WHERE IS MY PROGRAM?

- Does my microcontroller run C language code?
- Where does it store its instructions?
- How does the microcontroller know where to begin?
- What happens during a reset?
- How does a computer *program* a microcontroller
- Are there going to be burgers in today’s class?

MICROCONTROLLER PROGRAMMING

Daniel Martínez

I7266 - Programación de Sistemas Embebidos

CUCEI - UDG

MEMORY

Do you remember?

The 21st night of September?

Love was changing the minds of pretenders

While chasing the clouds away

MEMORY CLASSIFICATIONS

- Volatility
- Access method
- Read/write characteristics
- Physical location
- Functionality
- Technology
- Usage in CPU
- Data organization

VOLATILITY

- Volatile Memory:
  - Loses its stored information when power is turned off. Examples include RAM (Random Access Memory).
- Non-Volatile Memory:
  - Retains data even when power is turned off. Examples include CDs, flash memory, and hard drives.

ACCESS METHOD

- Random Access:
  - Allows data to be accessed in any random order. (RAM)
- Sequential Access:
  - Requires accessing data in a sequential manner. Examples include tape drives and optical drives.

READ/WRITE CHARACTERISTICS

- Read-Only:
  - Typically used for storing permanent or semi-permanent data, and the data cannot be easily modified. Examples include some optical drives.
- Read/Write:
  - Allows both reading and writing of data. Examples include RAM and Flash memory.

PHYSICAL LOCATION

- Internal Memory:
  - Embedded within the microprocessor or closely connected to it. Examples include Cache memory or SRAM.
- External Memory:
  - Separate from the microprocessor and connected via buses. Examples include EEPROM, hard drives, etc.

FUNCTIONALITY

- Primary Memory:
  - Fast and used for temporary storage (e.g., RAM).
- Secondary Memory:
  - Slower but provides larger storage capacity (e.g., hard drives).

TECHNOLOGY

- Dynamic Random Access Memory (DRAM):
  - Requires periodic refresh to maintain data integrity.
- Static Random Access Memory (SRAM):
  - More stable and does not require frequent refreshing.
- Electrically Erasable Programmable ROM (EEPROM):
  - Non-volatile but needs UV light to be erased.
- Flash Memory:
  - Non-volatile memory commonly used in storage devices.

USAGE INSIDE CPU

- Cache Memory:
  - High-speed memory used by the CPU to store frequently accessed instructions and data.
- Registers:
  - Fast, small, and internal memory locations within the CPU.

HELLO WORLD!

Our first program

NOT SO FAST...

THE MINIMAL SYSTEM

What is needed for a microcontroller to work?

A minimal system refers to the basic set of components required to create a functional electronic system.

The components in a minimal system may vary depending on the specific application, for embedded systems, some common elements include:

- **Microcontroller or Microprocessor:**
  - The central processing unit responsible for executing instructions and controlling the overall operation of the system.
- **Clock Source:**
  - A clock oscillator or crystal to provide the system with a timing reference, ensuring synchronization of operations.

- Memory: This includes both volatile memory (RAM) and non-volatile memory (ROM, Flash), for storing data and program instructions, respectively.
- Input/Output (I/O) Components: These could be sensors, actuators, communication interfaces, or other peripherals that allow the system to interact with its environment.

- Power Supply: A source of electrical power to provide the necessary voltage levels for the components in the system.
- Basic Support Components: Resistors, capacitors, and other passive components that are necessary for circuit stability and functionality.

ATMEGA328P MINIMAL SYSTEM

POWER SUPPLY

- VCC
- GND
- AVCC
- AREF

CLOCK SOURCE

- Low Power Crystal Oscillator
- Full Swing Crystal Oscillator
- Low Frequency Crystal Oscillator
- Internal 128kHz RC Oscillator
- Calibrated Internal RC Oscillator
- External Clock

RESET

HELLO WORLD!

No BS this time!

HELLO WORLD!

whut???

```c
#define F_CPU 16000000U

#include <avr/io.h>

int main(void)
{
    DDRD &= ~(1 << PORTD2);

    while (1)
    {
        PORTD |= (1 << PORTD2);
    }

}
```

TALKING COMPUTER LANGUGE

HEXADECIMAL

- The hexadecimal numbering system (*hex* for short) is used to represent *nibbles*.
  - Knowledge check:
    - A **bit** is a binary digit, i.e., *1* or *0*.
    - A **nibble** is a group of four bits, e.g.,*1011*.
    - A **byte** is a group of two nibbles (or eight bits), e.g., *11001111*
- Hexadecimal uses the first ten digits, plus, A, B, C, D, E and F to represent quantities. Let us see how it is used.

HEXADECIMAL

- In *hex* after the first ten numbers (from 0 to 9), the decimal values 10, 11, 12, 13, 14 and 15 are represented by A, B, C, D, E and F, respectively.
- Since four digits in binary can represent up to 15 (the binary number *1111*) then a single hex character can represent four bits.
- For example, the binary number *1100*, (a.k.a. 12 in decimal) is represented by the character *C*.
- This means we can represent a full byte with just two hex characters!

HEXADECIMAL

- To represent a byte in the common computer science slang, for example, 10100101 in binary (or 165 in decimal) we use the hexadecimal notation 0xA5, where:
  - “0x” means “we are talking hexadecimal”
  - “A” represents the first four bits (from left to right) in binary:
    - 1010 in binary is 10 in decimal. Since it is in the second character position, its value in decimal is multiplied by 16. $(10 \cdot 16 = 160)$
  - “5” represents the last four bits (least significative) in binary:
    - 0101 in binary is, unsurprisingly, 5 in decimal.

MEMORY

One more time…

WE ALREADY KNOW… DON’T WE?

- We have already seen different types of memory used in computer systems, however, we have yet to learn the uses for these kind of memory.
- Of all the classifications we explored, there are four key memory types:
  - ROM
  - RAM
  - Registers
  - Cache

ROM

Back to the burgers…

- In our “big company” examples, we know we must follow certain steps to achieve a desired output, these steps usually do not change. In computer systems, the place where we store our instructions is called ROM or Program Memory.
- ROM, in embedded systems, is located inside the microcontroller and it is not usually updated. It is intended to be programmed once and to need the very least number of updates possible.
- ROM is a type of non-volatile memory, since we need it to keep its contents for a long time.

RAM

More RAM, more FPS.

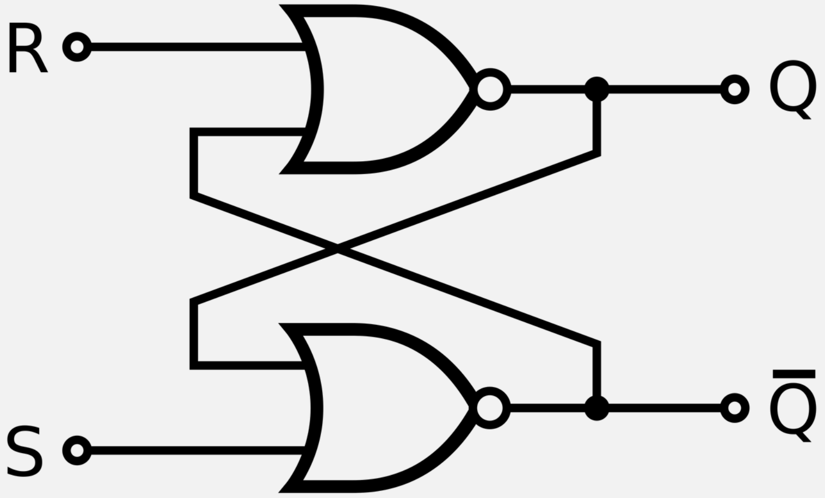
- RAM is our temporary storage; in our "big company" we need places to do what we are doing.
- We cannot prepare a burger on our hands, (and if we can, we should not), we need at least a table where we can place our ingredients.
- RAM, in embedded systems, is also located inside the microcontroller and it is written all the time while the microcontroller is working, it stores the data we need during the execution of a program.
- RAM is a usually a kind of volatile memory.

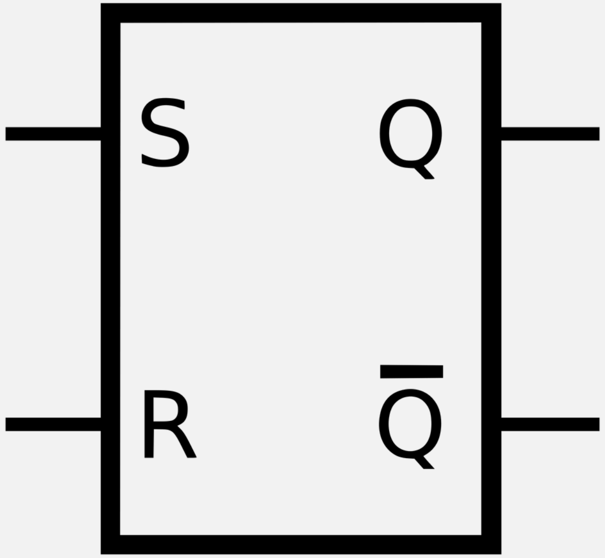
REGISTERS

More RAM, more FPS.

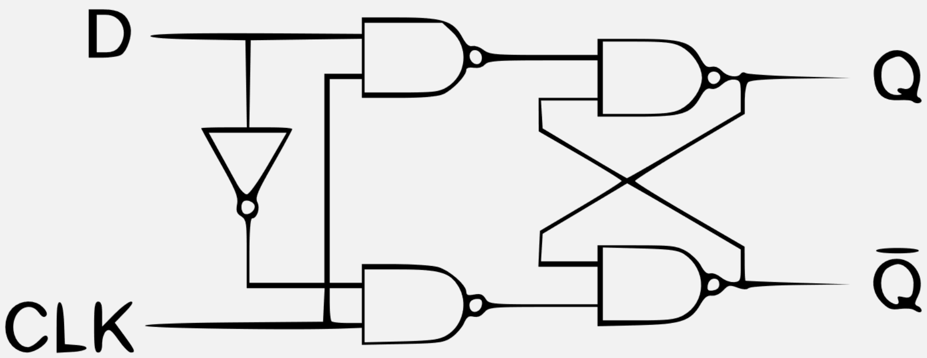
- The registers inside a microcontroller or microprocessor are small memory cells that are meant to hold some very important data that is useful for general purposes.
- In our “big company” these are like the boards where we place important company updates.
- Registers are usually implemented with some special logic cells like latches or flip-flops.
- When using the microcontroller, we will understand better how these registers are used and why.

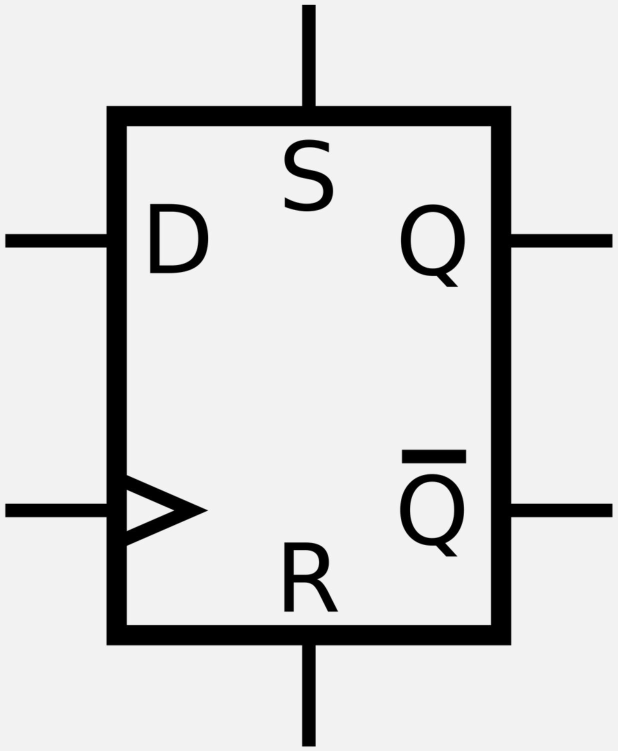
SR LATCH



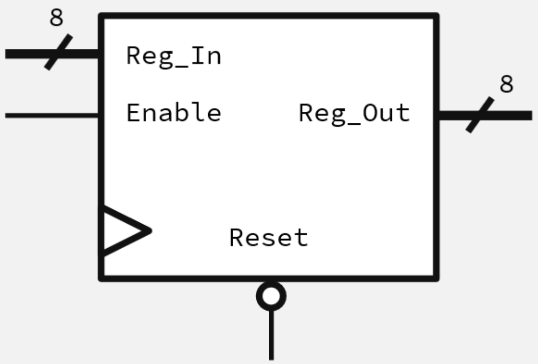


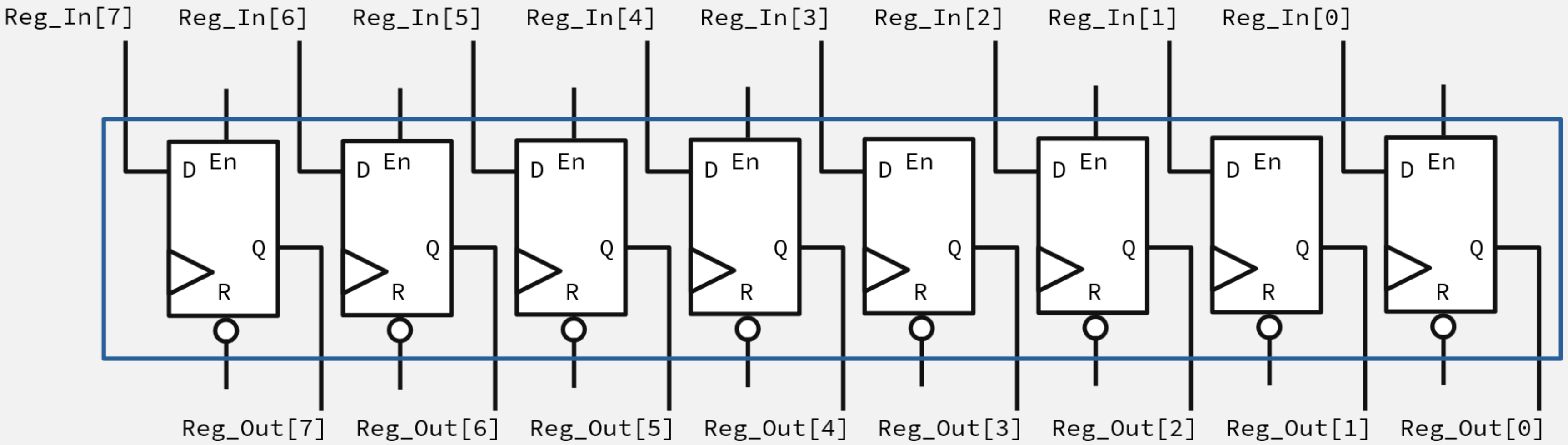
D FLIP-FLOP





8-BIT REGISTERS





ADDRESSES

Where you at rn?

ADDRESSES AND RELATED CONCEPTS

- A memory address is a unique identifier for a specific location in the microcontroller’s memory.
- We use addresses to identify *addressable units*. In most systems the smallest addressable unit is the byte, so, each memory cell can store a byte.
- The *address space* is the range of possible memory addresses that out control unit can generate.
- *Word Addressing* is accessing to memory with the addresses for words, this means, groups of bytes of a predefined size.
- *Endianness* refers to the order of multi-byte datatypes. We have *Little-Endian* and *Big-Endian*.

THE MEMORY MAP

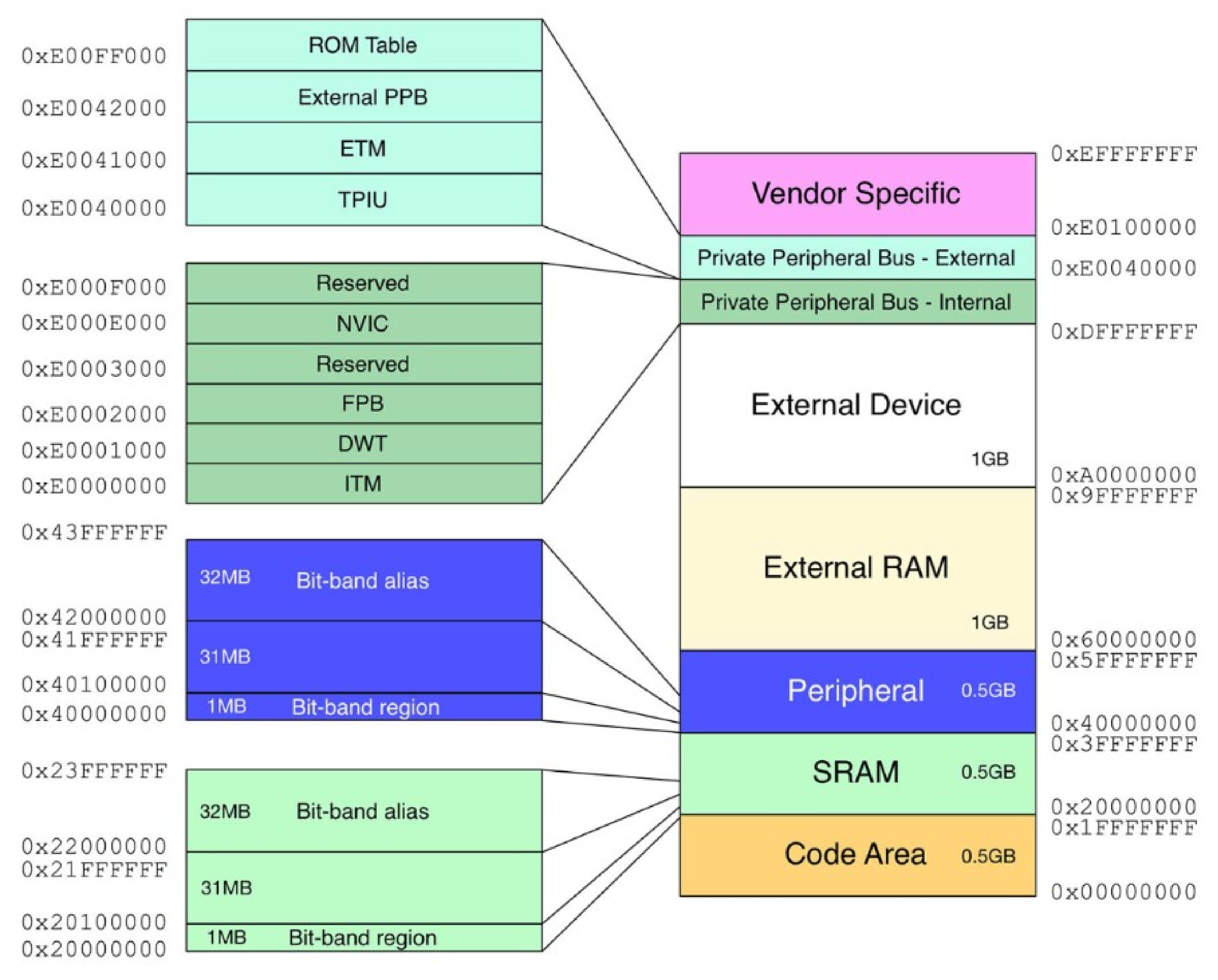
WHAT IS A MEMORY MAP?

- A memory map is the way a to describe how a computing system has its memory organized.
- It can represent different physical memory devices into one single address space.
- It can also represent multiple regions of a single memory device into the address map.
- We usually care about certain special *memory regions* that we will cover in detail later, this is the overview:

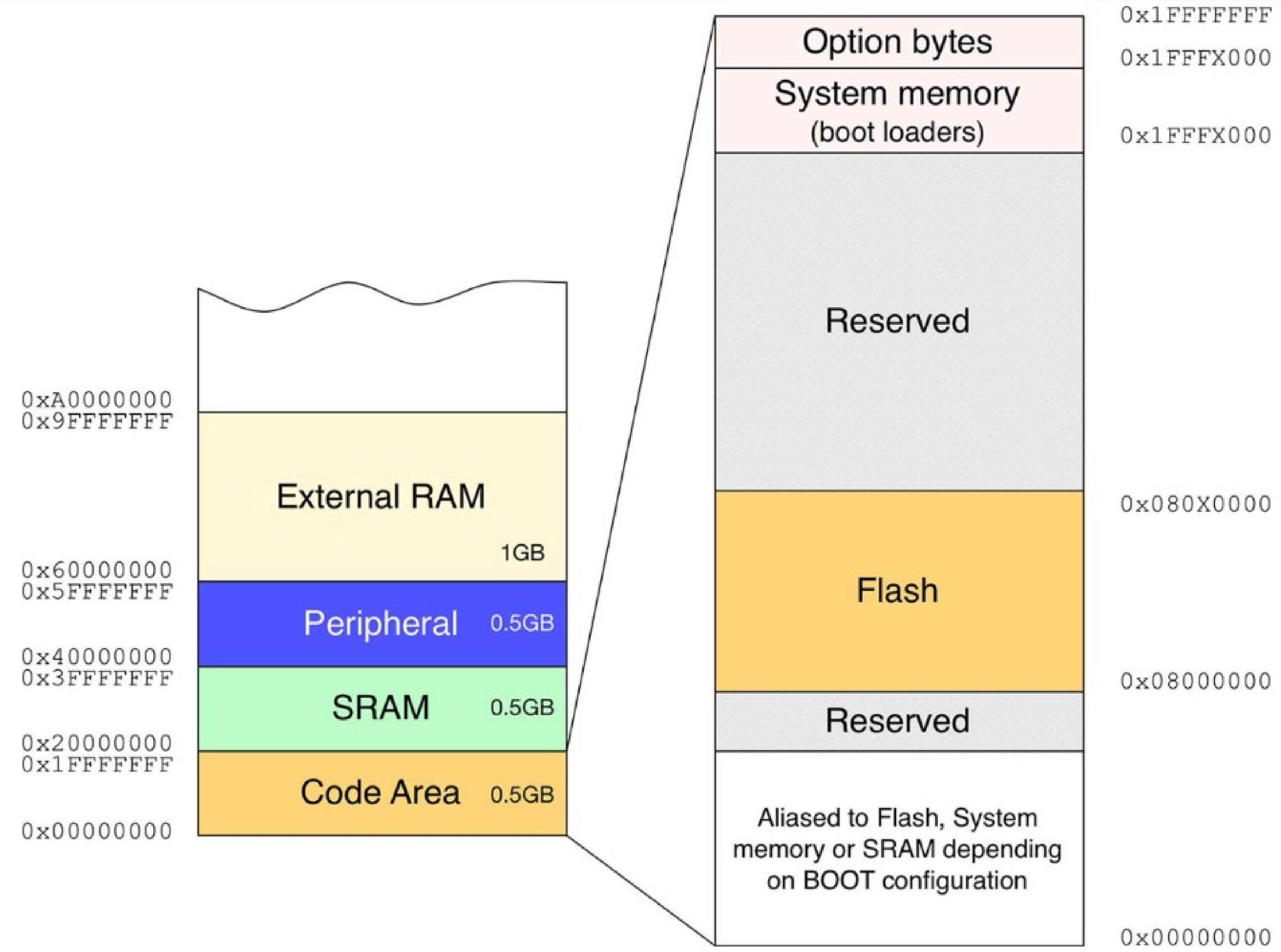
MEMORY REGIONS

- Program memory
  - This is the region where the code is stored, this refers to the ROM device.
- Data memory
  - This is the region where variables and constants are stored, this is usually in the RAM device.
- Registers
  - These are also mapped here.

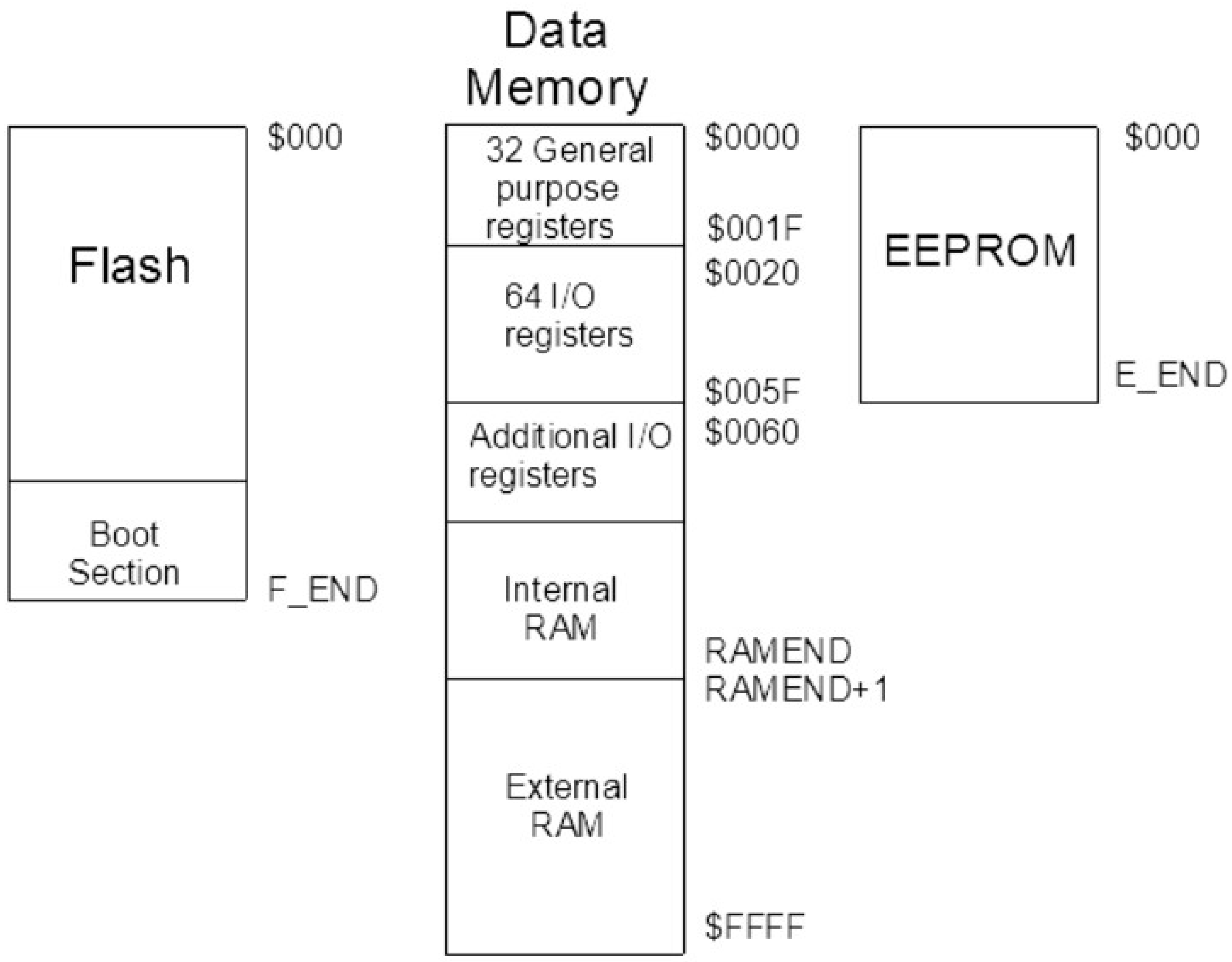
ARM CORTEX-M MEMORY MAP



0xE00FF000
ROM Table
0xE0042000
External PPB
0xE0041000
ETM
0xE0040000
TPIU
0xE000F000
Reserved
0xE000E000
NVIC
0xE0003000
Reserved
0xE0002000
FPB
0xE0001000
DWT
0xE0000000
ITM
0x43FFFFFF
32MB
Bit-band alias
0x42000000
0x41FFFFFF
31MB
0x40100000
1MB
Bit-band region
0x40000000
0x23FFFFFF
32MB
Bit-band alias
0x22000000
0x21FFFFFF
31MB
0x20100000
1MB
Bit-band region
0x20000000
0xEFFFFFFF
Vendor Specific
0xE0100000
Private Peripheral Bus - External
0xE0040000
Private Peripheral Bus - Internal
0xDFFFFFFF
External Device
1GB
0xA0000000
0x9FFFFFFF
External RAM
1GB
0x60000000
0x5FFFFFFF
Peripheral
0.5GB
0x40000000
0x3FFFFFFF
SRAM
0.5GB
0x20000000
0x1FFFFFFF
Code Area
0.5GB
0x00000000

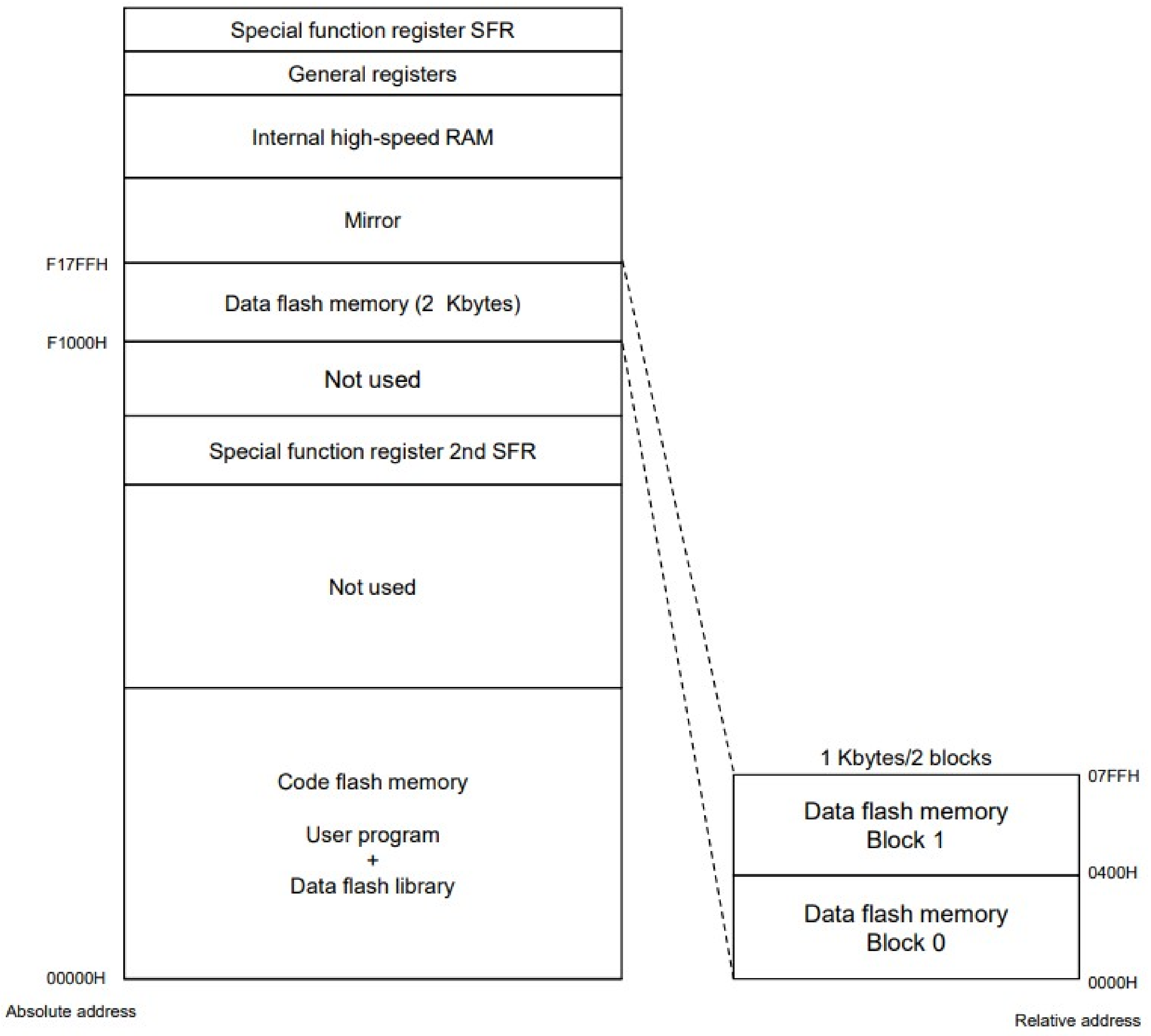

0xA0000000
0x9FFFFFFF
External RAM
1GB
0x60000000
0x5FFFFFFF
Peripheral
0.5GB
0x40000000
0x3FFFFFFF
SRAM
0.5GB
0x20000000
0x1FFFFFFF
Code Area
0.5GB
0x00000000
0x1FFFFFFF
Option bytes
0x1FFFX000
System memory
(boot loaders)
0x1FFFX000
Reserved
0x080X0000
Flash
0x08000000
Reserved
Aliased to Flash, System memory or SRAM depending on BOOT configuration
0x00000000

AVR MEMORY MAP



Data Memory
Flash
Boot Section
$000
F_END
32 General purpose registers
64 I/O registers
Additional I/O registers
Internal RAM
External RAM
$0000
$001F
$0020
$005F
$0060
RAMEND
RAMEND+1
$FFFF
EEPROM
$000
E_END

AVR MEMORY MAP


Special function register SFR
General registers
Internal high-speed RAM
Mirror
F17FFH
Data flash memory (2 Kbytes)
F1000H
Not used
Special function register 2nd SFR
Not used
Code flash memory
User program
+
Data flash library
00000H
Absolute address
1 Kbytes/2 blocks
07FFH
Data flash memory
Block 1
0400H
Data flash memory
Block 0
0000H
Relative address

I/O PORTS AND REGISTERS

HOW DO WE TALK TO THE EXTERIOR?

- Microcontrollers have input/output pins that can be used to receive and send signals with other devices.
- In AVR microcontrollers, we usually have *ports*, which are controlled by *i/o registers*, for example, in the Atmega328p there are four ports, namely PORTA, PORTB, PORTC and PORTD, and the behaviour and data in these ports is configured and read via the DDRx, PORTx and PINx registers.

I/O PORTS BASIC FUNCTIONALITIES

- We will employ the ports in the following manner:
- To set the direction of a port, we will use the Data Direction Register (DDRx).
  - If DDxn is written logic one, Pxn is configured as an output pin, if DDxn is written logic zero, Pxn is configured as an input pin.
- Remember, in this notation DDxn can be read as *the bit "n" in the Data Direction Register for port "x"*.
- This all means writing 00000001 binary to DDRx will enable bit 0 of port "x" as output.

I/O PORTS BASIC FUNCTIONALITIES

- Example 1:
- I look at the datasheet and I choose pin PC5 to output a digital signal.
- From the name of the pin I know that I need to search for the register DDRC and I need to write a logic one into the bit 5.
- In binary, a one in the bit 5 can be set in different equivalent ways. Let us explore some of them.

I/O PORTS BASIC FUNCTIONALITIES

- The objective is to send the binary value «00100000» to the DDRC register.
- We should not write that value directly to the register, this is, we cannot do *DDRC = 00100000* since that number would be interpreted as decimal.
- It is possible, however, to indicate that it is a binary number appending the suffix “0b”, e.g.: 0b00100000. Note that this is <u>not</u> recommended since it is out of the standard.

I/O PORTS BASIC FUNCTIONALITIES

- Since it is not recommended to use «0b00010000», we can use other methods to write that value to the DDRC register, the first one is using its decimal or hexadecimal value:
  - In decimal, that value represents 16. So, *DDRC = 16* is good.
  - In hex, that value represents 0x10. So, *DDRC = 0x10* is good too.
- Another way is doing a bit shift, if we want to put a *1* in the 5^{th} bit of a byte, we can do *1 ≪ 5*.
- Indeed, DDRC = 1 ≪ 5 would work just fine.

I/O PORTS BASIC FUNCTIONALITIES

- The aforementioned methods would do the job, but those are not the safest methods.
  - First, it is a good practice that instead of assigning a logic one to a bit, we OR-operate it so the rest of the bits are not affected.
  - Second, while using operators in C, we make use of parenthesis to make clearer the intention of the operator employed.
  - Third, we can use macros and constants to make things clearer.
- Here are some real life examples of this.

I/O PORTS BASIC FUNCTIONALITIES

- `DDRC |= (1 << PORTC5);`
  - PORTC5 is a preprocessor constant equivalent to *5*.
- `DDRC |= _BV(PORTC5);`
  - The macro *_BV* is a preprocessor function that does the shift during preprocessing, making things faster.

I/O PORTS BASIC FUNCTIONALITIES

- Example 2:
- I look at the datasheet and I choose pin PB1 to read a digital signal.
- From the name of the pin I know that I need to search for the register DDRB and I need to write a logic one zero the bit 1.
  - If I want to set a zero, this time I need to AND-operate the value in order to preserve the other bits unaltered, the value that we will use will be referred as *mask* from now on.

I/O PORTS BASIC FUNCTIONALITIES

- Our mask needs to be the opposite of «00000001», this means «11111110», however, if we try to OR-operate that with a register, we would be setting all the other bits to 1. This is why we need to AND-operate with it.

I/O PORTS BASIC FUNCTIONALITIES

- `DDRB &= ~(1 << PORTB1);`
- `DDRB &= ~(_BV(PORTB1));`

EXERCISE 1

PORT MANIPULATION

- Gather in groups.
- Every group has to figure out how to make the following pattern:
- The pattern must repeat indefinitely.
- You can draw the algorithm, write it in pseudocode or try to implement it in C.

THE COMPILER

Est-ce que tu comprends ?

WHAT IS A COMPILER

WHAT IS A COMPILER?

- A compiler is a computer program that *translates* source code from a high level programming language into a low level programming language. It does its job in multiple steps.
- In this lecture we will analize tools related to the GNU Compiler Collection for AVR devices.
- We will use avr-gcc to compile our code, and it does almost everything we want with a single command!

WHAT IS A COMPILER?

- To understand what the command does we have to read the documentation, however, that will take a lot of time, so we are going to get our hands dirty and learn in the process.
- I have a program, the one that turns an LED on. I am going to compile it with the following command:
  - **avr-gcc main.c -mmcu=atmega328p**
- What is the output? What is next? Let us see…

AVR-GCC

AVR-GCC

- avr-gcc is the program in charge of compiling our source code, we need to provide options to compile and a file to compile.

  (show **avr-gcc -help** output)

- If we only provide a file, we will not be able to compile successfully

  (show **avr-gcc main.c** output)

- If we provide a file and the option for the microcontroller we are using, then we get a file called **a.out** as output.

  (show **avr-gcc -mmcu=atmega328p main.c** output)

AVR-GCC

- We can indicate where we want out output.
  - (show **avr-gcc -mmcu=atmega328p main.c -o main.elf** output)
- That is the usual way to employ avr-gcc, but we can also run it so it shows us the intermediate steps. The first step is called *preprocessing*.

THE PREPROCESSOR

THE PREPROCESSOR

- Preprocessing is one of the first steps to compile software, it takes certain necessary steps for the compiler to understand what we want to compile.
- The C preprocessor performs these two main actions:
  - Remove comments
  - Substitue preprocessor directives:
    - Adding includes, substituting constants, expand macros and pragmas

THE PREPROCESSOR

- `avr-gcc -mmcu=atmega328p main.c -E -o main.i`
- `avr-gcc -mmcu=atmega328p main.c -g -fdump-tree-all-graph -O1`
- `xdot result.dot`

THE COMPILER

THE COMPILER

- Compiling requires the preprocessed code and three main analysis functions:
  - Lexical analysis
  - Sintax analysis
  - Semantic analysis
- The C compiler then:
  - Generates intermediate code
  - Optimizes it
  - Gives its output in assembly code

THE COMPILER

- avr-gcc -mmcu=atmega328p main.c -S -o main.s

- vim main.s

THE ASSEMBLER

THE ASSEMBLER

- Once the code is compiled into assembly, it is assembled into machine language:
- It translates, process directives and generates the object code.

THE ASSEMBLER

- `avr-gcc -mmcu=atmega328p main.c -o main.elf`
- `hexyl main.elf`

THE LINKER

THE LINKER

- The linker is in charge of linking (surprisingly) all the object files into a single file and bind the adresses according to the memory map provided.
- This is the last part of the process and this usualy ends with a .elf file.

THE LINKER

- avr-objdump main.elf -txS
- avr-objcopy -O ihex main.elf main.hex
- Hexyl main.hex

ROTABIT AND PATTERNS

ROTABIT AND PATTERNS

Check examples under

I7266/mega32A/002_Patterns/

- Once you get the hang of it, try to replicate the following patterns:
  - Binary count
  - Sweep
  - Rotabit
  - Wave
  - 2-1-toggle

SOFTWARE ARCHITECTURE

SOFTWARE ARCHITECTURE

- This term refers to the high-level design and organization of software components, modules, and subsystems that collectively form the software stack running on the embedded device.
- It encompasses the overall structure, relationships, and interactions between software elements, as well as the principles and patterns guiding their design and implementation.

THE C PROGRAM STRUCTURE

C PROGRAM STRUCTURE

- In embedded systems, a C program usually contains the following:
  - Configuration and Initialization
  - Main function and infinite loop
  - Utility functions and libraries
  - Hardware control functions
  - Boot or startup code

C PROGRAM STRUCTURE

- See the example from ArduinoClockv15

C PROGRAM STRUCTURE

- The parts of a program that we have used so far are two:
  - Configuration and initialization
  - Main loop
- The first refers to the code we use to set up the microcontroller to do what we want, for example, setting up outputs and inputs.
- The second refers to a software loop in which we put the code that we want the microcontroller to always execute.

INTERRUPTS

INTERRUPTS

- We are going back to the analogies: think about you in an evening having lots of stuff to do.
- Imagine you have no food in the fridge and you choose to cook some rice.
- The rice needs to cook for a few minutes, but you do not know the exact time, so you need to keep an eye on the pot so you do not have to eat burnt rice.
- We will call this: *Polling*.

INTERRUPTS

- The *event* that we are expecting is the rice being cooked, an event that we can recognize because the water in the pot has all evaporated.
- *Polling* refers to the action of taking a look at something every so often.
- An *event* is that something that we are expecting to happen.
- When we do this with the rice, if we do not check the rice at the right time, it might get burnt.

TOGGLE

- Let us try to read a button input.
- You will have to create a new program.
- In your new program, you will have to configure a pin as an input. Also, connect a pushbutton to that pin.
- You will also have to configure another pin as an output and connect an LED (with a resistor) to it.
- Your task is to toggle the LED each time you press the pushbutton.

TOGGLE

- To make things easier, let us review what the XOR bitwise operator does.
- The XOR operator outputs 1 if a single input (any of the two) is 1, but outputs 0 if both inputs are 1 or 0. In other words, outputs 1 when their inputs are different, and 0 when they are equal.
- If we apply an XOR operation to a byte with a mask, we will toggle the bits selected by the mask in the byte we are operating. Let us see some examples.

TOGGLE

- Check examples
  - I7266/mega32A/003_Toggle/001

INTERRUPTS

- Let us think of another analogy: what happens if you are doing your homework and then you feel the urge to go to the bathroom?
- In this case, the event is emptying your bladder, but this time you are not checking if your bladder is at a good level every so often. When nature calls, you have to run.
- This situation is the counterpart for *polling* an event, this is an event-driven routine, A.K.A. an *interrupt*.

INTERRUPTS

- There is another kind of interrupt. The one we just saw is an *internal interrupt*, since the event or the signal triggers the interrupt comes from inside our system (our body).
- Now think the same scenario of the rice, but this time imagine you say “alexa, set a timer for 12 minutes” when your rice starts cooking. This way we know we will have a signal coming from the exterior when the rice is done. You can think of some other cooking examples where this applies.

INTERRUPTS

- In these new examples, we are waiting for an *external interrupt*, since the signal that triggers the interrupt comes from the exterior.
- Internal interrupts are also called *software interrupts*.
- External interrupts are also called *hardware interrupts*.

- In short, interrupts make us stop whatever we are doing to switch to another task that needs our attention.

EXTERNAL INTERRUPTS

EXTERNAL INTERRUPTS

- Let us check the datasheed and look for information about external interrupts.

INTERNAL INTERRUPTS

NOTICES!

WhatsApp and classroom groups, missing data, class formality.

THANKS!

Please feel free to ask any related questions at any time.

C EMBEDDED PROGRAMMING

What is the difference?

EMBEDDED C

- It is a sort of variant of the C language. It keeps the rules, but it is also tailored in the way it is used to be more effective in embedded systems.
- Think of the constrains of embedded systems:
  - Fast response (real-time operation)
  - Limited storage space
  - Low possibilities of changing firmware after production
  - Among many others

FIRMWARE VS EMBEDDED SOFTWARE

- Although these may be concepts that are still evolving with the development of ever complex embedded technologies, my take on the question is the following:
  - Embedded software refers to any kind of software, (i.e., code), that runs in a micro-electronic system with a limited functional scope.
  - Firmware refers to the software in charge of the abstraction of hardware functionality as well as the basic behaviour of an embedded system.
- This is not the most important piece of information, but the topic leads to the question:
  - How complex and multi-purpose can an embedded system be?

EMBEDDED SYSTEMS SOFTWARE ARCHITECHTURE

SOFTWARE ARCHITECTURE

- With the improvements in integrated circuit manufacturing, microprocessors became microcontrollers, and microcontrollers became system-on-chips.
- As the hardware improves, the software has more room to grow too. This means that old programming structures or methods become obsolete.
- Let us use this time to explore how we will make a program