Robust–keeps system running in case of failure
- **Cheap** … goes without saying …

… or some combination of these features?

Is there a single clock in an embedded system?



Clock Management Unit

Dynamic power consumption

Power due to short-circuit current during transition

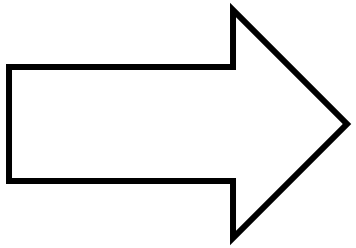
Power due to leakage current

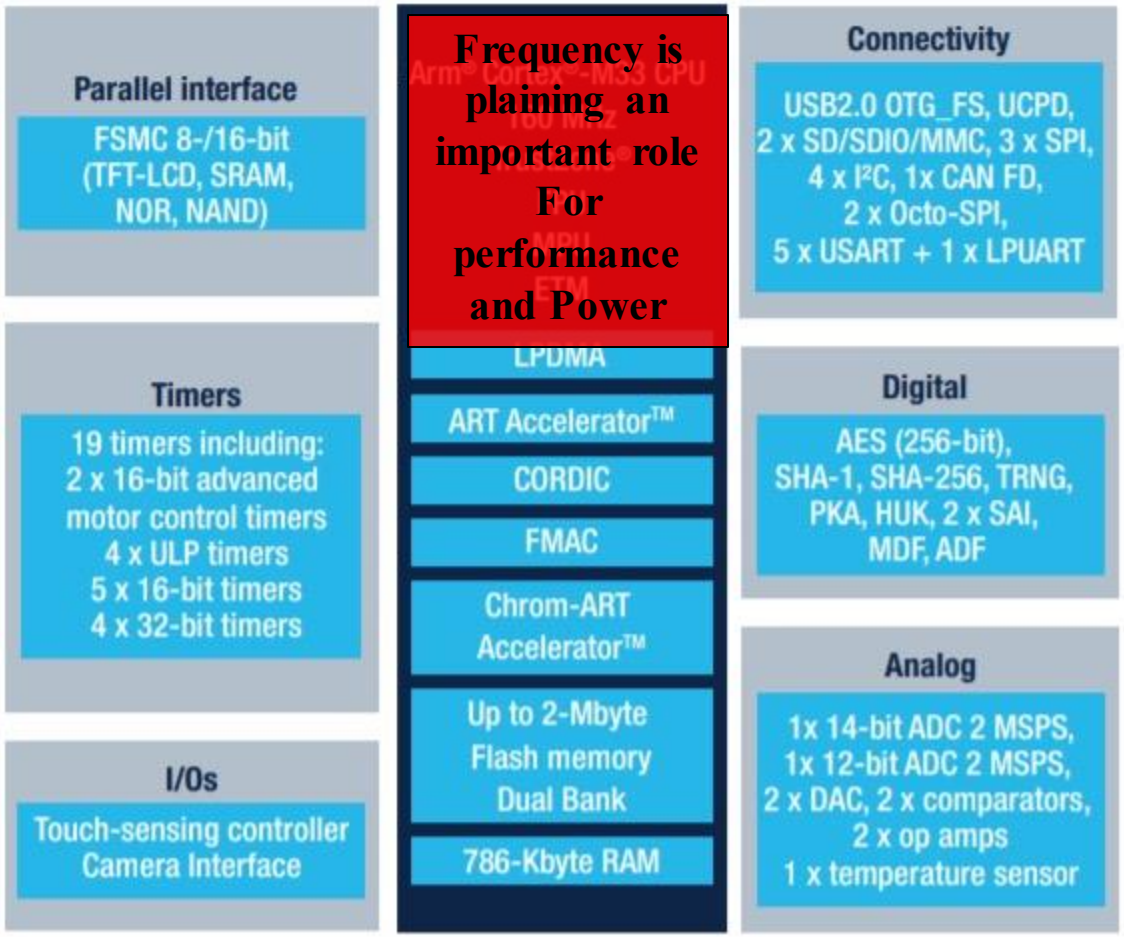
$$P = ACV^2f + \tau AVI_{short}f + VI_{leak}$$

But execution time of a task at parity of architecture is function of frequency

Trade-off energy vs latency especially in real time constrain

STM32U5- Block Diagram (ES-LAB)





STM32U5: Reset and Clock Controller. (ES-LAB)

Six internal clock sources

High-speed internal 16 MHz RC oscillator (HSI16)

Multi-speed internal RC oscillators for system clock and for peripherals kernel clock (MSIS and MSIK)

Low-speed internal 32 kHz RC oscillator (LSI)

High-speed internal 48 MHz RC oscillator (HSI48)

Secure high-speed 48 MHz RC oscillator (SHSI) dedicated to SAES

Two external oscillators

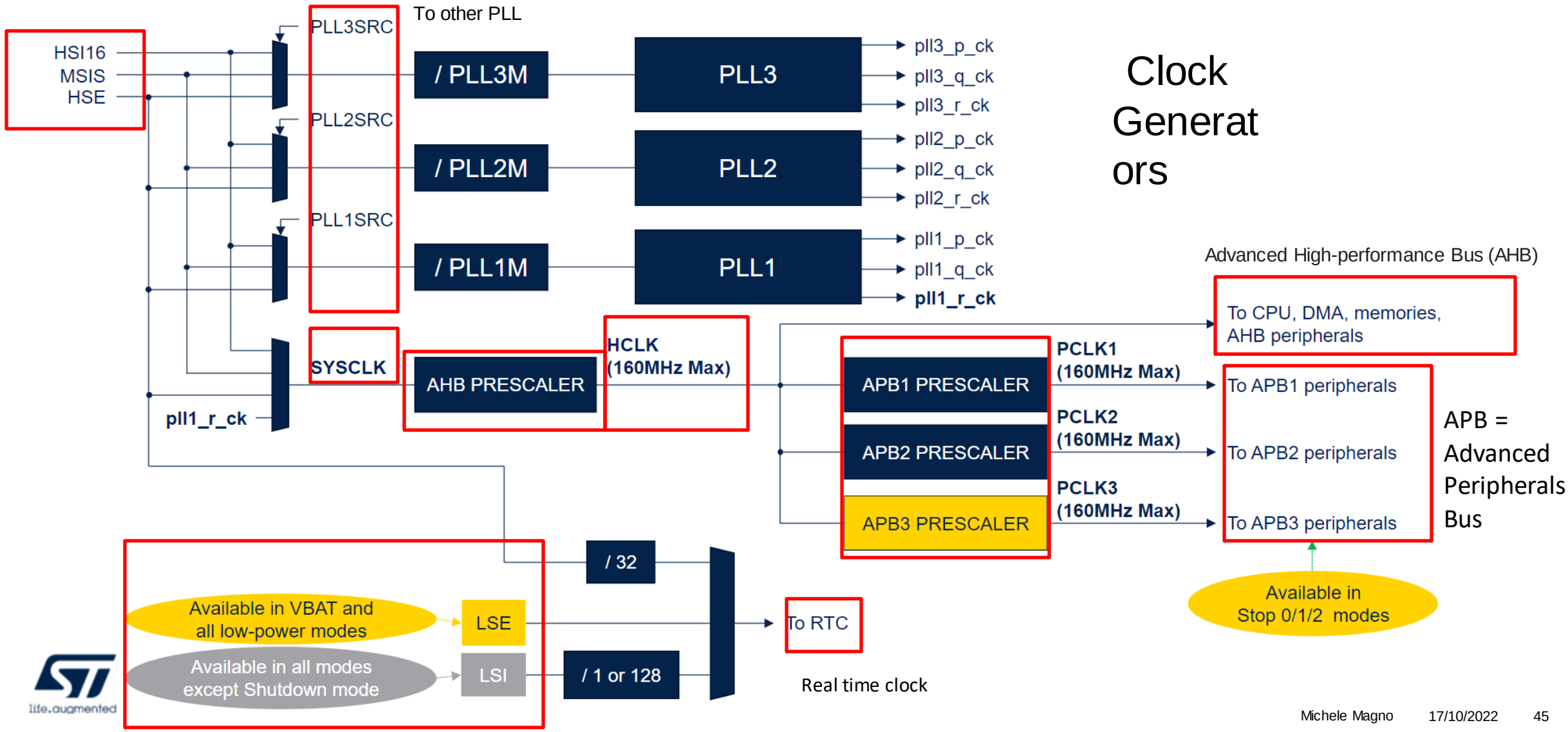
High-speed external 4 to 50 MHz oscillator (HSE) with clock security system

Low-speed external 32.768 kHz oscillator (LSE) with clock security system

Three PLL, each with 3 independent outputs

A phase-locked loop or phase lock loop (PLL) is a control system that generates an output signal whose phase is related to the phase of an input signal.

STL32F4: Simplify Clocks Tree



Comparison MSI vs HSI

	MSI (100 kHz to 48 MHz)		HSI16 (16 MHz)
	MSI mode (w/o LSE)	**PLL mode (w/ 32.768 kHz LSE)**	
Accuracy (typ.)	Over [0 - 85 C]: +/- 1.55 % Over [1.62 - 3.6 V]: +0.8/-4.5 %	Average accuracy = LSE accuracy Jitter < 0.25%	Over [0-85 C]: +/- 0.8 % Over [1.62 - 3.6 V]: +0.1/-0.2 %
Consumption (typ.)	100 kHz : 0.6 µA 800 kHz : 1.9 µA 1 MHz : 4.7 µA 8 MHz : 18.5 µA 16 MHz : 62 µA 48 MHz : 155 µA		150 µA
Startup time (typ.)	100 kHz : 10 µs 48 MHz : 2.5 µs	5% final frequency : 0.5 ms 1% final frequency : 1.5 ms max	1 µs

life.augmented

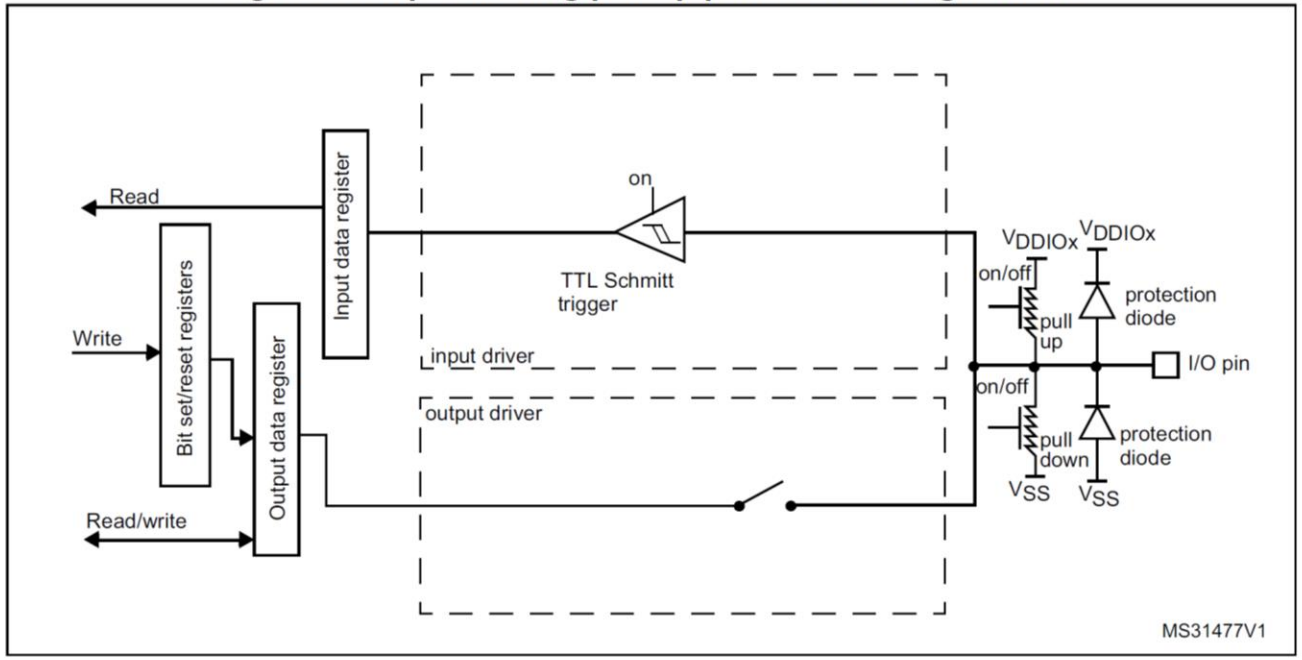
Lets see what is out of the MCU! General purpose In/Out pins GPIOs

General Purpose Input Out (GPIO): Theory

Each MCU pin can be used as a General Purpose digital input or output.

Figure 25 shows the input configuration of the I/O port bit.

Figure 25. Input floating/pull up/pull down configurations

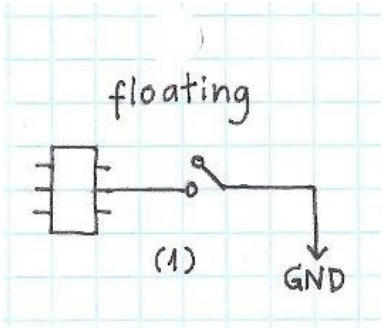


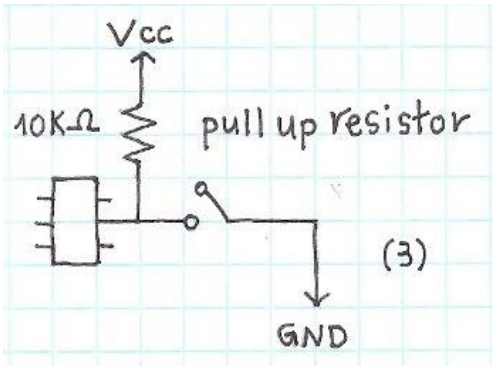
General Purpose Input Out (GPIO)

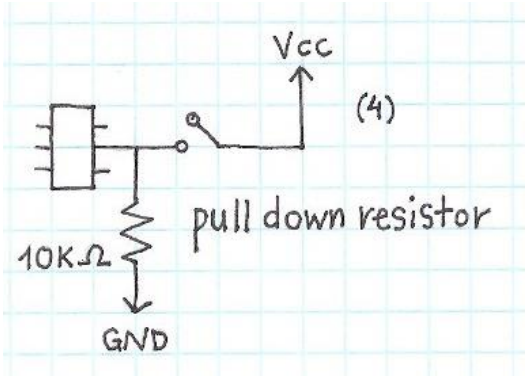
- Are used to control single pin devices (LEDS, Buttons, etc)
- First Hello World
- Are physically connected to other devices.
- Can have several functions

GPIO operating modes

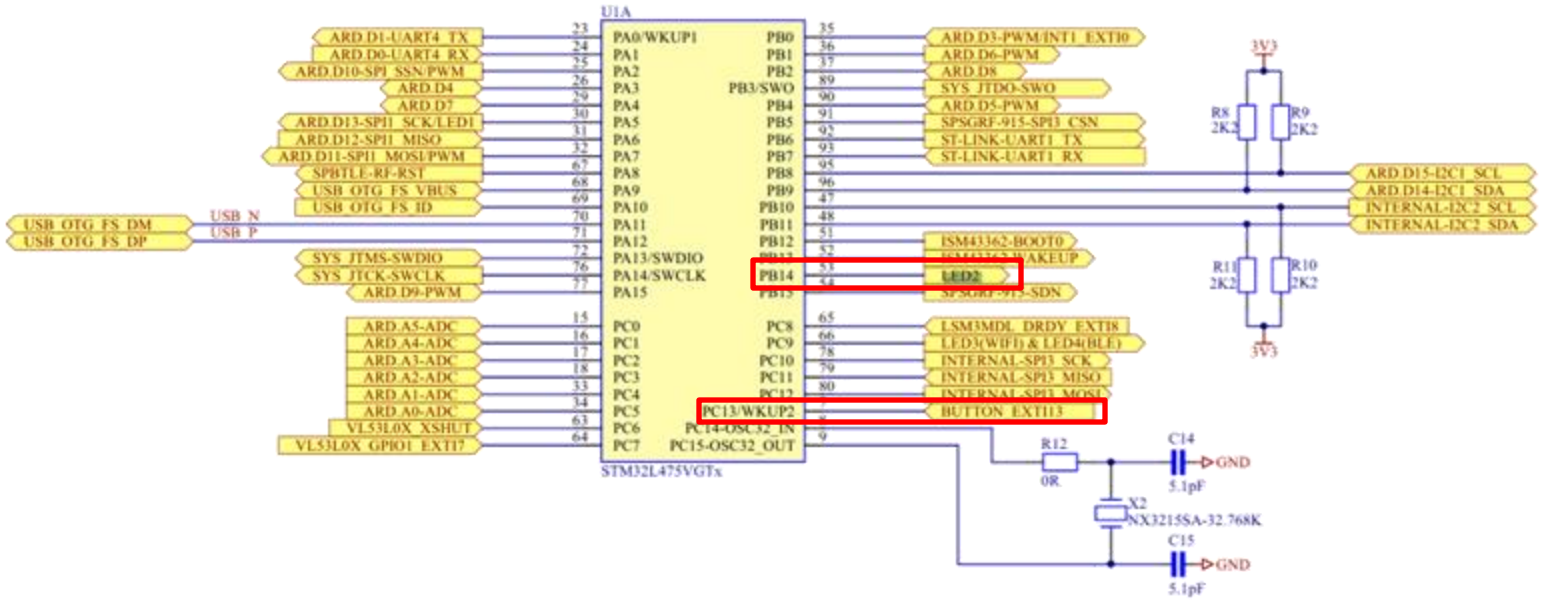
- Input mode
  - Floating
  - Input wit pull-up/down
  - Analog input mode
- Output mode
  - Push-pull, open drain





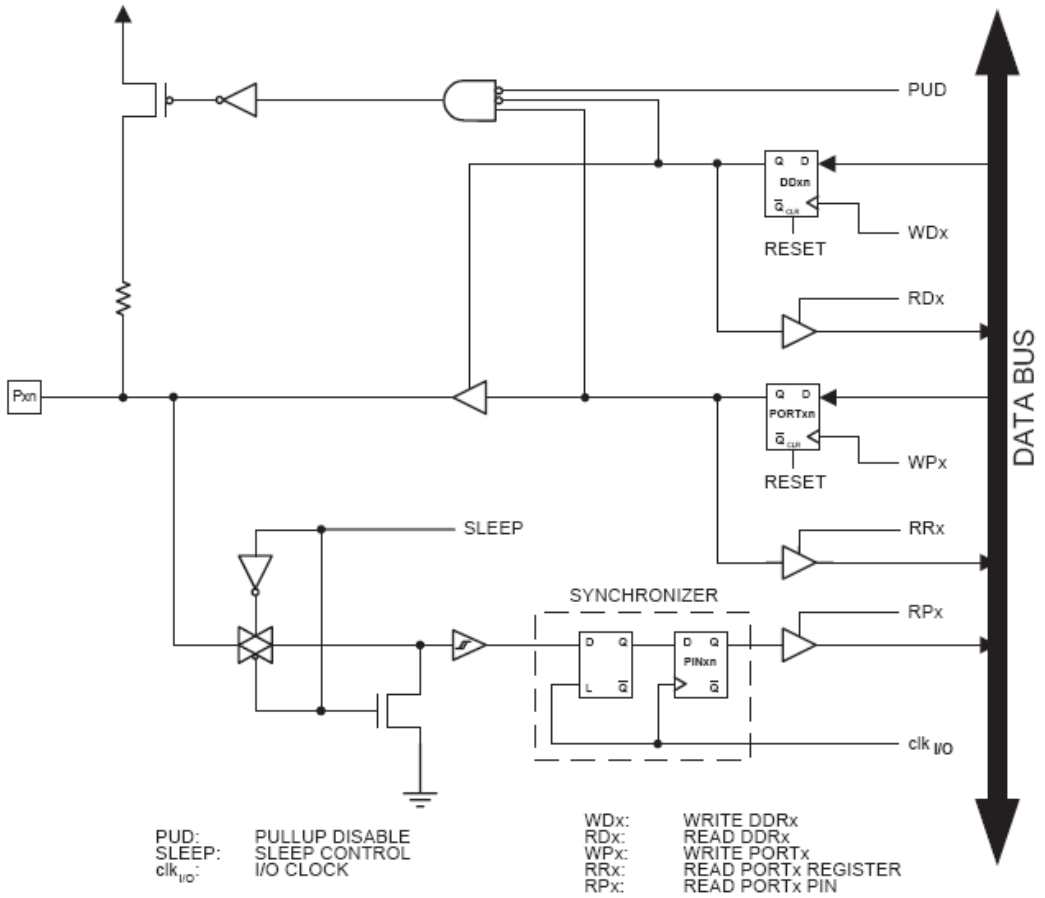


LED2 schematic and MCU connection

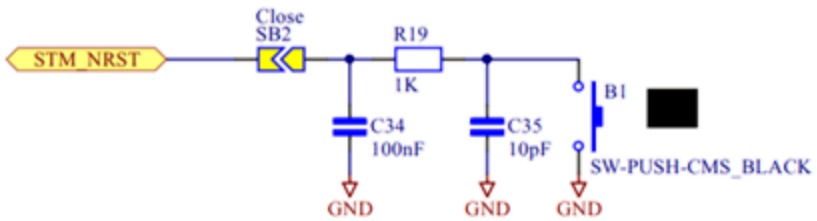


Inside GPIOs

- Each pin is independent
- *Ports* (out) and *Pins* (in) are **different**!!!

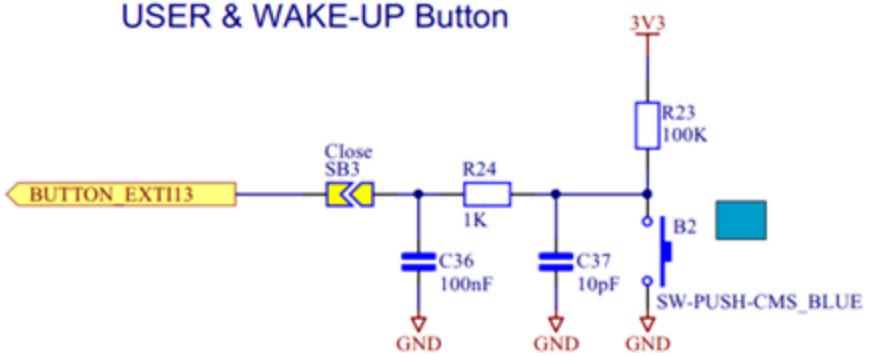


RESET BUTTON



100nF should be place close to the MCU
10pF and 1K should be place close to the button

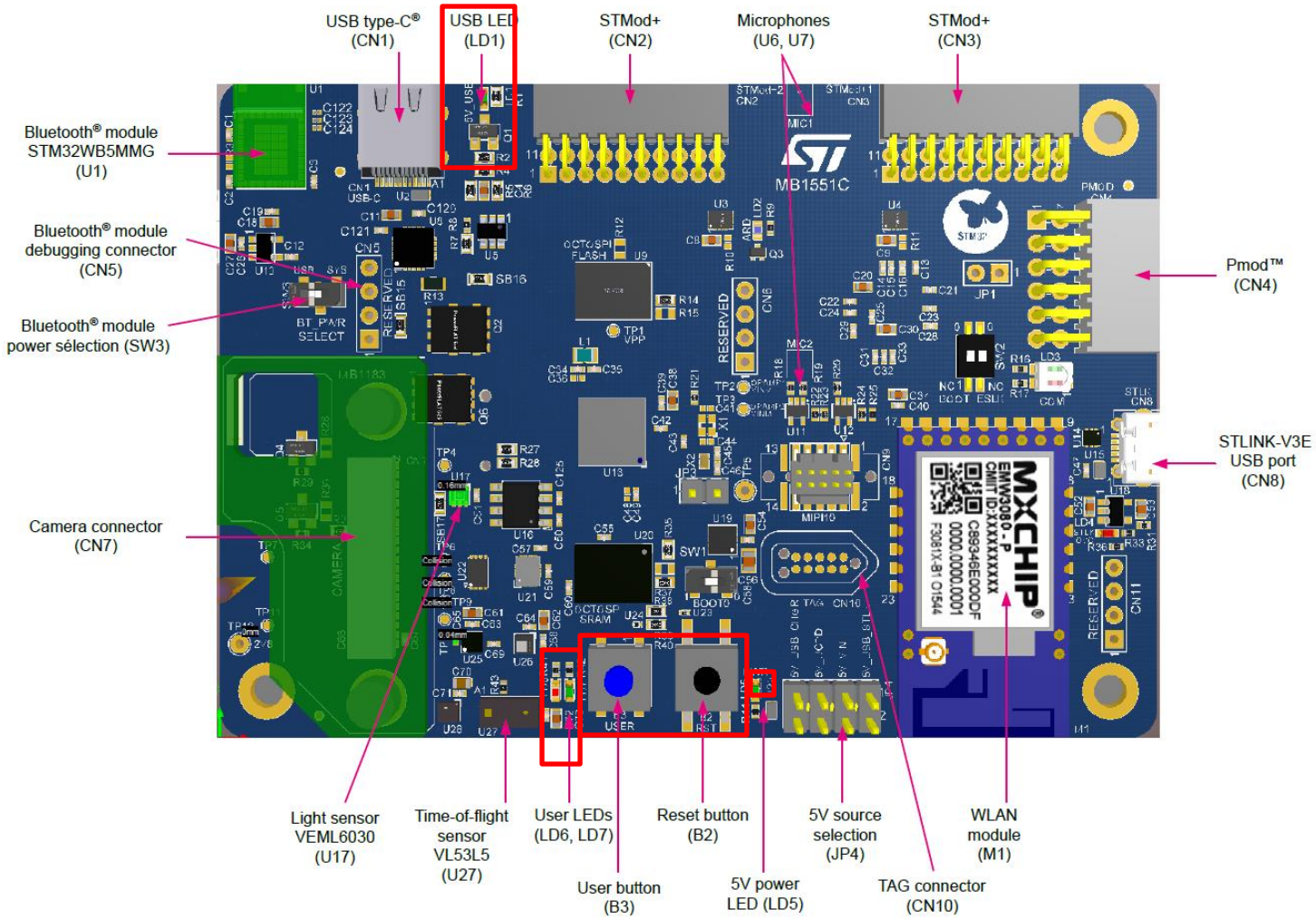
USER & WAKE-UP Button



100nF should be place close to the MCU
10pF and 1K should be place close to the button

UM2839 User manual (ES-LAB)

Figure 3. **B-U585I-IOT02A Discovery kit layout (top view)**



UM2839 section 7.1

I/O	Reference	Color	Name	Comment
NRST	B2	Black	RST	System reset
-	B1	Black	BT_RST	Bluetooth® module reset
PC13	B3	Blue	USER	User button
-	LD3	Bicolor (red and green)	COM	Blinking when flashing or debugging
-	LD4	Red	STLK_OVC	Red when the current is upper than 500 mA
PH7	LD7	Green	LD7	User LED lights up when PH7 is set to 0.
PH6	LD6	Red	LD6	User LED lights up when PH6 is set to 0.
-	LD5	Green	PWR	Available 5 V
PE13	LD2	Blue	ARD	ARDUINO® LED lights up when PE13 is set to 1.
-	LD1	Green	5V_USB	Available VBUS_C

What did you Learn?

- Compilation and deploying code on an embedded system
- Memory is crucial for MCU
  - What are the registers for data but also for instructions
  - Different type and architecture.
- Many clocks in a small device
  - Frequency, latency, energy tradeoffs
- External world
  - GPIO