

Embedded Computing

M. Briday



year 2020/2021

- 1 Introduction
- 2 How to deal only with 0 and 1?
- 3 Specific C language operations
- 4 General Purpose I/O
- 5 Clock Sources
- 6 Pin muxing
- 7 Timer
- 8 Pulse Width Modulation
- 9 Interrupts
- 10 External Interrupt Handling
- 11 Serial Comm. (UART/I2C/SPI)

1 Introduction

2 How to deal only with 0 and 1?

3 Specific C language operations

4 General Purpose I/O

5 Clock Sources

6 Pin muxing

7 Timer

8 Pulse Width Modulation

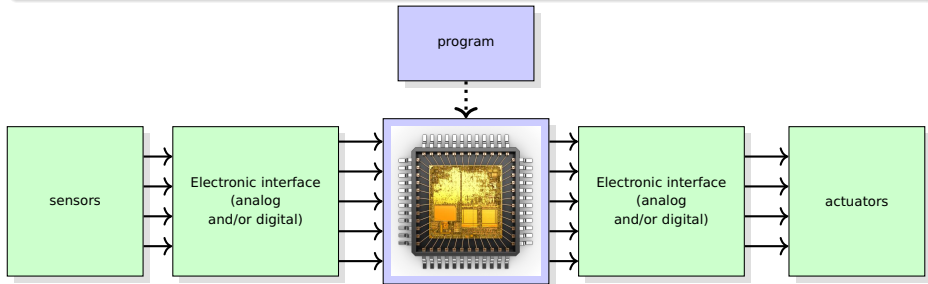
9 Interrupts

10 External Interrupt Handling

11 Serial Comm. (UART/I2C/SPI)

Positioning

At The Frontier Between Hardware and Software



A micro-controller is an integrated circuit that has:

- ») one (or more) *calculation unit(s)*;
- ») some *memory* (to store a program and data);
- ») some *internal peripherals* (to access the hardware).

The objective of this course is to learn:

- ») the software environment for deeply embedded systems
- ») the basic hardware peripherals of a micro-controller
- ») the design of a bare metal application

Main Features

- » *8/16 or 32 bits architecture*. This is the size of the data handled by the processor. The time required to add 2 numbers on 32 bits will be much longer on a 8-bits processor. . . ;
- » the *pin number* available on the chip;
- » *frequency*: a micro-controller is a synchronous system, with a clock. The faster the clock, the faster the calculations are;
- » *power consumption* is a key criterion for battery-powered systems;
- » *peripherals* implemented (I/O, Analog inputs, communication, . . .);
- » *cost* is a criterion for large series.

The micro-controller market

In 2019:

- » In value, market of 16.5 billion dollars (17.6 in 2018)
- » In volume, 26.9 billion microcontrollers (28.1 in 2018)

Trends:

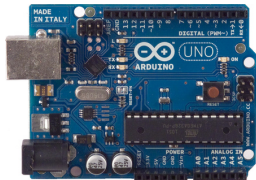
- » *volume growth* of 3.9%/year between 2018 and 2023;
- » *value growth* of 6.3%/year over the same period
- » more 32 bits micro-controllers than 4/8bits since 2015.

39% for the automotive industry. Growth in the IoT.

source: VIPress.net - 2019/8/23

Examples

Arduino Uno: AVR micro-controller ATmega328p from Atmel/Microchip:



-)) 8 bits
 -)) 32 KB flash (program)
 -)) 2 KB SRAM (data)
-)) 16 MHz
-)) 5V
-)) 28 pins

Nucleo 32: ARM Cortex-M4 based micro-controller STM32F303 from ST:



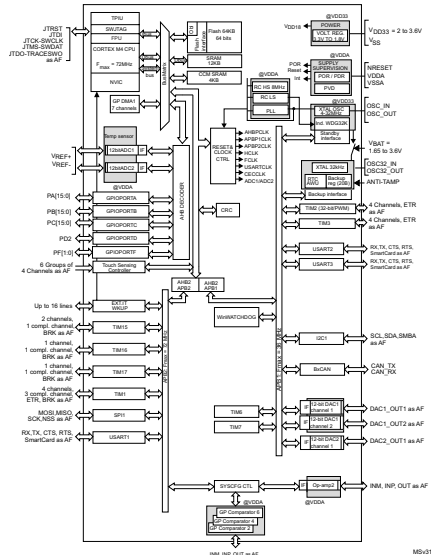
-)) 32 bits
 -)) 32 to 512 KB flash (program)
 -)) 16 to 80 KB SRAM (data)
-)) 72 MHz
-)) 3.3V
-)) 32 to 144 pins

This board is used in this course.

Block Diagram

This is a quite small micro-controller. . . But it

embeds many peripherals!



Block Diagram

This is a quite small
micro-controller. . . But it

embeds many peripherals!

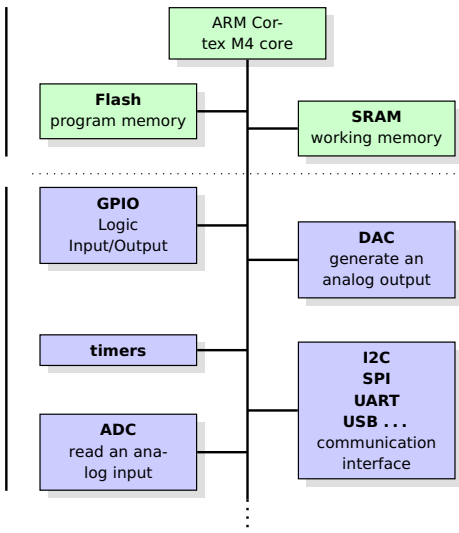
This commercial diagram
seems more readable. . .



Simplified Architecture

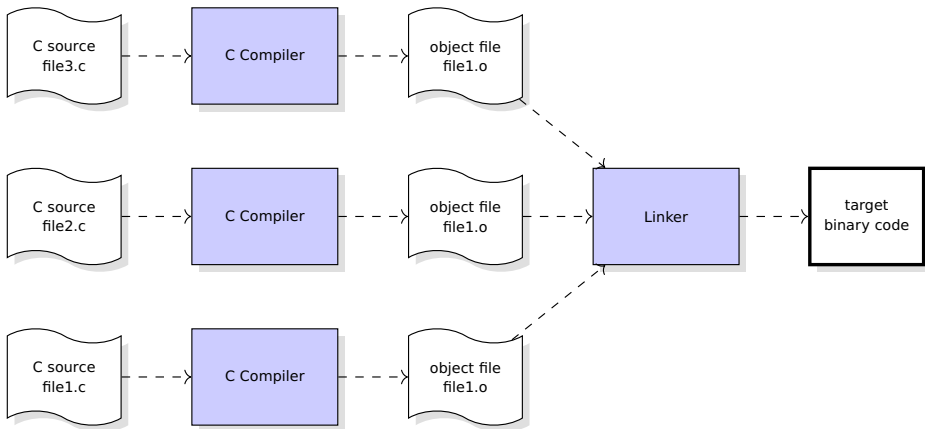
Memory + core

Peripherals



The core

Compilation chain: getting the binary code from the C source code.



The binary code is in fact a series of elementary instructions ordered in machine language. One elementary instructions can do:

- ») an *arithmetic or logical operation* between 2 data (addition, subtraction, AND, OR, ...);
- ») a *memory transfert*;
- ») a *branch* (to another program location);
- ») some specific operations (sleep, mode of operation, ...).

How a core works

So the core:

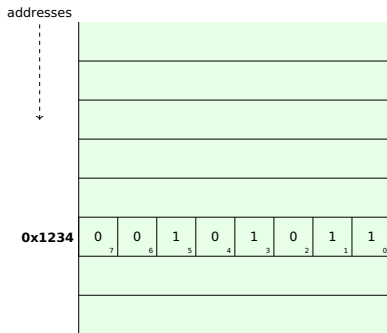
- ») runs a program:
 - ») performs calculations;
 - ») updates memory;
 - ») ... not much less!
- ») but also *interacts* with the rest of the system!

Peripheral access

The easiest way is to use memory access (read/write) to interact with the system.

we will see later on another way to interact with the core, with interrupts page 1.

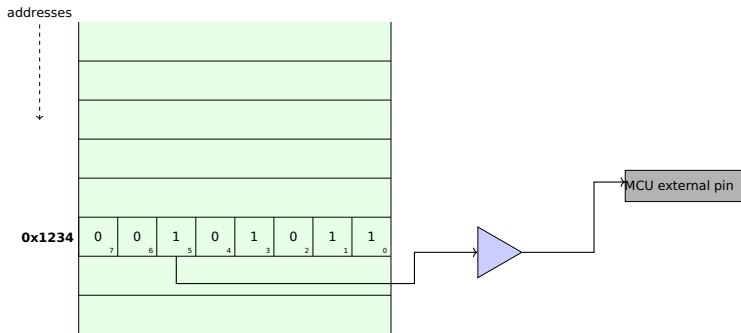
Peripheral registers...



Basic Principle

The status of each bit in a register (0 or 1) is used as input information for a device.

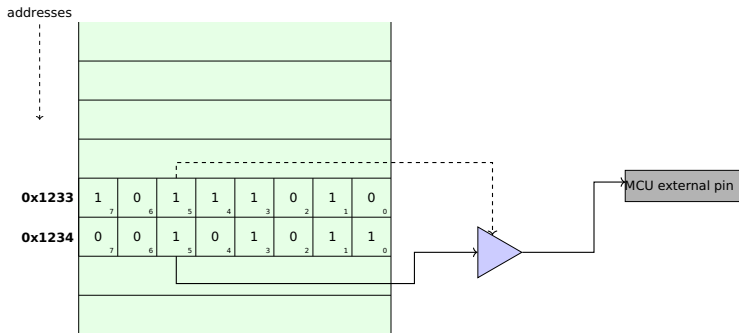
Peripheral registers...



Basic Principle

The status of each bit in a register (0 or 1) is used as input information for a device.

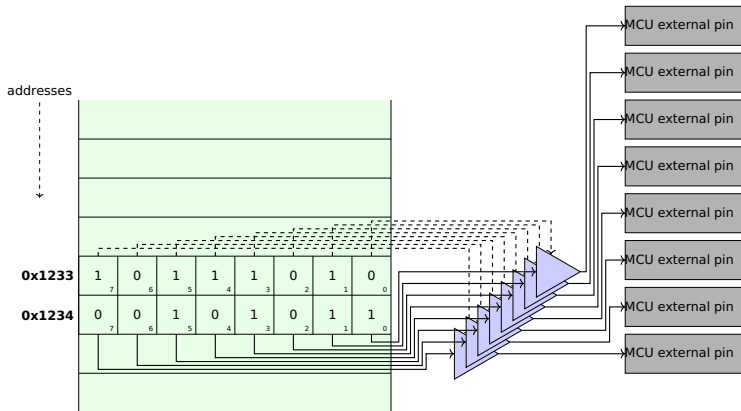
Peripheral registers...



Basic Principle

The status of each bit in a register (0 or 1) is used as input information for a device.

Peripheral registers...



Basic Principle

The status of each bit in a register (0 or 1) is used as input information for a device.

Peripheral registers...

A register makes the link between the *computer part* (memory access of the program) and the *electronic part* (command of a peripheral).

The device is a combinatorial or sequential system. There are 3 types of registers:

control register: access allows you to configure the device (write);

status register: access allows to read the device status (read);

data register: allows data to be exchanged with the peripheral (read/write).

In most cases, many registers are required to interact with a peripheral.

Contents

1 Introduction

2 How to deal only with 0 and 1?

3 Specific C language operations

4 General Purpose I/O

5 Clock Sources

6 Pin muxing

7 Timer

8 Pulse Width Modulation

9 Interrupts

10 External Interrupt Handling

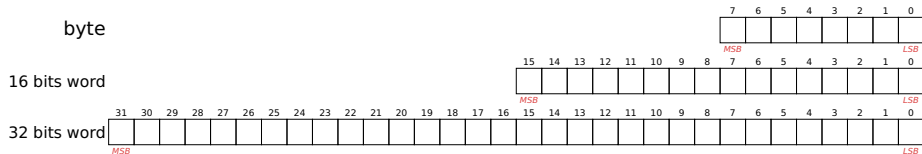
11 Serial Comm. (UART/I2C/SPI)

Data handled

Bit (*B*inary *d*igit), boolean data (0, 1);

Byte 8 bits;

Word 16 or 32 bits! Depends on the cpu!



LSB (*Least Significant Bit*)

MSB (*Most Significant Bit*)

The bits are numbered from the LSB (bit 0 \Leftrightarrow LSB)

Counting bytes...

Historically, bytes are counted in multiples of $2^{10} = 1024$, but the metric system counts in multiple of $10^3 = 1000$.
2 units are defined, in decimal and binary:

Name	Symbol	Value
kilo	ko	10^3
mega	Mo	10^6
giga	Go	10^9
tera	To	10^{12}
peta	Po	10^{15}

Name	Symbol	Value
kibi	kio	2^{10}
mebi	Mio	2^{20}
gibi	Gio	2^{30}
tebi	Tio	2^{40}
pebi	Pio	2^{50}

This has been validated in the ISO norm in 1998, ... but not always respected.

Memory organisation

On every modern processor, we can consider the memory organisation as just a flat tabular of 2^p bytes.

Processor	p	address space
Microchip PIC18	12	4096
Microchip ATmega328	16	64 kio
ARM Cortex-M	32	4 gio

Practically, it is often possible to read/write several bytes (2/4) to accelerate memory transfers.

The access in that last case should be *aligned*: for instance, a 32-bits access should have a memory address that is a multiple of 4.

Unsigned integers coding

The *hexadecimal* base is very often used in embedded systems.

The conversion from binary \Leftrightarrow hex basis is straightforward: an hex number is a group of 4 binary digits.

Numbers in hex will be prefixed by 0x, as in C language.

Example:

```
int val1 = 0x1234; //hexa
int val2 = 4660;   //decimal
```

Here, val1 and val2 will have the same value in memory.

0001 0010 0011 0100

The compiler performs the base change during code generation.

decimal value	Hexadecimal	Binary
0	0x0	0 0000
1	0x1	0 0001
2	0x2	0 0010
3	0x3	0 0011
4	0x4	0 0100
5	0x5	0 0101
6	0x6	0 0110
7	0x7	0 0111
8	0x8	0 1000
9	0x9	0 1001
10	0xa	0 1010
11	0xb	0 1011
12	0xc	0 1100
13	0xd	0 1101
14	0xe	0 1110
15	0xf	0 1111
16	0x10	1 0000
17	0x11	1 0001
18	0x12	1 0010
19	0x13	1 0011

As a consequence, in *hexadecimal*, a byte requires 2 digits:

0x00 \Rightarrow 0

...

0xFF \Rightarrow 255

A value is coded on the interval $[0, 2^8 - 1] = [0, 255]$

How to represent a negative value?

We can only deal with 2 symbols: 0 and 1...

Signed numbers - a first naive approach

we split the sign and the absolute value:

- ») one bit (MSB) codes the sign (0 is +, 1 is -)
- ») 2^{n-1} bits for the absolute value

The interval is $[-(2^{n-1} - 1), 2^{n-1} - 1] \Rightarrow [-127, 127]$

1st problem: *Zero is coded twice:*

- ») $+0 = 0000\ 0000$
- ») $-0 = 1000\ 0000$

How to do the code `if(val==0) ...?`

Warning

This representation is not used in real processors

Signed numbers - a first naive approach

The electronic circuit that performs the addition should be updated:

	7	6	5	4	3	2	1	0
30 (0x1E)	0	0	0	1	1	1	1	0
+								
-12 (-0xC)	1	0	0	0	1	1	0	0
.....								
	7	6	5	4	3	2	1	0

Warning

This representation is not used in real processors

Signed numbers - a first naive approach

The electronic circuit that performs the addition should be updated:

	7	6	5	4	3	2	1	0
30 (0x1E)	0	0	0	1	1	1	1	0
+								
-12 (-0xC)	1	0	0	0	1	1	0	0
.....								
	7	6	5	4	3	2	1	0
30+(-12) = 18 \neq 0xAA = -42	1	0	1	0	1	0	1	0

Warning

This representation is not used in real processors

Signed integers - 2's Complement

Signed integers are coded using the 2's complement: 2^n .

With 8 bits, -12 is coded $2^8 - 12 = 244$, in binary 1111 0100

As a consequence:

» (-0) is coded $2^8 - 0 = 256$, but using 8 bits \Rightarrow 0000 0000

unicity of 0

» The interval is *no more symetric*: $[-2^{n-1}, 2^{n-1} - 1] \Rightarrow [-128, 127]$

Signed integers - 2's Complement

The electronic circuit that performs the addition is the same:

	7	6	5	4	3	2	1	0
30 (0x1E)	0	0	0	1	1	1	1	0
+								
	7	6	5	4	3	2	1	0
-12 (0xF4)	1	1	1	1	0	1	0	0
.....								
	7	6	5	4	3	2	1	0

Signed integers - 2's Complement

The electronic circuit that performs the addition is the same:

	7	6	5	4	3	2	1	0
30 (0x1E)	0	0	0	1	1	1	1	0
+								
-12 (0xF4)	1	1	1	1	0	1	0	0
.....								
30+(-12) = 18 = 0x12	0	0	0	1	0	0	1	0

As a side effect, the MSB gives the sign.

Coding Intervals

Coding intervals for n bits are:

- » $[0, 2^n - 1]$ with unsigned int
- » $[-2^{n-1}, 2^{n-1} - 1]$ with signed int

data size	unsigned	signed
8 bits	[0,255]	[-128,127]
16 bits	[0,65 535]	[-32 768,32 767]
32 bits	[0,4 294 967 295]	[-2 147 483 648,2 147 483 647]
64 bits	[0,18 446 744 073 709 551 616]	[-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]

On $[0, 2^{n-1} - 1]$, the coding of both unsigned and signed integers is the same.

Integers in C/C++

The C type **int** is the basic data that is handled by the CPU. its size must be at least 16 bits.

- 》 the AVR architecture (8-bits CPU), one **int** is *16 bits*.
- 》 the ARM architecture (32-bits CPU), one **int** is *32 bits*.

Classical types are:

type	common size	norm
char	8 bits	≥ 8 bits
short	16 bits	≥ 16 bits
int	32 bits	≥ 16 bits
long	32 bits	≥ 32 bits
long long	64 bits	≥ 64 bits

Warning

An **int** may be signed or unsigned!! To be sure, one can write **unsigned int**.

Integers in C/C++

To be sure of the size of the manipulated data, we can use (with the header file **#include** <stdint.h>):

data size	unsigned	signed
8 bits	uint8_t	sint8_t
16 bits	uint16_t	sint16_t
32 bits	uint32_t	sint32_t
64 bits	uint64_t	sint64_t

Overflow with C/C++

Warning

With a 16/32 bits CPU, overflows can easily occur!
They are not handled by the C (operation using modulo 2^n).

What is the result of the following code ?

```
uint8_t val = 400;  
  
sint8_t t = 100;  
sint8_t u = 50;  
sint8_t v = t+u;  ///
```

```
uint8_t nbItem = 100;  
while(nbItem >= 0)  
{  
    //user code  
    nbItem--;  
}  
  
//or  
  
for(uint8_t i=0;i<256;i++){  
    //user code  
}
```

Byte order in memory - Endianness

For data greater than 1 byte, 2 solutions are available.
Let the 32-bits value 305419896 (or 0x12345678):

Big Endian

The highest significant byte is at *the highest address*

addresses	
↓	
0x1234	12
0x1235	34
0x1236	56
0x1237	78

Little Endian

The highest significant byte is at *the lowest address*

addresses	
↓	
0x1234	78
0x1235	56
0x1236	34
0x1237	12

Byte order in memory - Endianness

Impact:

- ») communication between 2 systems \Rightarrow the endianness should be defined for the network: *Network Byte Order* of the IP protocol IP for instance.
- ») memory dump

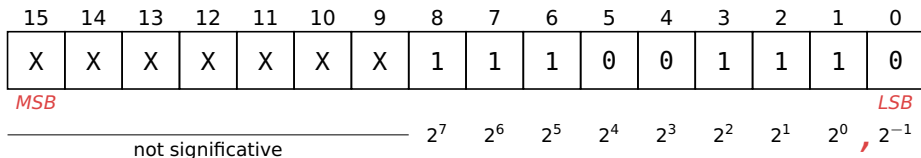
As there are 2 possibilities, founders didn't made the same choice!

- ») *Intel* (x86) for Little Endian
- ») Motorola (PowerPC) for Big Endian (also Alpha, Sparc, Mips, . . .)
- ») *ARM* cores can workk with the 2 approaches.

Fixed point numbers

We use the same hardware as for integers. Results should be interpreted.

Example: temperature sensor DS1620 (Maxim), the value is coded in 0.5°C increment, using 9 bits (2's complement):



Here, the value is -25°C.

Floating point numbers

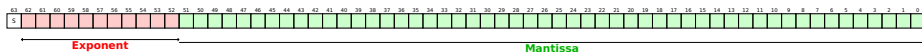
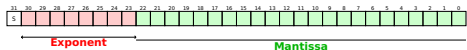
In that case, there is a trade-off between range and precision. The position of the point is not fixed. The standard *IEEE754* defines the number:

$$val = (-1)^S \times 1, M \times 2^E$$

where:

- » S is the *sign* (0 \Rightarrow positive, 1 \Rightarrow negative)
- » M is the fractionnal part of the *mantissa*, 23 or 52 bits;
- » E is the *exponent* (coded en with a bias of 127 with 32 bits, and 1023 with 64 bits), 8 or 11 bits;

the number is coded using 32 bits (**float**), or 64 bits (**double**):



Floating point numbers - addition steps

$$\begin{array}{rcll} & 1 & , & 0 & 1 & 1 & & \times 2^4 & & (10110)_2 \text{ or } 22_{10} \\ + & 1 & , & 0 & 0 & 1 & & \times 2^2 & & (100,1)_2 \text{ or } 4,5_{10} \end{array}$$

Étapes

- » Decoding values
- » Adapt to the same exponent

Floating point numbers - addition steps

$$\begin{array}{rcll} & 1 & 0 & 1 & , & 1 & & \times 2^2 & & (10110)_2 \text{ or } 22_{10} \\ + & 1 & , & 0 & 0 & 1 & & \times 2^2 & & (100,1)_2 \text{ or } 4,5_{10} \end{array}$$

Étapes

- » Decoding values
- » Adapt to the same exponent

Floating point numbers - addition steps

$$\begin{array}{r} 1\ 0\ 1\ ,\ 1\ x2^2 \\ + \quad 1\ ,\ 0\ 0\ 1\ x2^2 \\ \hline \end{array} \quad \begin{array}{l} (10110)_2 \text{ or } 22_{10} \\ (100,1)_2 \text{ or } 4,5_{10} \end{array}$$

Étapes

- » Decoding values
- » Adapt to the same exponent
- » Perform operation

Floating point numbers - addition steps

$$\begin{array}{r} 1\ 0\ 1\ ,\ 1\ x2^2 \\ + \quad \quad 1\ ,\ 0\ 0\ 1\ x2^2 \\ \hline 1\ 1\ 0\ ,\ 1\ 0\ 1\ x2^2 \end{array}$$

$(10110)_2$ or 22_{10}

$(100, 1)_2$ or $4, 5_{10}$

$(11010, 1)_2$ or $26, 5_{10}$

Étapes

- » Decoding values
- » Adapt to the same exponent
- » Perform operation
- » Normalize the result

Floating point numbers - addition steps

$$\begin{array}{r} 1\ 0\ 1\ ,\ 1\ x2^2 \\ + \quad\quad 1\ ,\ 0\ 0\ 1\ x2^2 \\ \hline 1\ ,\ 1\ 0\ 1\ 0\ 1\ x2^4 \end{array}$$

$(10110)_2$ or 22_{10}

$(100, 1)_2$ or $4, 5_{10}$

$(11010, 1)_2$ or $26, 5_{10}$

Étapes

- » Decoding values
- » Adapt to the same exponent
- » Perform operation
- » Normalize the result
- » Encode result

Floating point numbers - in an MCU

These steps are not straightforward:

- ») Hardware solution: *Floating Point Unit*
 - ») operation is fast (few MCU cycles)
 - ») better energy efficient
 - ») require some surface on the silicon (cost)
- ») Software: use a software lib (included in libc)
 - ») slow operation. . .
 - ») library requires a lot of memory (flash);
 - ») do we always need **float** in an MCU?

Floating point numbers - in an MCU

These steps are not straightforward:

- » Hardware solution: *Floating Point Unit*
 - » operation is fast (few MCU cycles)
 - » better energy efficient
 - » require some surface on the silicon (cost)
- » Software: use a software lib (included in libc)
 - » slow operation. . .
 - » library requires a lot of memory (flash);
 - » do we always need **float** in an MCU?

Tests on a MCU Teensy 3.1 (no FPU)

For 1 million of add operations (20+6):

- » 31,3 ms using **int**;
- » 700,3 ms using **float** (×22);
- » 1618,7 ms using **double** (×51);

ASCII code for characters

American Standard Code for Information Interchange

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	w	X	Y	Z	[\]	^	_
6x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Standard using 7 bits.

Coding a character in C

The type **char** uses 1 byte.

Warning

Depending on the compiler, it may be signed or not

char \neq **signed char** \neq **unsigned char**

A simple character may be coded using simple `""`. The 3 statements are similars:

```
char c = 'A'; //c gets ASCII code of character A
```

```
char d = 0x41; //ASCII code using hex
```

```
char e = 65; //or decimal
```

The character `"\"` defines among others:

) `'\n'` \Rightarrow new line;

) `'\\'` \Rightarrow simple `"\"`;

) `'\0'` \Rightarrow NUL (end of string)

Coding a character string in C

The type **char** * is a pointer to a 1-byte integer, or a tabular (size unknown) of 1-byte integers.

This is the historical way to code strings. NUL is used to define the *end of string*.

A character string uses ". Example:

```
char *txt="Bonjour_!";
```

Warning

NUL is not explicitly defined but is present in memory:

	B	o	n	j	o	u	r		!	
in memory	42	6F	6E	6A	6F	75	72	20	21	0

The ASCII uses 7 bits, so there are 128 codes available for country specific codes

- » standard ISO-8859-1 for Western Europe (*accented characters*)
- » standard ISO-8859-7 for Greece

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3	SP	!	~	#	\$	%	&	'	()	*	+	=	-	/	?
4	0039	0051	0021	0023	0024	0025	0036	0028	0029	0029	002A	002B	002C	002D	002E	002F
5	8	1	2	3	4	5	6	7	8	9	:	<	=	>	:	
6	0039	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
7	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
9	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
10	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
11	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
12	0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
13	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
14	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
15																
16																
17																
18																
19																
20	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F
21	0	1	2	3	4	5	6	7	8	9	:	<	=	>	:	
22	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F
23	A	A	A	A	A	A	A	A	C	E	E	E	E	E	I	I
24	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F
25	B	N	N	U	U	U	U	U	U	U	U	U	U	U	Y	P
26	00A0	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
27	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b
28	00A0	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	00B0	00B1	00B2	00B3	00B4	00B5	00B6	00B7	00B8	00B9	00BA	00BB	00BC	00BD	00BE	00BF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3	SP	!	~	#	\$	%	&	'	(*	+	-	.	/		
4	00	0001	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	0020	0021	0022	0023
5	01	0101	0110	0111	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123
6	02	0201	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
7	03	0301	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319	0320	0321	0322	0323
8	04	0401	0410	0411	0412	0413	0414	0415	0416	0417	0418	0419	0420	0421	0422	0423
9	05	0501	0510	0511	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523
10	06	0601	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
11	07	0701	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719	0720	0721	0722	0723
12	08	0801	0810	0811	0812	0813	0814	0815	0816	0817	0818	0819	0820	0821	0822	0823
13	09	0901	0910	0911	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923
14	0A	0A01	0A10	0A11	0A12	0A13	0A14	0A15	0A16	0A17	0A18	0A19	0A20	0A21	0A22	0A23
15	0B	0B01	0B10	0B11	0B12	0B13	0B14	0B15	0B16	0B17	0B18	0B19	0B20	0B21	0B22	0B23
16	0C	0C01	0C10	0C11	0C12	0C13	0C14	0C15	0C16	0C17	0C18	0C19	0C20	0C21	0C22	0C23
17	0D	0D01	0D10	0D11	0D12	0D13	0D14	0D15	0D16	0D17	0D18	0D19	0D20	0D21	0D22	0D23
18	0E	0E01	0E10	0E11	0E12	0E13	0E14	0E15	0E16	0E17	0E18	0E19	0E20	0E21	0E22	0E23
19	0F	0F01	0F10	0F11	0F12	0F13	0F14	0F15	0F16	0F17	0F18	0F19	0F20	0F21	0F22	0F23
20	10	1001	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023
21	11	1101	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120	1121	1122	1123
22	12	1201	1210	1211	1212	1213	1214	1215	1216	1217	1218	1219	1220	1221	1222	1223
23	13	1301	1310	1311	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323
24	14	1401	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
25	15	1501	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519	1520	1521	1522	1523
26	16	1601	1610	1611	1612	1613	1614	1615	1616	1617	1618	1619	1620	1621	1622	1623
27	17	1701	1710	1711	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723
28	18	1														

- ») 128 codes are not sufficient. . .
- ») some editors use their own extension;
- ») it is impossible to guess whose code is used;
- ») how to code a text that mixes 2 languages?

The solution?

- ») 128 codes are not sufficient. . .
- ») some editors use their own extension;
- ») it is impossible to guess whose code is used;
- ») how to code a text that mixes 2 languages?

The solution? *Unicode!*

Developed by the *Unicode Consortium*, which defines a universal character set, i.e. it aims to code all human languages. Each character has a unique number (the code point) between 0x0 and 0x10FFFFFF. It also specifies for the properties of each code point:

- ») its general category (letter, marque, number, separator, command, punctuation, symbol) ;
- ») the lowercaser, uppercase, corresponding title case (for a letter) ;
- ») its value (for a number) ;
- ») ...

The Unicode Consortium publishes free access files containing this information. Links :

- ») Wikipedia : <http://en.wikipedia.org/wiki/Unicode>
- ») Consortium Unicode : <http://unicode.org/>
- ») Unicode Character Database : <http://www.unicode.org/ucd/>

Unicode - Memory coding

Data representation is defined by *UTF*: Unicode Transformation Format, with different flavors:

UTF-32 simply using a 32-bit data:

- ») large;
- ») sensitive to *endianness*: UTF-32BE, UTF-32LE

UTF-16 1 or 2 16-bits words:

- ») memory trade-off (preferred version in memory)
- ») most of codes only require 1 word;
- ») sensitive to *endianness*: UTF-16BE, UTF-16LE.

UTF-8 characters 0 to 127 only use 1 byte:

- ») memory trade-off (preferred for files);
- ») ASCII compatible;
- ») insensitive to *endianness*.

Unicode - Memory coding

Using *UTF-8*:

UTF-8 binary representation	Meaning
0xxxxxxx	1 byte \Rightarrow 7 bits
110xxxxx 10xxxxxx	2 bytes \Rightarrow 8 to 11 bits
1110xxxx 10xxxxxx 10xxxxxx	3 bytes \Rightarrow 12 to 16 bits
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4 bytes \Rightarrow 17 to 21 bits

Example:

```
string str="École";
```

	É		c	o	l	e	
in memory	c3	89	63	6f	6c	65	0

Contents

1 Introduction

2 How to deal only with 0 and 1?

3 Specific C language operations

4 General Purpose I/O

5 Clock Sources

6 Pin muxing

7 Timer

8 Pulse Width Modulation

9 Interrupts

10 External Interrupt Handling

11 Serial Comm. (UART/I2C/SPI)

Specific C language operations

The handling of registers in C language requires some additions on the C language to manage:

- ») bit-level manipulation operations;
- ») pointers. . .
- ») structured data
- ») *static* and *volatile* data

We will finally see the structure of an embedded program, which differs slightly from a program running with an OS.

NOTE

Some C basics blocks are given in the first section of this chapter. This is only a remainder, and NOT a C language course!

The type of a variable is explicitly defined in C:

```
uint16_t val; //val is a 16 bit unsigned value
```

```
uint16_t val2; //val2 is another variable
```

```
val = 12; //assignment
```

```
val += 5; //same as val = val+5
```

```
val ++; //same as val = val+1
```

```
val2 = val-1; // assignement of value val2
```

Note: each statement end with ;

```
if(condition) {  
    //code executed if the condition is true  
} else {  
    //code execute if the condition is false  
}
```

the **else** block is optional.

C - condition

Example:

```
if(val > 12) {  
    //code executed if the condition is true  
}  
else {  
    //code execute if the condition is false  
}
```

NOTE

In C, the = is used for an assignment.
To compare 2 numbers, you need to use ==

```
if(val == 12) {  
    //code executed if the condition is true  
}
```

C - loop (1)

```
while(condition)
{
    //code executed until condition get false
}
```

Often used when we do not know the number of loops.

Example

```
while(val < 10)
{
    //code executed until condition get false
}
```

C - loop (2)

```
for(initialization; condition; update) //separator is ';'
{
    //loop code
}
```

It works as:

```
initialization; //done once at startup
while(condition)
{
    //loop code
    update;    //done at end of loop
}
```

C - loop (2)

```
for(int i=0; i<10; i++){  
    {  
        //loop code done 10 times.  
        //with i from 0 to 9.  
    }  
}
```

C - Function

```
void setup()
{
    //code of the function loop
}

void loop()
{
    //code of the function loop
}

int main()
{
    while(1)
    {
        loop();
    }
}
```

Note

-)) the `main()` function is the entry point of a program
-)) we *can not* define a function inside another function
-)) *No code* outside of a function
-)) the **void/int** is the returned value of the function
-)) the compiler parses the input text sequentially: a function *should* be defined *before* being called.
-)) a variable may be declared outside of any function. It becomes global (*i.e. usable everywhere in the code*)

C - Function

```
//function definition with parameters.
int max(int a, int b)
{
    //local variable usable only
    //in the function
    int result;
    if(a > b) {
        result = a;
    } else {
        result = b;
    }
    return result; //value returned to the caller
}

int main()
{
    int a;
    a = max(12,34); //function call with parameters
}
```

C - Mixing all together

```
int main()                                //function def
{
    const int threshold = 100; //constant value
    int boundMin = 1000;       //variable declaration with initial value
    int boundMax = 0;
    while(1)                   //loop
    {
        int val = readSensor(); //call a function
        if(val > threshold)      //test
        {
            alert(val);
        }
        boundMin = min(val, boundMin); //call
        boundMax = max(val, boundMax); //call
    }
}
```

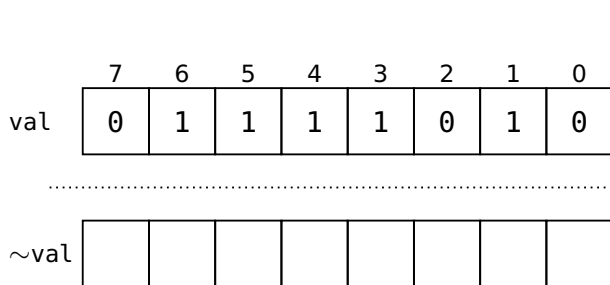
Note

- » a variable defined inside a block {} is defined only in this block.
- » indentation makes codes readable: Each time a { is started, the code is shifted right with some space.

C - bit-to-bit NO operation

The C *unary* operator \sim means: *bit-to-bit NO*:

bit	\sim bit
0	1
1	0



```
char val = 0x7A;  
val = ~val; //0x85
```

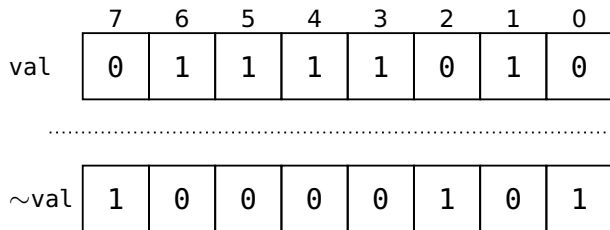
Usage

The operation \sim inverts each bit of a data.

C - bit-to-bit NO operation

The C *unary* operator \sim means: *bit-to-bit NO*:

bit	\sim bit
0	1
1	0



```
char val = 0x7A;  
val = ~val; //0x85
```

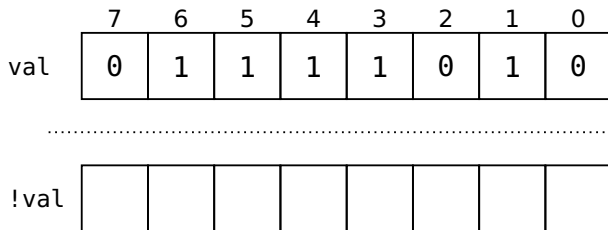
Usage

The operation \sim inverts each bit of a data.

C - Logical NO operation

The C *unary* operator ! means: *Boolean NO*:

```
char val = 0x7A;  
val = !val;  
// => val = 0  
val = !val;  
// => val != 0
```



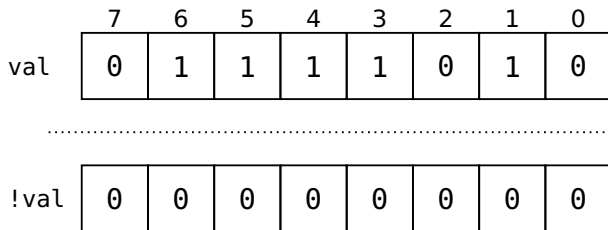
Usage

The operation ! acts on the *whole value* of the variable.

C - Logical NO operation

The C *unary* operator ! means: *Boolean NO*:

```
char val = 0x7A;  
val = !val;  
// => val = 0  
val = !val;  
// => val != 0
```



Usage

The operation ! acts on the *whole value* of the variable.

C - Complementing operations

Do not confuse:

») *bit-to-bit NO*: \sim

```
int x = ~val;
```

Each bit is complemented

») *Boolean NO*: $!$

```
int x = !val;
```

The Boolean meaning of the C language is:

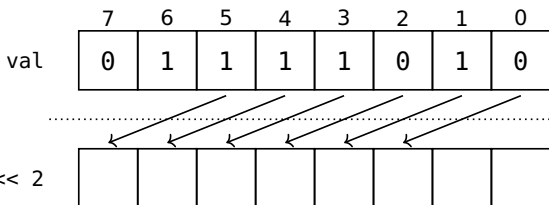
- ») 0 (FALSE in C) becomes a value different from 0 (not necessarily 1!);
- ») a value different from 0 (TRUE in C) becomes 0;

This value depends on the compiler!

C - Shift operator <<

The *binary* operator '<<' means: *left shift*:

```
char val = 0x7A;  
char val2 = val << 2;  
// => val2 = 0xE8;
```



The n bits shift to the left:

- » insert n 0 from the *Lowest Significant Bit* (LSB);
- » same operation as multiplying to 2^n .

Usage

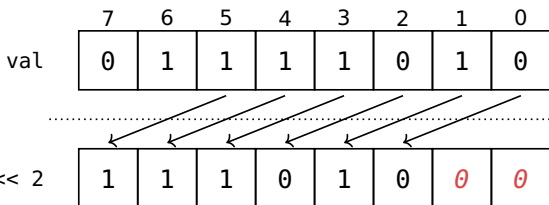
It is used to avoid calculating bit numbers:

```
int x = 1 << 5;    // => x= 100000 in binary
```


C - Shift operator <<

The *binary* operator '<<' means: *left shift*:

```
char val = 0x7A;  
char val2 = val << 2;  
// => val2 = 0xE8;
```



The n bits shift to the left:

- » insert n 0 from the *Lowest Significant Bit* (LSB);
- » same operation as multiplying to 2^n .

Usage

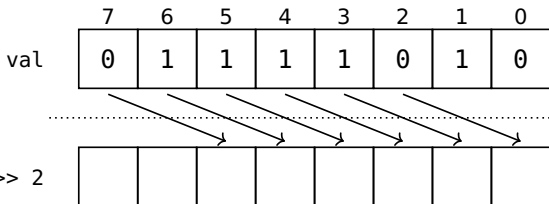
It is used to avoid calculating bit numbers:

```
int x = 1 << 5;    // => x= 100000 in binary
```

C - Shift operator >>

The *binary* operator '>>' means: *right shift*:

```
char val = 0x7A;  
char val2 = val >> 2;  
// => val2 = 0x1E;
```



The n bits shift to the right:

» same operation as dividing to 2^n .

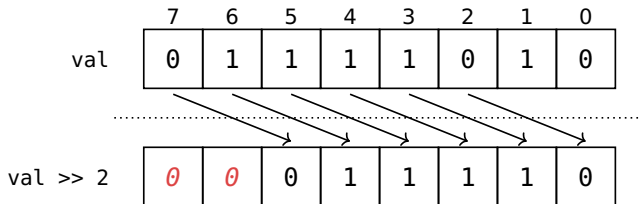
Warning!

When a variable is *signed*, if the *Most Significant Bit* (MSB) is 1 (negative value), the shift introduces n 1 (it keeps the sign).

C - Shift operator >>

The *binary* operator '>>' means: *right shift*:

```
char val = 0x7A;  
char val2 = val >> 2;  
// => val2 = 0x1E;
```



The n bits shift to the right:

» same operation as dividing to 2^n .

Warning!

When a variable is *signed*, if the *Most Significant Bit* (MSB) is 1 (negative value), the shift introduces n 1 (it keeps the sign).

C - OR operator

The *binary* | operator means: *bit-to-bit OR*:

bit_A	bit_B	bit_A or bit_B
0	0	0
0	1	1
1	0	1
1	1	1

	7	6	5	4	3	2	1	0
val	X	X	X	0/1	X	X	X	X

|

0x10	0	0	0	1	0	0	0	0
------	---	---	---	---	---	---	---	---

val | 0x10

--	--	--	--	--	--	--	--	--

Usage

The OR masking allows to *force* one or more bits to 1.

C - OR operator

The *binary* | operator means: *bit-to-bit OR*:

bit_A	bit_B	bit_A or bit_B
0	0	0
0	1	1
1	0	1
1	1	1



bit_A	bit_B	bit_A or bit_B
0	bit_B	bit_B
1	bit_B	1

	7	6	5	4	3	2	1	0
val	X	X	X	0/1	X	X	X	X

|

0x10	0	0	0	1	0	0	0	0
------	---	---	---	---	---	---	---	---

val | 0x10

--	--	--	--	--	--	--	--	--

Usage

The OR masking allows to *force* one or more bits to 1.

C - OR operator

The *binary* | operator means: *bit-to-bit OR*:

bit_A	bit_B	$bit_A \text{ or } bit_B$
0	0	0
0	1	1
1	0	1
1	1	1



bit_A	bit_B	$bit_A \text{ or } bit_B$
0	bit_B	bit_B
1	bit_B	1

	7	6	5	4	3	2	1	0
val	X	X	X	0/1	X	X	X	X

|

0x10	0	0	0	1	0	0	0	0
------	---	---	---	---	---	---	---	---

val | 0x10

X	X	X	1	X	X	X	X
---	---	---	---	---	---	---	---

Usage

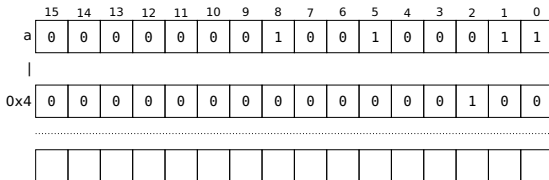
The OR masking allows to *force* one or more bits to 1.

C - OR mask : setting a bit

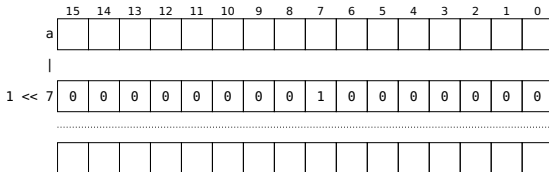
Example:

```
short a = 0x0123; //=> in binary: 0000 0001 0010 0011
```

```
a = a | 0x0004; //=> set bit 2
```



```
a = a | (1 << 7); //=> set bit 7
```

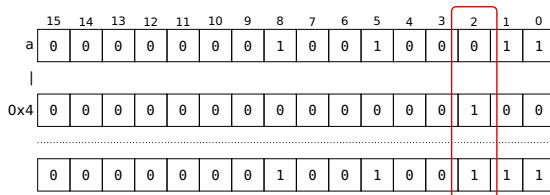


C - OR mask : setting a bit

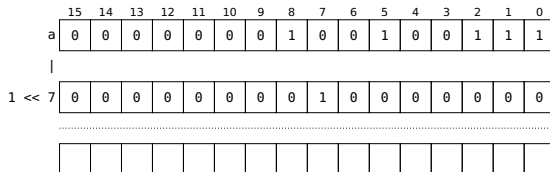
Example:

```
short a = 0x0123; //=> in binary: 0000 0001 0010 0011
```

```
a = a | 0x0004; //=> set bit 2
```



```
a = a | (1 << 7); //=> set bit 7
```

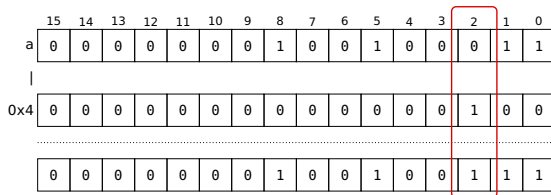


C - OR mask : setting a bit

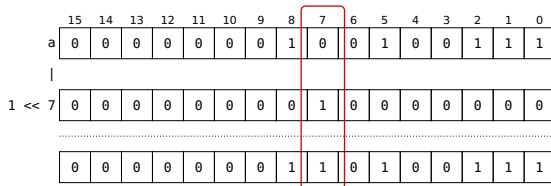
Example:

short a = 0x0123; *//=> in binary: 0000 0001 0010 0011*

a = a | 0x0004; *//=> set bit 2*



a = a | (1 << 7); *//=> set bit 7*



C - OR mask : setting a bit

shift and mask...

The use of shifting and masking operations will be *required*: registers are 32-bits wide!

Example:

```
a = a | (1<<31); //=> set bit 31
```

```
a |= (1<<31);      //same operation, shorter
```

```
//set many bits at the same time:
```

```
a |= (1 <<8) | (1<<4); //set bits 8 and 4.
```

C - AND operator

The *binary* & operator means: *bit-to-bit AND*:

bit_A	bit_B	$bit_A \text{ and } bit_B$
0	0	0
0	1	0
1	0	0
1	1	1

	7	6	5	4	3	2	1	0
val	X	X	X	0/1	X	X	X	X

&

0x10	0	0	0	1	0	0	0	0
------	---	---	---	---	---	---	---	---

val & 0x10

--	--	--	--	--	--	--	--	--

Usage

The AND masking allows to:

- » isolate one or more bits, resetting the others for a test;
- » reset 1 or more bits.

C - AND operator

The *binary* & operator means: *bit-to-bit AND*:

bit_A	bit_B	$bit_A \text{ and } bit_B$
0	0	0
0	1	0
1	0	0
1	1	1



bit_A	bit_B	$bit_A \text{ and } bit_B$
0	bit_B	0
1	bit_B	bit_B

	7	6	5	4	3	2	1	0
val	X	X	X	0/1	X	X	X	X

&

0x10	0	0	0	1	0	0	0	0
------	---	---	---	---	---	---	---	---

val & 0x10

--	--	--	--	--	--	--	--	--

Usage

The AND masking allows to:

- » isolate one or more bits, resetting the others for a test;
- » reset 1 or more bits.

C - AND operator

The *binary* & operator means: *bit-to-bit AND*:

bit_A	bit_B	bit_A and bit_B
0	0	0
0	1	0
1	0	0
1	1	1



bit_A	bit_B	bit_A and bit_B
0	bit_B	0
1	bit_B	bit_B

	7	6	5	4	3	2	1	0
val	X	X	X	0/1	X	X	X	X

&

0x10	0	0	0	1	0	0	0	0
------	---	---	---	---	---	---	---	---

val & 0x10

	7	6	5	4	3	2	1	0
	0	0	0	0/1	0	0	0	0

Usage

The AND masking allows to:

- » isolate one or more bits, resetting the others for a test;
- » reset 1 or more bits.

C - AND Mask: to test...

») Use for testing

```
//binary sensor associated to bit 4
int val=readSensor();
if(val & 0x10) {
    ...
}
```

The result of the operation (`val & 0x10`) is either:

`0x0` *FALSE* in C: bit 4 of `val` is *not* set;

`0x10` *TRUE* in C: bit 4 of `val` is set

This is compatible with the condition in the `if` statement.

C - AND Mask: to test...

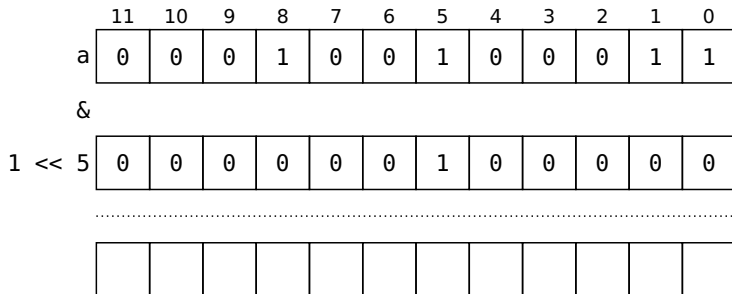
Example:

```
//=> in binary: 0001 0010 0011
```

```
int a = 0x123;
```

```
//=> test bit 5
```

```
if(a & (1<<5)) {...}
```



C - AND mask: ...and reset

» usage to reset a bit

Example: reset bit 4 of 32-bits variable val:

```
» //not so readable..  
val = val & 0xFFFFFEEF;
```

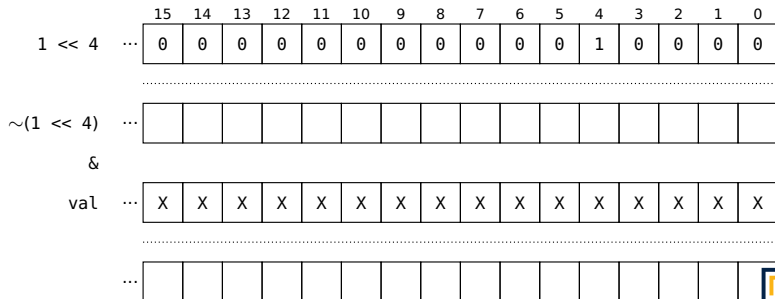

C - AND mask: ...and reset

» usage to reset a bit

Example: reset bit 4 of 32-bits variable val:

» `//not so readable..`
`val = val & 0xFFFFFEEF;`

» `//or simpler with the complementary operator:`
`val = val & ~(1 << 4);`



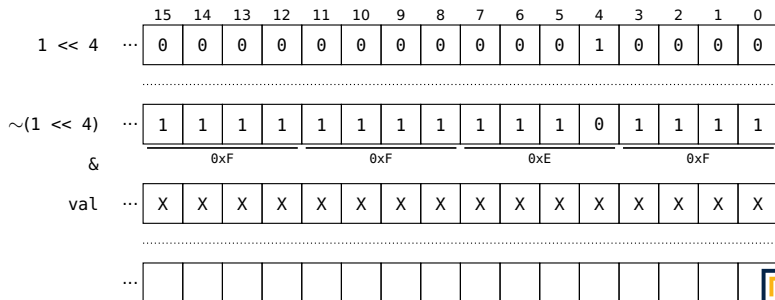
C - AND mask: ... and reset

» usage to reset a bit

Example: reset bit 4 of 32-bits variable val:

» `//not so readable..`
`val = val & 0xFFFFFEEF;`

» `//or simpler with the complementary operator:`
`val = val & ~(1 << 4);`



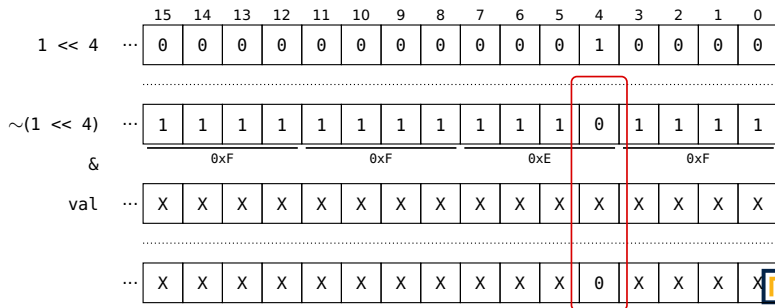
C - AND mask: ...and reset

» usage to reset a bit

Example: reset bit 4 of 32-bits variable val:

» `//not so readable..`
`val = val & 0xFFFFFEEF;`

» `//or simpler with the complementary operator:`
`val = val & ~(1 << 4);`



C - Mask operations - Summary

set a bit => OR mask '|'

```
// set bit 25 of a:  
a = a | (1 << 25);  
a |= (1 << 25);
```

reset a bit => AND mask '&' *and* complementation '~'

```
// reset bit 25 of a:  
a = a & ~(1 << 25);  
a &= ~(1 << 25);
```

test a bit=> AND mask '&'

```
// test bit 25  
if(a & (1 << 25)) {  
    .. //code executed if the bit is set  
}
```

Mask operations exercices

ex1

Give the value of val between each line:

```
uint16_t val = 0x4567;  
val = val | 0x1513;  
// val =>  
val = val & 0xFF22;  
// val =>
```

ex2

Give the value of val2:

```
uint16_t val2 = 0x74F0;  
val2 = val2 & ~(0xF << 8) | (0xA << 8) ;  
// val2 =>
```

ex3

val is an input value (unknown) of type `uint16_t` Write the code to set bits 4 and 5 of val:

val =

Write the code to reset bits 7 and 8 of val:

val =

Write the code to both set bits 4 and 5, and reset bit 2 and 3 of val

val =

C - Pointer basics

A data in memory holds 2 data:

the address of `val` is `0x34`;

the value of `val` is `0x4C` ($0100\ 1100$)₂

With C language:

The assignment of value `0x12` at address `0x34`

```
char val = 0x4C;
```

Definition

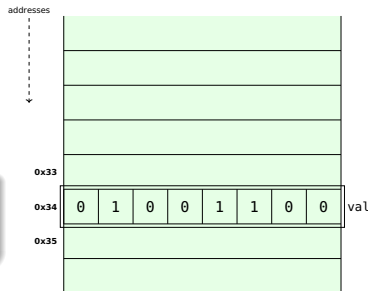
A pointer is a variable that contains the **address** of another variable

To define a variable `b` that store **the address** of `val`, we write:

```
int b = &val; // b=0x34
```

But, we have no information of the type (`char`, `int`, ...), only its address.

Note: we simplify here with variables/addresses on 8 bits... but the data are on 32 bits in reality!



C - Pointer basics (2)

Pointers allows to know the type of the data. It contains:

- ») the type of the data that is pointed;
- ») a * to show that it is a pointer;

A pointer stores an address, so *pointers have all the same size*:

int* x; //x is a pointer to an integer

char* x; //x is a pointer to a character

//x is a pointer to a 8-bits unsigned integer:

unsigned char* x;

C - pointer basics (3)

To remember

» & means "*the address of*": &a \Leftrightarrow address of variable a.

```
char* b; //b is a pointer to a char  
b = &val; // b=0x34
```

In this way, b is *a pointer to a char data*. And val type is char...

C - pointer basics (3)

To remember

- ») & means "*the address of*": &a \Leftrightarrow address of variable a.
- ») * allows to "*dereference un pointer*"

```
char* b; //b is a pointer to a char  
b = &val; // b=0x34
```

In this way, b is *a pointer to a char data*. And val type is char...

Dereference a pointer means *access to the pointed value*.

We can then do the manipulations:

```
*b = 0x12; //dereference a pointer
```

We write value 0x12 in the address pointed by b:

- ») b always contains the address of val;
- ») the value of val is modified.

C - arrays

An array is a contiguous list of elements of the same type. To define a tabular with 10 unsigned 8-bits integers:

```
unsigned char tab[10];
```

We can get a data in the array with its index. Here, `val` gets the value from data at address `0x36`.

```
unsigned char val = tab[2];
```

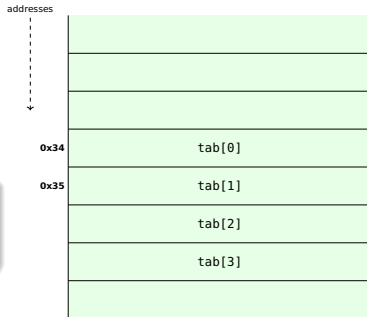
Pointers?

The name of an array is a pointer, which points to the first element of the array

As a consequence, type of `tab` is `unsigned char *`

and:

```
) tab[0] <=> *tab  
) tab <=> &tab[0]
```



C - Type definition

Custom types may be *defined* with the **typedef** keyword:

```
//definition of type 'byte'  
typedef unsigned char byte;
```

We enhance basic types. The definition of a *variable* respects the same syntax:

```
int a;  
byte b; //like an unsigned char  
...  
a = 0x1234;  
b = 12;
```

Note: Redefined types such as **uint32_t** are found in the standard C library to compensate the lack of portability of data sizes: **int** can be 16 or 32 bits depending on the architecture.

C - Structures (1)

The C language allows to define basic scalar types (**char**, **int**, ...) and homogeneous arrays.

It allows to *declare* structured types with the keyword **struct**:

```
typedef struct {  
    int data1;  
    char data2;  
    unsigned char data3;  
} newStruct;
```

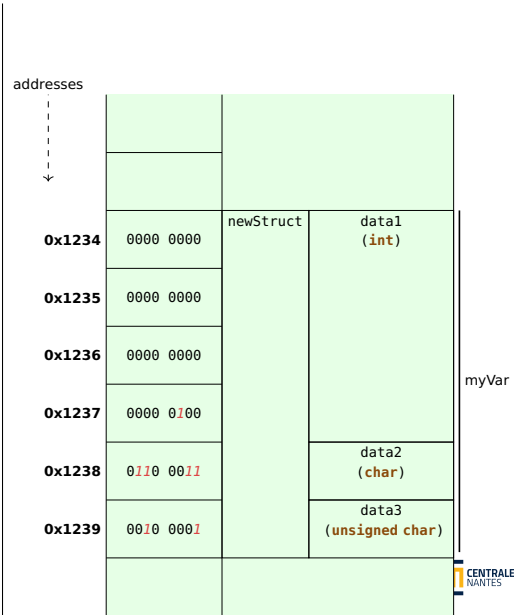
From this structure definition, we can instantiate a variable:

```
newStruct myVar;
```

C - Structures (2)

To access the different fields of the structure, we use the notation:

```
// type int
myVar.data1 = 4;
// type char
myVar.data2 = 'c'; // = 0x63
// type unsigned char
myVar.data3 = 33;
```



C - Structures (3)

To access a structure field, a pointer should be dereferenced:

```
(*GPIOA).MODER = ...
```

A simplified writing is available in C (completely equivalent):

```
GPIOA->MODER = ...
```

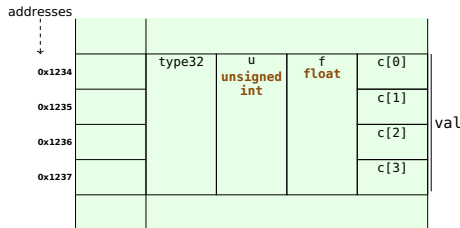
C - Unions

The *union* keyword in C language allows to use *the same memory location* with different forms.

```
typedef union
{
    unsigned int    u;
    float f;
    unsigned char c[4];
} type32;
```

a type32 variable needs 32 bits that may be interpreted in different ways:

- » an unsigned 32 bits value: type32.u
- » a 32 bits float value (norm ieee754p): type32.f
- » an array of 4 unsigned 8-bits values: type32.c[0]...



Example:

```
type32 val;
val.u = 0x12345678; //int
val.c[2] = 0xAA;
⇒ val = 0x1234AA78;
```


C - **static** modifier

Example:

```
int val1= 0; //global variable : accessible anywhere
```

```
void function1()
{
    val1++; //val1 = number of calls to the function
}
```

```
void function2()
{
    int val2 = 0; //local variable
    val2 ++; //val2 = 1
} //val2 est destroyed: access only IN function2()
```

```
void function3()
{
    //init only done the first time
    static int val3 = 0;
    val3 ++; //val3 = number of calls to the function
} //variable not destroyed! access only IN function3()
```

C - **volatile** modifier

During code generation, the compiler makes optimizations to speed up code execution, in particular:

- ») removing unnecessary code;
- ») instantiate variables directly into CPU general purpose registers for a faster access.

Behavior

The keyword **volatile** constrains the compiler to *effectively* perform the memory access.

Example:

```
volatile int i;  
for(i=0; i<1000; i++);
```

Without the **volatile** keyword, the compiler can remove the waiting loop... because it wastes time!

C - Structure of an embedded code

An embedded code *should never terminate*. In this way, the main function has in most cases the following form:

```
int main()
{
    setup(); //function run only once
    while(1) {
        .. //code executed inside a loop
    }
}
```

» function `setup()` init peripherals;

» code inside the loop `while(1)` is a never-ending process.

C - To go further. . .

Arduino uses a dialect of C/C++, named *wiring*. There are few differences between Wiring and C++.

There are many tutorials and guides for C / C++ / Wiring programming:

- » C tutorial: <http://www.zentut.com/c-tutorial/>
- » C++ tutorial: <http://www.cplusplus.com/reference/>
- » Arduino reference: <https://www.arduino.cc/reference/en/>

Contents

1 Introduction

2 How to deal only with 0 and 1?

3 Specific C language operations

4 General Purpose I/O

5 Clock Sources

6 Pin muxing

7 Timer

8 Pulse Width Modulation

9 Interrupts

10 External Interrupt Handling

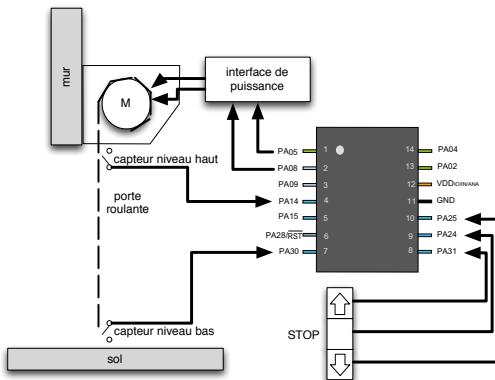
11 Serial Comm. (UART/I2C/SPI)

parallel ports

Objective

Parallel ports allow to control the pins of the microcontroller in *On-Off* mode

Basic example of a garage door:



- » 5 inputs:
 - » 2 high and low level limit switches;
 - » 3 push buttons (human / Machine interface).
- » 2 outputs:
 - » motor control (up, down, stop) through a power interface.

The access is *bi-directionnal* and can be configured as:

input (point of view of the μC) to *get* an information:

-) limit switches;
-) state of a push button.

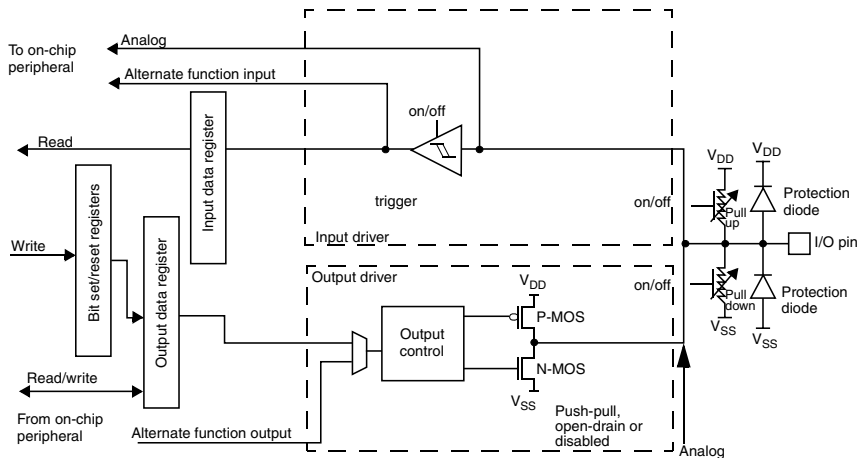
output to *control* an external peripheral:

-) command a LED;
-) digital device through several logical lines

It's a *parallel port* because it is possible to control several pins at the same time.

It is also defined as *General Purpose I/O*.

I/O architecture on the STM32F303

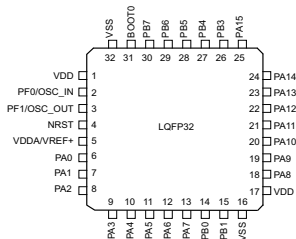


ai15938

I/O ports on STM32-F303-K8

Many ports are available (A to F on the STM32F303K8), but potentially many more:

- » This is a 32-bit μ C, but can only control up to 16 pins at the same time
ex: pin **PA15**: on/off pin 15 of port A.
- » Some pins may have no physical output.
ex: PB2 is not available
- » logical levels are:
 - logic 0** \Rightarrow 0V;
 - logic 1** \Rightarrow 3.3V (Warning, not TTL compatible!).



Each pin needs a configuration:

reset input floating

input with 3 configurations:

- ») input floating;
- ») input pull-up (a resistor that pulls the electric potential to VCC)
- ») input pull-down (a resistor that pulls the electric potential to GND)

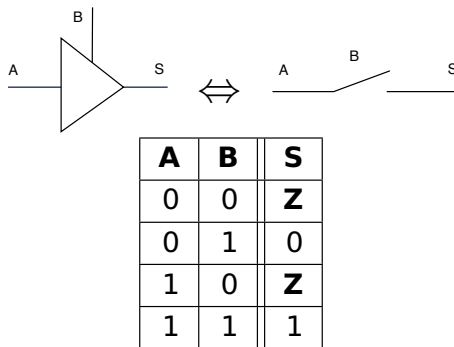
output with 2 configurations:

- ») push/pull
- ») open-drain

Each configuration is detailed hereafter.

3 states output

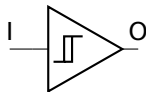
A 3-state output can be schematized as follows:



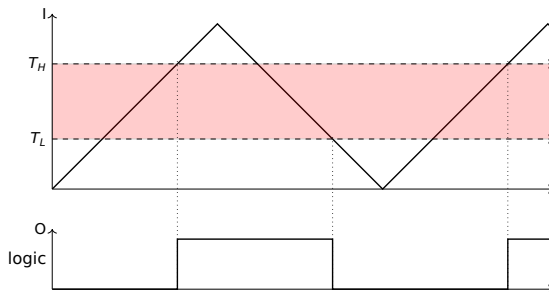
If B is not set ($B=0$), the output is *high impedance* (**Z**).

input with Schmitt trigger

a Schmitt trigger is a comparator circuit with hysteresis:



It is used to ensure the stability of a logic signal when the input (I) is located between the low (T_L) and high (T_H) thresholds, zone between 0 and 1 logic.



Pin configuration

Each pin of a GPIO port has independant configuration bits:

MODER *MODE* Register: input/output/alternate function. . .

OTYPER *Output TYPE* Register: push-pull or open drain

OSPEEDR *Output SPEED* Register

PUPDR *Pull UP / Down* Register: enable a pull resistor.

Data registers are:

IDR *Input Data* Register: get input value of the whole port

ODR *Output Data* Register: get output value of the whole port

BSRR *Bit Set Reset* Register: bit access to the port

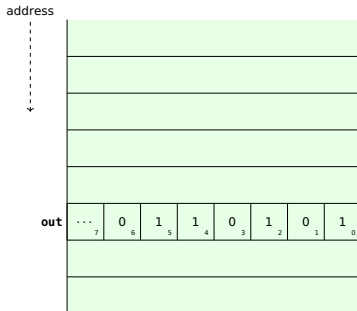
Each GPIO port is disabled at reset, and a clock source *should* be given to the port (see section 5).

The *RCC* (Reset and Clock Control) peripheral is used. GPIOs are connected to the AHB port:

```
RCC->AHBENR |= RCC_AHBENR_GPIOBEN_Msk; //clock for GPIOB
//wait until GPIOB clock is Ok.
__asm("nop");
```

You have to replace the ...GPIO*B*EN... symbol with the appropriate port.

Output access

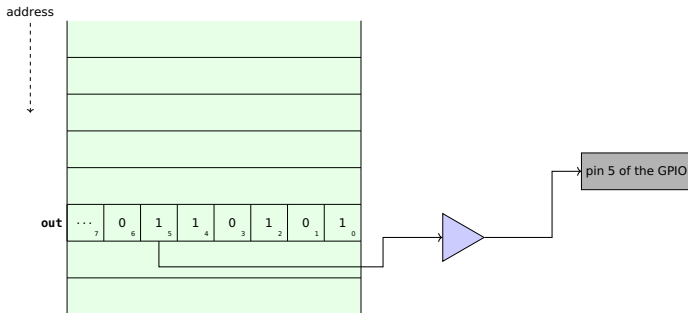


Up to **16** pins may be controlled simultaneously. . . ⇔ in parallel!

Warning

Mask operations will be **required** to avoid overwriting a previous configuration

Output access

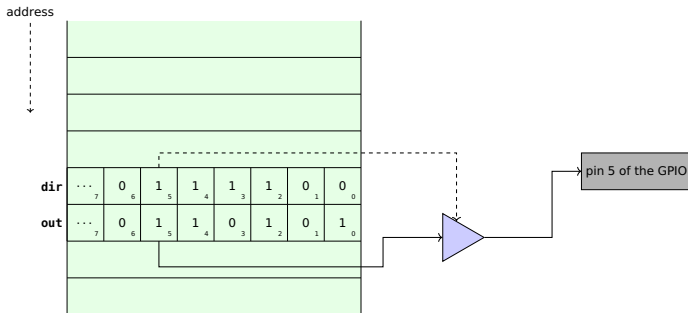


Up to **16** pins may be controlled simultaneously. . . \Leftrightarrow in parallel!

Warning

Mask operations will be **required** to avoid overwriting a previous configuration

Output access

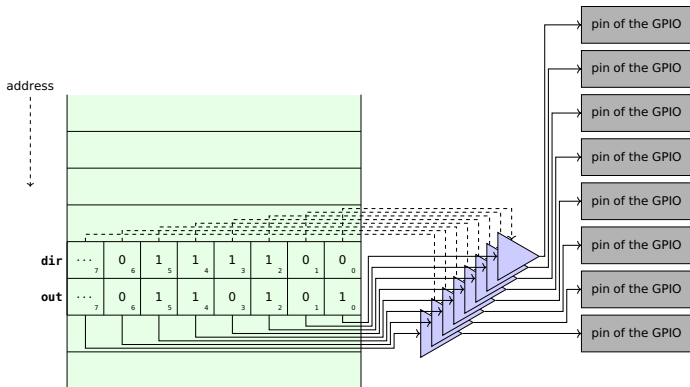


Up to **16** pins may be controlled in simultaneously. . . ⇔ in parallel!

Warning

Mask operations will be **required** to avoid overwriting a previous configuration

Output access



Up to **16** pins may be controlled in simultaneously. . . ⇔ in parallel!

Warning

Mask operations will be **required** to avoid overwriting a previous configuration

MODE Register

MODER (*MODE* Register) uses 2 bits to configure the mode for each pin:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

00 Input mode

01 output mode

10 alternate function mode (see section 6)

11 analog mode

Output type Register

OTYPER Output **TYPE** Register selects the push-pull or open drain output:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

0 push-pull

1 open-drain

Only the low 16 bits are used.

PUPDR Register

PUPDR Pull UP / Down Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

00 no pull-up, no pull-down

01 pull-up

10 pull-down

11 reserved

Output Data Register

ODR Output **Data** **Register** controls the output state of the pin:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

0 output is low

1 output is high

Only the low 16 bits are used.

Output configuration are:

MODER	OTYPER	PUPDR	state
01	0	00	output - push-pull
01	0	01	output - push-pull + pull-up
01	0	10	output - push-pull + pull-down
01	0	11	<i>reserved</i>
01	1	00	output - open-drain
01	1	01	output - open-drain + pull-up
01	1	10	output - open-drain + pull-down
01	1	11	<i>reserved</i>

Example: LED access

A led is available on the board, on PB3. This is a basic *push-pull* configuration.

Reset state of **MODER** is 0x0000.

```
//output configuration
GPIOB->MODER |= 1 << (3*2); //PB3 output
//or (better)
GPIOB->MODER |= 1 << GPIO_MODER_MODER3_Pos; //PB3 output
//or (even better)
GPIOB->MODER &= ~GPIO_MODER_MODER3_Msk; //reset PB3 mode
GPIOB->MODER |= 1 << GPIO_MODER_MODER3_Pos; //PB3 output
```

light the LED:

```
//output high
GPIOB->ODR |= 1 << 3;
```

turn off the LED:

```
//output low
GPIOB->ODR &= ~(1 << 3);
```


Selective access to a port

To prevent mask operations, register **BSRR** (**B**it **S**et/**R**eset **R**egister) allows to update a single bit (hardware mask operation)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

BS_x Bit Set

BR_x Bit Reset

Hardware mask

- » Writing a 1 performs the operation (set/reset)
- » Writing a 0 *does not update* the GPIO

Selective access to a port

Moreover, **BSRR** access is *atomic*: example:

```
//switch off LED  
GPIOB->ODR &= ~(1 << 3);
```

is translated into asm code:

```
ldr r3, [r1, #20]  
bic.w  r3, r3, #8  
str r3, [r1, #20]
```

What happens if there is an interrupt between the load and the store instructions?

Selective access to a port

Moreover, **BSRR** access is *atomic*: example:

```
//switch off LED  
GPIOB->BSRR = 1 << (3+16); //reset PB3
```

is translated into asm code:

```
mov.w r4, #524288 ; 0x80000  
...  
str r4, [r1, #24]
```

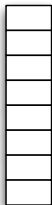
Atomic access

There is *no side effect* during the access. No need to protect the variable access.

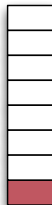
Exercice - Bargraph

8 LEDs are connected to the pins PB0 to PB7. The objective is to use LEDs to represent a value in the form of a bargraph:

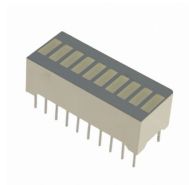
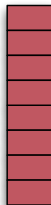
$val < max/8$



$max/8 \leq val < 2*max/8$



$val \geq max$



We consider that the value to be displayed $value$ is most of the time between 0 and a value max which is a parameter of the procedure.

Exercise - Bargraph

The routine is bargraph:

```
void bargraph(unsigned int value, unsigned int max);
```

Following the value of the parameter value, we have:

- ») if $value < \frac{max}{8}$ then no led is on;
- ») if $\frac{max}{8} \leq value < 2 \frac{max}{8}$, only the first led is on;
- ») if $2 \cdot \frac{max}{8} \leq value < 3 \frac{max}{8}$, the first 2 leds are on;
- ») ...
- ») if $value \geq max$, all the leds are on;

Note: You can implement the function:

- ») first listing all cases iteratively (list of **if**)
- ») then with a loop
- ») or directly, by identifying the number of leds to light

Correction - exercise Bargraph

```
void bargraph(unsigned int value, unsigned int max)
{
    //init
    PORTB->MODER |= 0x5555;
    unsigned int nbLed = (value*8)/max;
    if(nbLed>8) PORTB->BSRR = 0xFF;
    else {
        const unsigned int mask = (1 << nbLed) - 1;
        PORTB->BSRR = mask | (~mask & 0xFF) << 16;
    }
}
```

As seen in the GPIO internal structure (slide 86), the input driver:

- » inserts a Schmitt trigger before the logic part
- » is configured with **MODER** (config 00)
- » may use the push-pull resistors (**PUPDR**)

Input Data Register

IDR / Input **D**ata **R**egister returns the input state of the pin:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

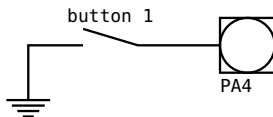
0 input is low

1 input is high

Only the low 16 bits are used.

Exercise - push button

We consider a basic push-button:



- ▶ What is the configuration of the pin PA4?
- ▶ How to read the button state?

Exercise - push button

If we don't press the button, the state is in *high impedance*. The pull-up resistor is mandatory!

```
void setup() {  
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN_Msk; //clock for GPIOA  
    __asm("nop"); //wait until GPIOA clock is Ok.  
    GPIOA->MODER &= ~GPIO_MODER_MODER4_Msk; //PA4 as input (0)  
    GPIOA->PUPDR &= ~GPIO_PUPDR_PUPDR4_Msk; //reset pupd for PA4  
    GPIOA->PUPDR |= 1 << GPIO_PUPDR_PUPDR4_Pos; //pull-up for PA4  
}
```

Exercise - GPIO driver

The objective of a driver is to hide the complexity of accessing configuration registers and to offer high-level functions. The 3 functions to manage inputs/outputs are inspired by the Arduino universe:

pinMode allows to configure a pin (input/output with pull up/down resistor)

digitalWrite sets an output pin state

digitalRead reads in anput pin state

5 modes are defined (in `pinAccess.h`):

```
#define DISABLE      0
#define OUTPUT       1 //only push/pull mode
#define INPUT        2
#define INPUT_PULLUP 3
#define INPUT_PULLDOWN 4
```

Exercise - pinMode

Give an implementation of the 3 functions:

```
int pinMode(GPIO_TypeDef *port,
            unsigned char numBit,
            unsigned char mode);

int digitalWrite(GPIO_TypeDef *port,
                unsigned char numBit,
                unsigned char value);

int digitalRead(GPIO_TypeDef *port,
               unsigned char numBit);
```

Where:

- port** refers to the hardware mapped structure: **GPIOA**, ...
- numBit** is the bit number: 0 to 15
- mode** is the defined mode (previous slide)
- value** is the output value (true/false as in C)

Warning

Particular attention should be paid to unexpected values (access to pin 54 for instance. . .).

Exercise - pinMode

```
int pinMode(GPIO_TypeDef *port,
            unsigned char numBit,
            unsigned char mode)
{

    //check arguments
    if(!IS_GPIO_ALL_INSTANCE(port)) return -1;
    if(numBit >  ) return -1;
    //

    switch(mode)
    {
        case DISABLE: //MODER = 0, PUPDR = 0

            break;
        case OUTPUT: //

            break;
        ...
    }
```

Correction - pinMode

```
int pinMode(GPIO_TypeDef *port,
            unsigned char numBit,
            unsigned char mode)
{
    int mask2Bits; //mask for 2bit fields
    //check arguments
    if(!IS_GPIO_ALL_INSTANCE(port)) return -1;
    if(numBit > 15) return -1;
    //
    mask2Bits = (3 << (numBit*2));
    switch(mode)
    {
        case DISABLE: //MODER = 0, PUPDR = 0
            port->MODER &= ~mask2Bits;
            port->PUPDR &= ~mask2Bits;
            break;
        case OUTPUT: //MODER = 1, PUPDR = 0
            clockForGpio(port);
            port->MODER &= ~mask2Bits;
            port->MODER |= (1<<(numBit*2));
            port->PUPDR &= ~mask2Bits;
            break;
        ...
    }
```

Exercice - digitalWrite

```
void digitalWrite(GPIO_TypeDef *port,
                  unsigned char numBit,
                  unsigned int value)
{
    if(!IS_GPIO_ALL_INSTANCE(port)) return;
    if(numBit >  ) return;

}
```

Correction - digitalWrite

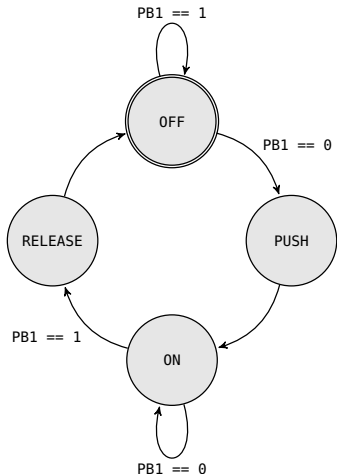
```
void digitalWrite(GPIO_TypeDef *port,
                  unsigned char numBit,
                  unsigned int value)
{
    if(!IS_GPIO_ALL_INSTANCE(port)) return;
    if(numBit > 15) return;

    if(value) port->BSRR = 1 << numBit;
    else      port->BSRR = 1 << (numBit+16);
}
```

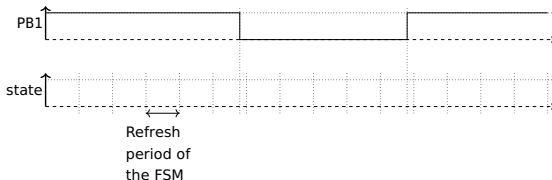

Exercise: Using an FSM

We use here a Finite State Machine (FSM) to get information about a push button:

Let the following FSM:



The FSM has 4 states, with 2 of them PUSH and RELEASE only for 1 cycle.



Exercise GPIO with FSM

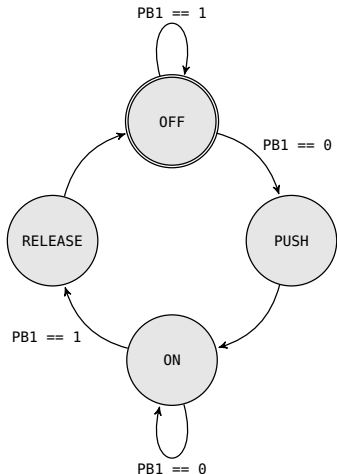
Write a program that toggles the state of the LED (PB0) each time the push button is pushed. The refresh frequency will be $\sim 100\text{Hz}$.

We consider here that there is a function delay (xx) to wait for xx ms.

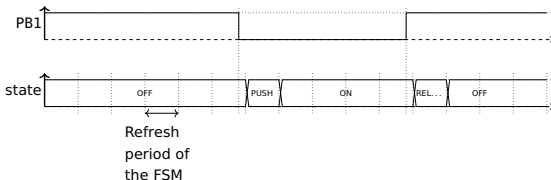
Exercise: Using an FSM

We use here a Finite State Machine (FSM) to get information about a push button:

Let the following FSM:



The FSM has 4 states, with 2 of them PUSH and RELEASE only for 1 cycle.



Exercise GPIO with FSM

Write a program that toggles the state of the LED (PB0) each time the push button is pushed. The refresh frequency will be $\sim 100\text{Hz}$.

We consider here that there is a function delay (xx) to wait for xx ms.

Exercise: Using an FSM

Implementation with a dedicated function:

```
enum PBState {OFF, PUSH, ON, RELEASE};
```

```
//return button state
```

```
enum PBState managePB0(){  
    static enum PBState state =  
    switch(state) {  
        case OFF:  
            break;  
        case PUSH:  
            break;  
        case ON:  
            break;  
        case RELEASE:  
            break;  
    }  
    return state;  
}
```

Exercise: Using an FSM

Implementation with a dedicated function:

```
enum PBState {OFF, PUSH, ON, RELEASE};

//return button state
enum PBState managePB0(){
    static enum PBState state = OFF;
    switch(state) {
        case OFF: if((GPIOB->IDR & 0x2) == 0) state = PUSH;
            break;
        case PUSH: state = ON;
            break;
        case ON: if(GPIOB->IDR & 0x2) state = RELEASE;
            break;
        case RELEASE: state = OFF;
            break;
    }
    return state;
}
```

Contents

1 Introduction

2 How to deal only with 0 and 1?

3 Specific C language operations

4 General Purpose I/O

5 Clock Sources

6 Pin muxing

7 Timer

8 Pulse Width Modulation

9 Interrupts

10 External Interrupt Handling

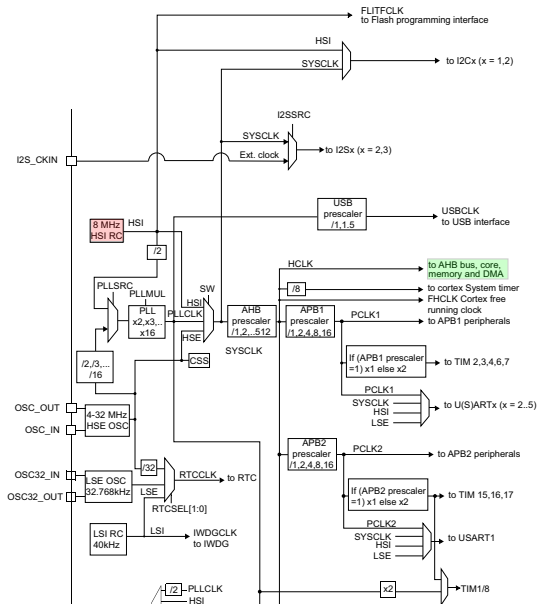
11 Serial Comm. (UART/I2C/SPI)

Devices are sequential systems that require a *clock source*.

The current trend of μ c founders is to limit consumption as much as possible. This requirement leads to certain technological choices:

- ») Reduce core consumption
 - ») fine management of the core's sleep modes;
 - ») peripheral functions without core use (*DMA, sleepWalking, ...*).
- ») reduce the consumption of devices
 - ») turn off the power to unused devices;
 - ») cut off the clock source;

Clock Tree of the STM32F303K8



Clocks should be configured at startup for

- ») the main core clock;
- ») devices clocks;

Let's have a look at the `SystemInit()` function.

1 Introduction

2 How to deal only with 0 and 1?

3 Specific C language operations

4 General Purpose I/O

5 Clock Sources

6 Pin muxing

7 Timer

8 Pulse Width Modulation

9 Interrupts

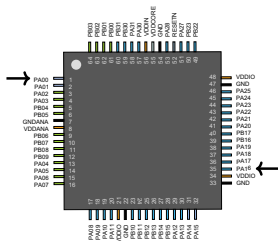
10 External Interrupt Handling

11 Serial Comm. (UART/I2C/SPI)

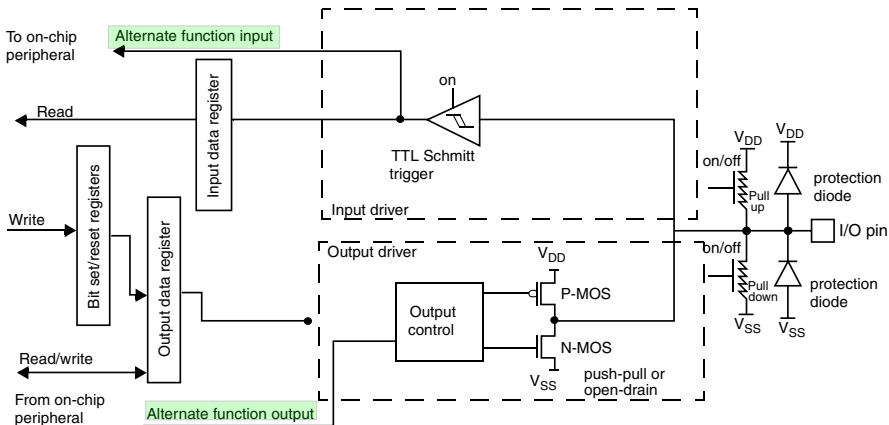
Principle

Some devices use pins of the microcontroller as input or output.

- » By default, a pin is used as a GPIO (digital input/output).
- » We should know which device will use the pin
- » Some devices may have pins that are driven to different physical pins (easier electronic schema)
 - » example: The first pin of the serial communication of this μ C (SAMD21J18A) can be affected to physical pins PA0 or PA16.



Hardware part...



...and software part

The founder allows up to 16 alternate configurations for each pin, called AF0 to AF15.

2 registers should be updated:

- » MODER register (slide 94), with configuration 10
- » AFRL (and AFRH) that gives the alternate configuration number, from 0 (AF0) to 15 (AF15), for respectively the lowest 8 pin numbers and the highest ones.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFR7[3:0]				AFR6[3:0]				AFR5[3:0]				AFR4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFR3[3:0]				AFR2[3:0]				AFR1[3:0]				AFR0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

1 Introduction

2 How to deal only with 0 and 1?

3 Specific C language operations

4 General Purpose I/O

5 Clock Sources

6 Pin muxing

7 Timer

8 Pulse Width Modulation

9 Interrupts

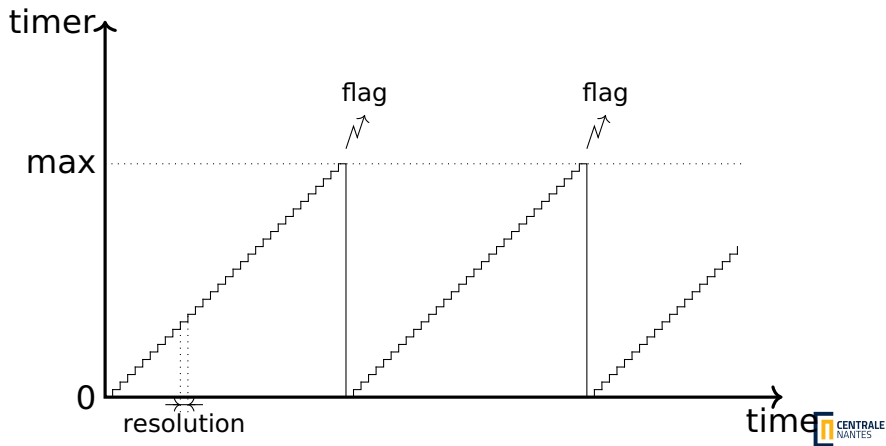
10 External Interrupt Handling

11 Serial Comm. (UART/I2C/SPI)

- » Provide a time base (*timer*):
 - » perform a precise standby function;
 - » generate a periodic behavior such as flashing an LED, sending a periodic message, ...
- » Counting events (*counter*):
 - » number of engine revolutions in a car
- » With some additional logic, create more advanced functions such as:
 - » timestamp with a capture input;
 - » with a comparator, a Pulse Width Modulation (*PWM*) signal (→ chapter p. 8);
 - » decode a quadrature signal (encoder sensor);
 - » measurement of the width of a pulse (high state duration).
 - » ...

Basic principle

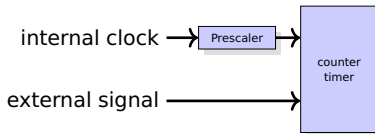
- » the timer value increases/decreases periodically;
- » When the timer reaches its maximum value, its value is reloaded (not necessarily at 0)



Timer resolution

- » the *timer resolution* is the time required to change its value by one unit;
- » A timer usually offers a *prescaler* of the input frequency to set the resolution:
 - » If the frequency divider is high, the timer resolution is higher and the timer takes longer to reach its maximum value;
 - » If, on the other hand, the frequency divider is low, the timer resolution is smaller, and the time measurement is more accurate.

The prescaler makes no sense if an external signal is used as input.

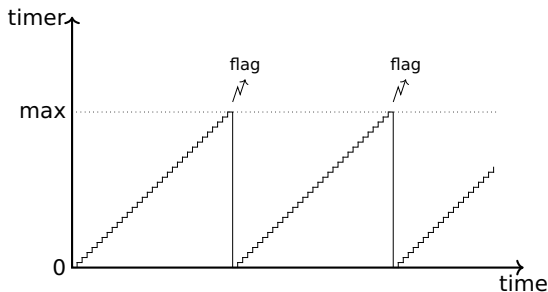


timer overflow

A timer overflows when:

- » its value gets from its maximal value to its reload value (*overflow*), when counting;
- » its value gets from 0 to its reload value (*underflow*), when decounting

A *flag* signals that an overflow/underflow has occurred. This flag can be used either under interrupt (see chapter p.1), or with software (*polling*).



The capacity of the timer is the set of values it can take. A N -bits timer have its values in:

$$[0; 2^N - 1]$$

In general, values for N are:

- » 8 bits \Rightarrow from 0 to 255;
- » 16 bits \Rightarrow from 0 to 65 535;
- » 32 bits \Rightarrow from 0 to 4 294 967 295;
- » 64 bits \Rightarrow from 0 to ... (read only usage).

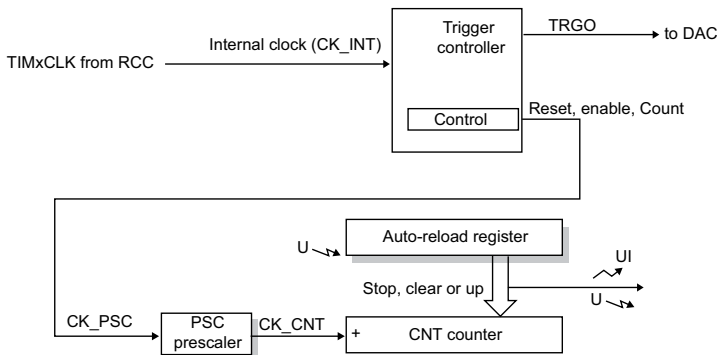
We will use 16-bits and 32-bits timers here.

On the STM32F303 target, different kind of timers are provided from basic to more advanced timers:

- » basic timers (low complexity, limited features): **TIM6**, **TIM7**
- » General purpose timers (medium complexity): **TIM2**, **TIM3**
- » General purpose timers (other features): **TIM15**, **TIM16** and **TIM17**
- » Advanced purpose timers (many features): **TIM1**, **TIM8** and **TIM20**

Basic timers TIM6, TIM7

- » 16 bit, only up counter, with auto-reload
- » can be linked to the DAC



The main input clock for each timer should be first enabled (see p. 117):

```
//input clock = 64MHz.  
RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;  
__asm("nop");  
//reset peripheral (mandatory!)  
RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST;  
RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM6RST;  
__asm("nop");
```

Control register CRx

CR1 Control Register 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	UIF RE- MAP	Res	Res	Res	ARPE	Res	Res	Res	OPM	URS	UDIS	CEN
				rw				rw				rw	rw	rw	rw

OPM One Pulse Mode: counter stops at next overflow

CEN Counter enable: should be set to 1

CR2 Control Register 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	MMS[2:0]			Res	Res	Res	Res
									rw	rw	rw				

Not used here.

Prescaler Register PSC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The **PSC** prescaler register is a 16-bit register that divides the input frequency by a programmable factor from 1 to 65535:

$$f_{CNT} = \frac{f_{PSC}}{PSC[15 : 0] + 1}$$

ex: if PSC=63, the input frequency for the counter **CNT** is 1MHz.

Note: On most micro-controllers, the prescaler is a power of 2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
UIF CPY	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res
r															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

The **CNT** register contains the current value of the timer (16-bit R/W register).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

The **ARR** (Auto Reload Register) is the max value of the **CNT** register: the **CNT** register counts from 0 to **ARR** (**ARR**+1 units).

Status Register SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	Res	UIF
															rc_w0

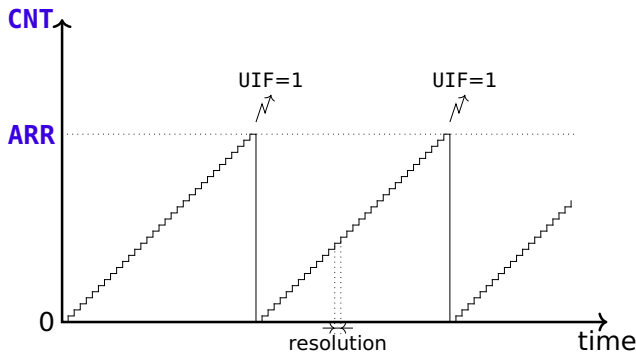
UIF the flag is set when the timer overflows (from **ARR** to 0):
Update Interrupt Flag.

The flag should be reset by software:

```
TIM6->SR = 0; //reset UIF
```

Registers summary

<i>register</i>	<i>field</i>	<i>bit</i>	<i>function</i>
CR1	CEN	0	Count <i>EN</i> able.
PSC			prescaler (frequency divider)
SR	UIF	0	overflow flag
CNT			<i>timer</i> current value
ARR			<i>Auto Reload R</i> egister



Exercise - Implement a delay function

We want to implement a function that performs a simple delay. The input clock is set to 64MHz.

- ▶ What is the max delay that can be done (with one timer loop only)?
- ▶ Implement a function that simply waits:

```
//ms in milli-seconds  
//ms should be <= 60 000  
void delay(unsigned int ms);
```

exercice 1

```
void delay(unsigned int ms);  
{
```

```
}
```

Correction - exercice 1

```
void delay(unsigned int ms);
{
    //check argument
    int arr = ms;
    if(arr > 60000) arr = 60000;

    //input clock = 64MHz.
    RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
    __asm("nop");
    //reset peripheral (mandatory!)
    RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST;
    RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM6RST;
    __asm("nop");

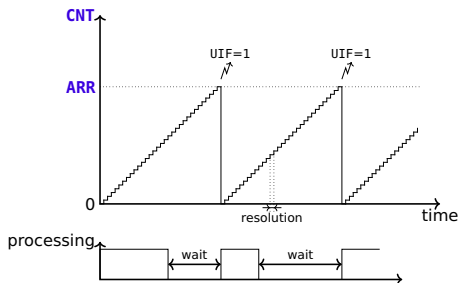
    TIM6->PSC = 64000-1;      //prescaler : tick@1ms
    TIM6->CNT = 0;
    TIM6->ARR = arr-1;        //auto-reload: counts 100 ticks
    TIM6->CR1 |= TIM_CR1_CEN; //config reg : enable
    while(! (TIM6->SR & TIM_SR_UIF)); //wait...
}
```

Synchronization

Objective

Include the calculation *inside the waiting* function, so as not to accumulate delays.

Example: a process requires from 1 to 3 ms, and should be repeated each 10ms...



Synchronization

In the synchronization loop:

- ») reinit the overflow flag;
- ») *insert here the processing part*, while the timer is counting;
- ») synchronization part: wait until the overflow flag occurs

note:

On some MCU, the overflow value is hardwired to the 65536 (on 16-bits). So you will have to update the timer value so that it performs the required number of steps.

Example

We have a stepper motor to control, and a function `motorStep()` is available to send a pulse to the power interface. We have to call this function at a frequency of 500Hz, so that the motor turns in continuous mode.

We also consider that there is another function `otherStuff()` that should be called at the same frequency. The duration of this function is set between 0.1 and 1.3ms.

- ▶ implement the `setup()` function to initialize the timer (we will use a `tick@1μs`)
- ▶ implement the control loop (inside `main()`) that calls periodically these 2 control functions.

Correction - synchronization

```
void setup(void) {
```

```
}
```

```
int main() {
```

```
}
```

Correction - synchronization

```
void setup(void) {
    RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
    __asm("nop");
    //reset peripheral (mandatory!)
    RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST;
    RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM6RST;
    __asm("nop");

    TIM6->PSC = 64-1;           // prescaler : tick@1us
    TIM6->CR1 |= TIM_CR1_CEN;   // config reg : enable
    TIM6->ARR = 2000-1;         // each 2ms (2000 ticks)
}

int main() {
    setup();
    while(1) {
        TIM6->SR &= ~TIM_SR_UIF; //reinit overflow flag
        motorUStep();           //application stuff, when the
        otherStuff();           //timer counts up
        while(! (TIM6->SR & TIM_SR_UIF)); //synchro
    }
}
```

Contents

1 Introduction

2 How to deal only with 0 and 1?

3 Specific C language operations

4 General Purpose I/O

5 Clock Sources

6 Pin muxing

7 Timer

8 Pulse Width Modulation

9 Interrupts

10 External Interrupt Handling

11 Serial Comm. (UART/I2C/SPI)

Extend timer possibilities with:

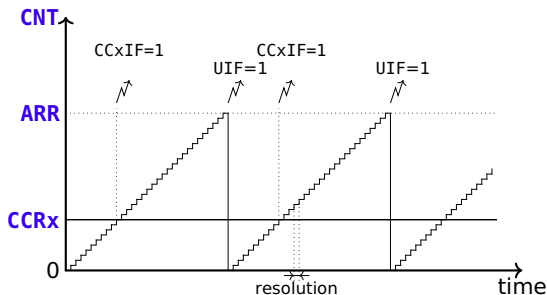
- ») the *timer* of chapter p. 7;
- ») a *comparator*: it is a mechanism that will continuously compare the value of a register with the value of the *timer*;
- ») possibly a physical *output* (on a μC pin).

This system will mainly be used to generate a digital output with a Pulse Width Modulation (PWM), in an autonomous way.

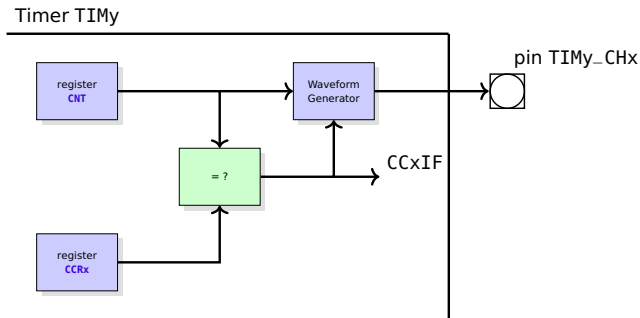
Principle

At all times, the value in the register **CCRx** is compared with the current value of the *timer*.

When the 2 values are matching, the flag CCxIF is set.



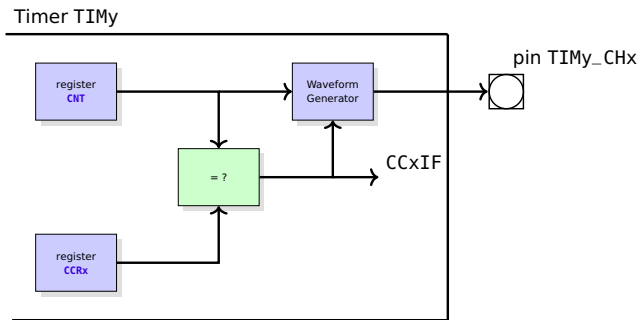
Basic block diagram



the PWM is used to control many types of actuators:

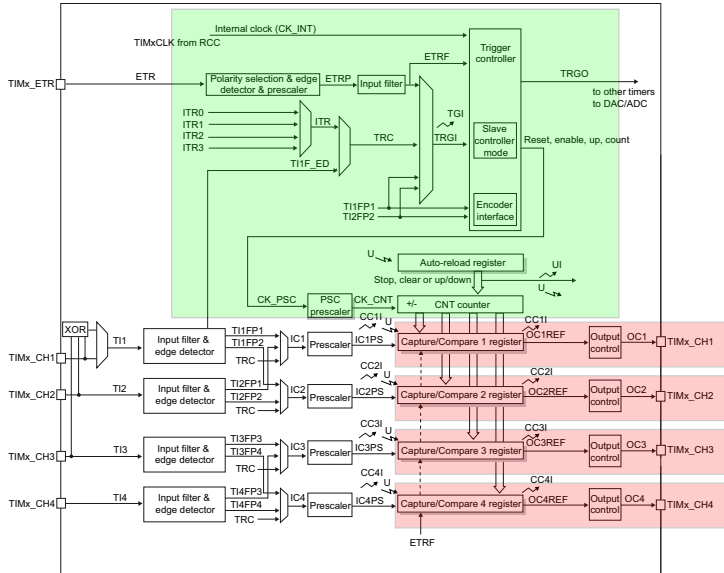
- » DC Motor (with power mosfet)
- » stepper motor
- » brushless motors
- » LEDs
- » ...

Basic block diagram



4 channels are associated to timers **TIM2/3/4**, but none for basic timers **TIM6/7**.

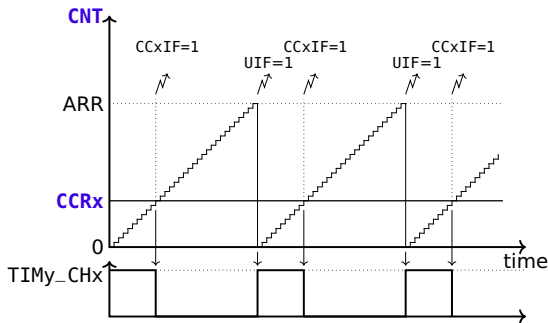
Block Diagram of TIM2/3/4



PWM mode edge-aligned

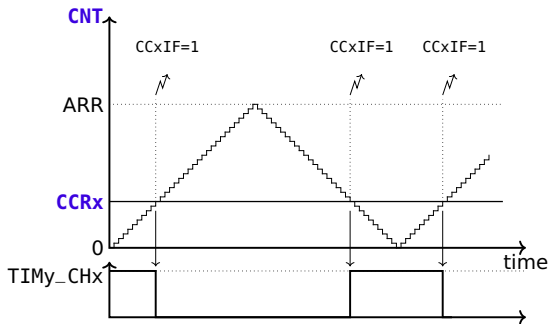
The simplest mode is edge-aligned, where:

- » the output is *set* after an overflow;
- » the output is *cleared* when the comparison matches.



PWM mode centered-aligned

In this mode the timer counts up and down, alternatively.
All the PWM channels are synchronized.



As a consequence

- » the PWM *frequency* is defined *only* from the timer frequency
For instance, with a 1MHz timer (prescaler=63), with ARR=99 (100 ticks):
 - ⇒ resolution is 1μs;
 - ⇒ PWM frequency is $\frac{1000000}{100} = 10\text{KHz}$
- » the *duty cycle* is defined with the comparison register **CCRx**:
The duty cycle is $\frac{\text{CCRx}}{\text{ARR}+1} = \frac{\text{CCRx}}{100}$ (often defined in %).

key interest

Once configured, the signal on the pin TIMy_CHx evolves autonomously, i.e. without any software.

The configuration is done in three steps:

- ») pin (alternative config.). See p. 6;
- ») timer (\Rightarrow PWM frequency), see p. 7;
- ») output comparison (\Rightarrow PWM duty cycle).

Control register CRx

CR1 Control Register 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	UIF RE- MAP	Res.	CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
				rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

OPM One Pulse Mode: counter stops at next overflow

CEN Counter enable: should be set to 1

CMS Center-aligned mode selection

00 edge-aligned mode

01 centered-aligned mode 1: comparison
interrupt flag only when counting down

10 centered-aligned mode 2: comparison
interrupt flag only when counting up

01 centered-aligned mode 3: comparison
interrupt flag when counting up and down

Capture Compare Mode Register CCMRx

CCMRx Capture Compare Mode Register x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	OC2M[3]	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OC1M[3]
							Res.								Res.
							rw								rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]			IC1PSC[1:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

OCxM Output Compare Mode: PWM mode is 0110

CCxS Capture Compare Selection: output is 00

OCxPE Preload Enable: new duty value taken into account only when there is an overflow.

Capture Compare Enable Register CCER

CCERx Capture Compare Enable Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Reserved	CC4P	CC4E	CC3NP	Reserved	CC3P	CC3E	CC2NP	Reserved	CC2P	CC2E	CC1NP	Reserved	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

CCxE Capture Compare Enable

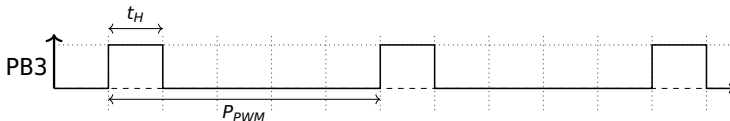
CCxP Capture Compare Polarity

Example: PWM signal

We want to realize the following signal on the pin PB3.

- » PWM frequency ($F_{PWM} = \frac{1}{P_{PWM}}$): 1KHz
- » duty cycle: 20% ($\frac{t_H}{P_{PWM}}$);

We fix **ARR** to 99, so that the duty cycle may be changed with 1%.



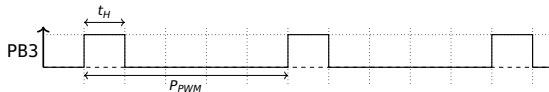
Example: PWM signal

The documentation (STM32F303 datasheet) shows:

Port	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8
	SYS_AF	TIM2/TIM15/ TIM16/TIM17/ EVENT	TIM1/TIM3/ TIM15/ TIM16	TSC	I2C1/TIM1	SPI1/ Infrared	TIM1/ Infrared	USART1/USA RT2/USART3/ GPCOMP6	GPCC GPCC GPCC
⋮									
PB3	JTDO/TRACE SWO	TIM2_CH2	-	TSC_G5_IO1	-	SPI1_SCK	-	USART2_TX	

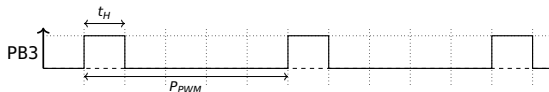
- ▶ Configure pin PB3 for the PWM;
- ▶ Configure timer to get the correct frequency;
- ▶ Configure comparison value to get a 20% ratio.

Example: PWM signal



```
void setup (void){  
  //1 - pin configuration:  
  //  alternate config 1 for PB3  
  pinAlt(GPIOB,3,1);  
  
  //...
```

Example: PWM signal



```
//...
```

```
//2 - timer configuration (use TIM2@10KHz)
```

```
RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
```

```
__asm("nop");
```

```
//reset peripheral (mandatory!)
```

```
RCC->APB1RSTR |= RCC_APB1RSTR_TIM2RST;
```

```
RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM2RST;
```

```
__asm("nop");
```

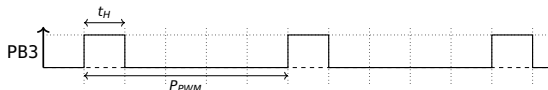
```
//config timer@10KHz, with 100 ticks (duty cycle at 1%)
```

```
TIM2->PSC = 64-1; //prescaler : tick@1us
```

```
TIM2->ARR = 100-1; //auto-reload: counts 100 ticks
```

```
//...
```

Example: PWM signal



```
//...
```

```
//3- PWM configuration
```

```
TIM2->CCMR1 &= ~TIM_CCMR1_CC2S_Msk; //channel 2 as output
```

```
TIM2->CCMR1 &= ~TIM_CCMR1_OC2M_Msk;
```

```
TIM2->CCMR1 |= 6 << TIM_CCMR1_OC2M_Pos; //output PWM mode 1
```

```
TIM2->CCMR1 |= TIM_CCMR1_OC2PE; //pre-load register TIM2_CCR2
```

```
TIM2->CR1 &= ~TIM_CR1_CMS_Msk; //mode 1 // edge aligned mode
```

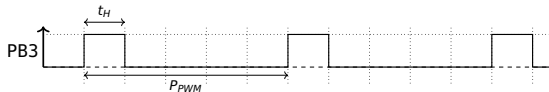
```
TIM2->CCER |= TIM_CCER_CC2E; //enable
```

```
TIM2->CR1 |= TIM_CR1_CEN; //config reg : enable
```

```
TIM2->CCR2 = 20-1; //20%
```

```
}
```

Example: PWM signal

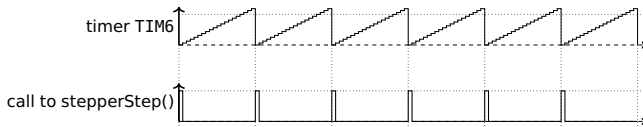


```
int main()
{
    setup()
    while(1){
        //nothing to do
        //signal generation is
        //autonomous
    }
}
```

- 1 Introduction
- 2 How to deal only with 0 and 1?
- 3 Specific C language operations
- 4 General Purpose I/O
- 5 Clock Sources
- 6 Pin muxing
- 7 Timer
- 8 Pulse Width Modulation
- 9 Interrupts**
- 10 External Interrupt Handling
- 11 Serial Comm. (UART/I2C/SPI)

Interest for interrupts

Ex: rotation of a stepper motor, with a precise frequency of 500Hz:



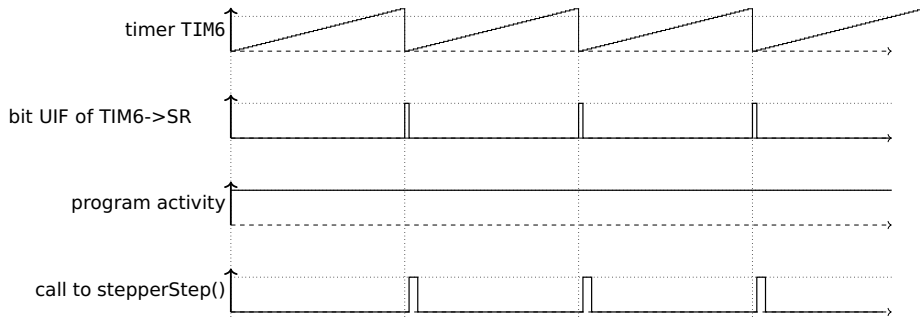
```
void setup(void)
{
    //input clock = 64MHz.
    RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
    __asm("nop");
    //reset peripheral (mandatory!)
    RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST;
    RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM6RST;
    __asm("nop");

    TIM6->PSC = 6400-1; //tick@100us
    TIM6->ARR = 20-1; //counts 20 ticks
    TIM6->CR1 |= TIM_CR1_CEN;
    //setup stepper motor
    stepperSetup();
}
```

```
int main() {
    setup();
    while(1)
    {
        //reset flag
        TIM6->SR &= ~TIM_SR_UIF;
        //1 step
        stepperStep();
        //wait...
        while(!(TIM6->SR & TIM_SR_UIF));
    }
}
```

Interest for interrupts

Time sequence of the application (after initialization):



Conclusion

The program spends its time running in a waiting loop!

- » It would be more appropriate to unload the microcontroller from this test, but to make the *device notify to the microcontroller*, by a logical signal, when the transition is detected
- » At this point, the *microcontroller would have to interrupt what it is doing* to process transition detection.

This means that:

- » the time sequence of the program is now *event triggered* from the external environment;
- » We exploit waiting times so that the microcontroller works on *other tasks*.

Interest for interrupts

2 questions:

- » what is the *hardware structure* that allows the microcontroller to process external requests?
- » What is the *software structure* associated?

An external signal requiring the microcontroller's attention is called:

- » an *interrupt* or
- » an *interrupt request* or
- » an *external request* or ...

- » the device is initialized;
 - » it runs in parallel with the core (which execute instructions);
 - » it has a defined objective:
 - » detect an edge on a pin (falling/rising)
 - » make an analog to digital conversion ;
 - » wait for a defined duration (chap. p.7);
 - » send or receive a message on a bus i2c, spi, uart, usb, ...
- » When the objective is reached:
 - » the device sends an *interrupt* to the microcontroller;
 - » the microcontroller *suspends its execution*;
 - » it executes the associated *Interrupt Handler*;
 - » At the end of the interrupt handlers, it resumes its normal behavior.

The STM32F303 defines 43 different interrupt sources, including

- » 8 for the timers (TIMx);
- » 7 for external interrupts (EXTI);
- » 1 for the analog to digital converters (ADC);
- » 10 for serial communication (3 *uart*, 1 *spi*, 2 *i2c* ,4 *can*);
- » ...

Routing of an interrupt request

NVIC

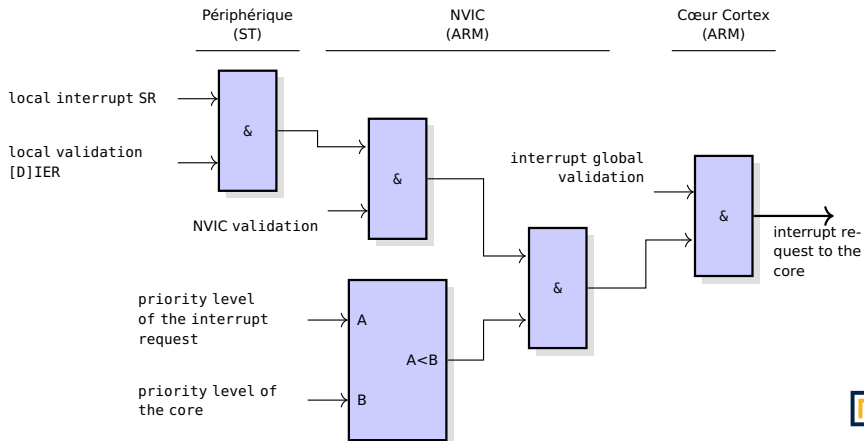
Interrupt management is done by a dedicated device, the **NVIC**: *Nested Vector Interrupt Controller*.

Enabling an interrupt is done at 3 levels:

- » at the device level: this is the **local activation**;
- » at the NVIC level: for the concurrent routing of interrupts
- » at the core level: this is the **global activation**. If interrupts are disabled at the core level, there is no more interrupt at all.

Routing of an interrupt request

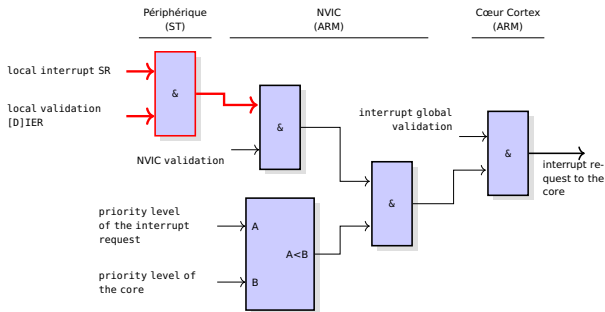
- 1 local validation (register [D]IER)
- 2 NVIC validation
- 3 interrupt priority (see p. 174);
- 4 global validation;



Routing of an interrupt request: local validation

The device should be configured. the validation is done through device register [D]IER (Interrupt Enable Register)

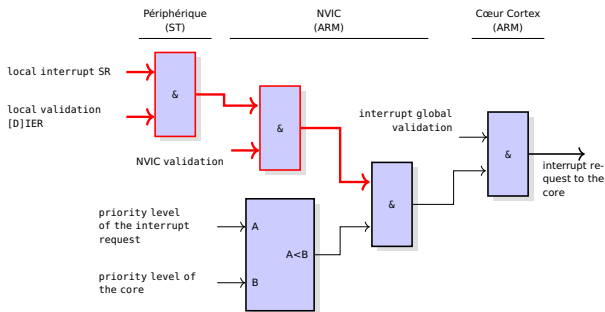
The register has the same structure as the status register (SR). When the device has done its work, a flag is set in the register SR.



Routing of an interrupt request: NVIC

Device validation NVIC is common to all ARM CortexM processors.
2 functions are provided by ARM (IRQ = *I*nterrupt *ReQ*uest):

```
void NVIC_EnableIRQ(int src);    //validation  
void NVIC_DisableIRQ(int src);  //invalidation
```



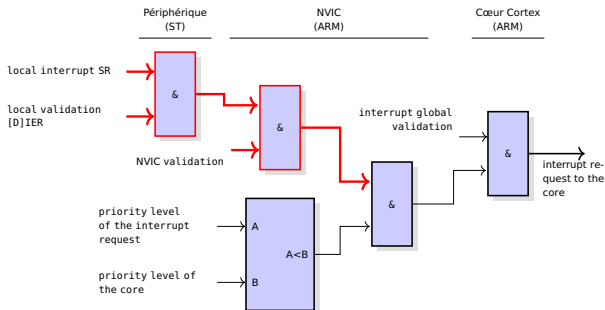
Routing of an interrupt request: NVIC

The argument (src) is the interrupt source id. ST defines symbolic name in the register definition file (stm32f303x8.h), with the name of the source, followed by _IRQn:

» TIM2 \Rightarrow TIM2_IRQn...

» but TIM6 for instance shares its interrupt source with the first DAC: \Rightarrow TIM6_DAC1_IRQn

NVIC_EnableIRQ(TIM3_IRQn); //ex timer TIM3

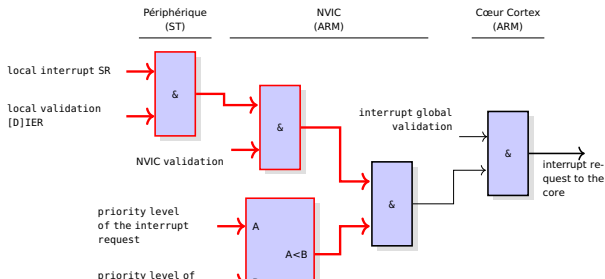


Routing of an interrupt request: priorities

The priority is introduced to manage *nested interrupt*).

When the core executes the code associated to an interrupt, it *inherits* the priority of the interrupt:

- ») if another interrupt with a higher priority occurs, the execution of the current interrupt is preempted (and the current core priority increases);
- ») if another interrupt with a lower priority occurs, the execution of the new interrupt is delayed until the end of the execution of the current interrupt handler.



Routing of an interrupt request: priorities

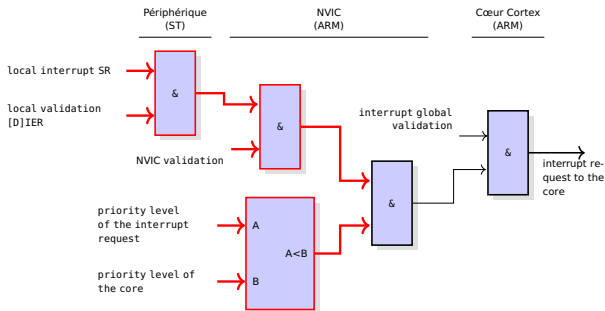
The priority is introduced to manage *nested interrupt*).

This Cortex-M4 core supports up to 16 priority levels¹: de 0 to 15.

Caution

The higher priority is the lowest value!

0 is the highest priority. . .



¹ARM allows up to 256 levels for the Cortex-M4

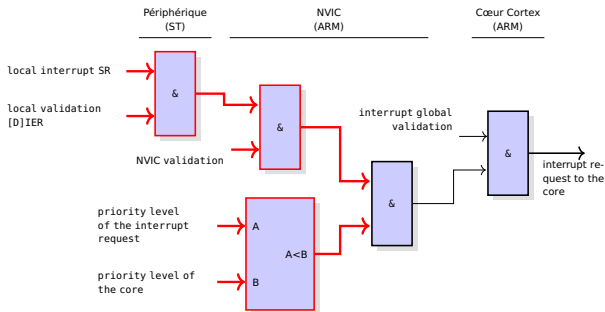
Routing of an interrupt request: priorities

As for the NVIC validation, ARM gives a function:

```
void NVIC_SetPriority(int src, int priority);
```

src the interrupt source;

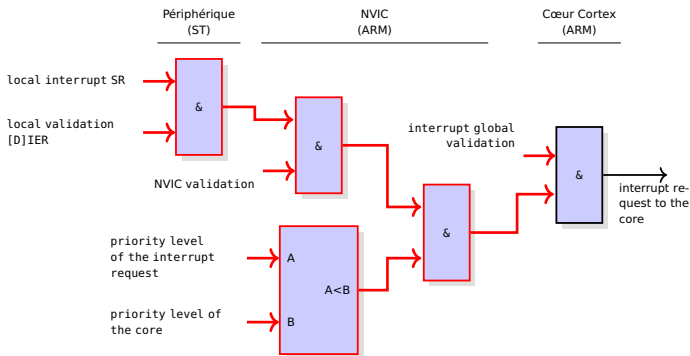
priority the priority (higher the priority, lower the value)



Routing of an interrupt request: global validation

Global disabling of interrupts quickly blocks interrupts from *all devices*.
By default, at startup, there is no global interruption blocking.

```
void __disable_irq (void); //no interrupt  
void __enable_irq (void);
```



Interrupt request

Once the interrupt request is validated, the processor:

- » *suspends the ongoing* execution of the program;
- » save the current context of the processor (registers);
- » execute the *interrupt request*;
- » restores the previous context;
- » *resumes the execution* of the program where it was interrupted.

There is a different *interrupt routine* for each interrupt source, whose name is formed by the device name, followed by `_IRQHandler...` but some interrupt routines may be shared by different devices:

```
void TIM6_DAC1_IRQHandler();
```

Interrupt request

As a result, for the synchronization:

- » if the device is correctly configured (TIM2 for instance)
- » if there is no other interrupt with a higher priority under execution;
- » if interrupts are validated at the core level (global)

Then:

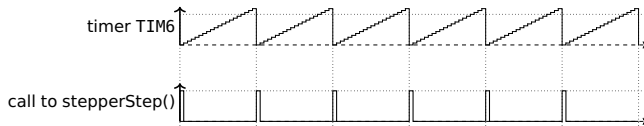
- » The processor suspends its execution and runs the interrupt request TIM2_IRQHandler.
- » The systems behaves *like if* the hardware was calling the function TIM2_IRQHandler().

Caution

The interruption routine *can't* have an argument (in or out)!
Communication between the interrupt routine and the rest of the program can only be done by global variables!

Full Example

get back to the example with the stepper motor (p. 163)



```
void setup(void)
{
    //input clock = 64MHz.
    RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
    __asm("nop");
    //reset peripheral (mandatory!)
    RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST;
    RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM6RST;
    __asm("nop");

    TIM6->PSC = 6400-1; //tick@100us
    TIM6->ARR = 20-1; //counts 20 ticks
    TIM6->CR1 |= TIM_CR1_CEN;
    //setup stepper motor
    stepperSetup();

    //enable interrupt
    TIM6->DIER |= TIM_DIER_UIE;
    NVIC_EnableIRQ(TIM6_DAC1_IRQn);
}
```

```
void TIM6_DAC1_IRQHandler()
{
    //1 step
    stepperStep();
    //acknowledge
    TIM6->SR &= ~TIM_SR_UIF;
}

int main() {
    setup();
    while(1)
    {
        //nothing!
    }
}
```

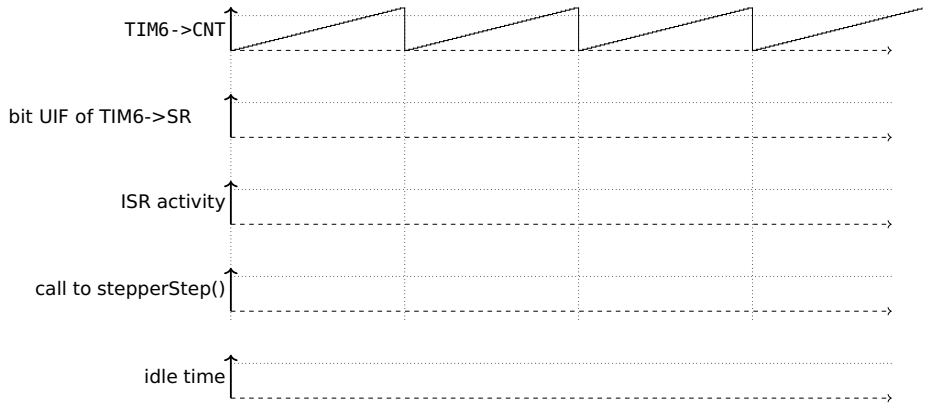

Full Example

To be noted:

- ») the initialization of the device is identical, only the local validation is added (DIER);
- ») the configuration of the NVIC is limited to a function call: no priority here, because there is only one interruption!
- ») the structure of the interrupt routine is similar to the *polling* approach as before but:
 - ») the routine is called only where there is effectively an overflow
 - ») the while loop is *removed*. If the code is executed, this is because there was an overflow.
 - ») the interrupt *should* be acknowledged. In the other case, the routine is called again and again!
- ») there is *many* cpu time to compute something else (in the main).

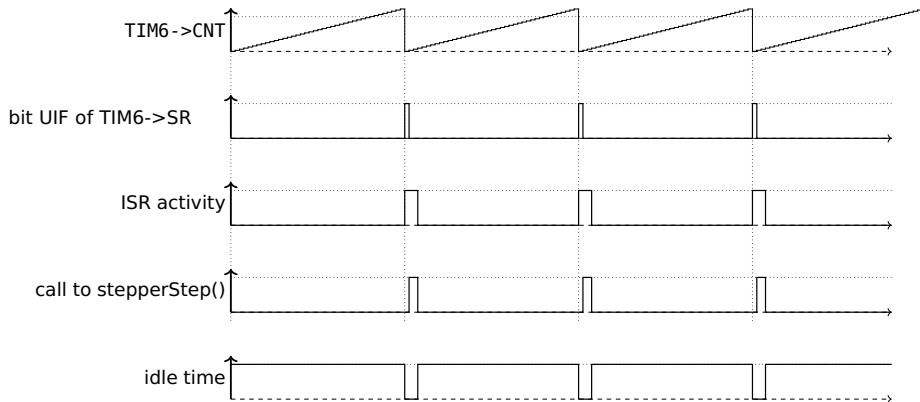
Full Example

Time sequence of the application (after initialization):



Full Example

Time sequence of the application (after initialization):



Exercise: Signal generation

We want to program a chaser under interrupt, with the leds (associated to pins PA0 to PA7). The refresh rate is set at 10Hz. The track will go to the left, then to the right, etc....

- ▶ give the I/O inits;
- ▶ give the timer init (TIM6);
- ▶ enable the interrupt and give the ISR code

The *idle task* (code executed in the `main()` loop), we toggle the output on PB0, at the max frequency of the processor.

Exercise: Signal generation

```
void setup() {  
    //leds chaser  
  
    //I/O signal  
  
    //TIM6  
  
}
```

Exercise: signal generation

```
void setup() {  
    //leds chaser  
    for(int led=0; led<8;led++)  
        pinMode(GPIOA, led, OUTPUT);  
  
    //I/O signal  
    pinMode(GPIOB,0, OUTPUT);  
  
    //TIM6 - input clock = 64MHz.  
    RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;  
    __asm("nop");  
    //reset peripheral (mandatory!)  
    RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST;  
    RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM6RST;  
    __asm("nop");  
  
    TIM6->PSC = 64000-1; //tick@1ms  
    TIM6->ARR = 10-1;    //counts 10 ticks  
    TIM6->CR1 |= TIM_CR1_CEN;  
}
```

Exercise: signal generation

- ▶ enable interrupt on TIM6:
 -) local validation (device)
 -) NVIC validation

```
void configIT()  
{  
    //local validation  
  
    //NVIC validation  
  
}
```

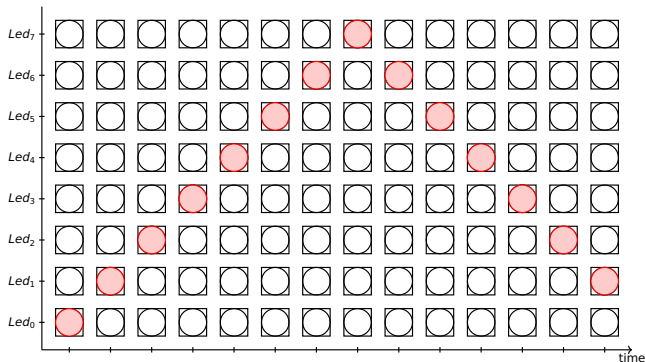
Exercise: signal generation

- ▶ enable interrupt on TIM6:
 - ») local validation (device)
 - ») NVIC validation

```
void configIT()
{
    //local validation
    TIM6->DIER |= TIM_DIER_UIE;
    //NVIC validation
    NVIC_EnableIRQ(TIM6_DAC1_IRQn);
}
```


Exercise: signal generation

The sequence to reproduce is:



- ▶ give the content of the ISR
 - ») acknowledge the interrupt;
 - ») get the value in the structure;

Exercice: signal generation

```
typedef struct {
    int size;
    int val[];
} seqType;

const seqType seq = {
    .size = ,
    .val = {

};

void TIM6_DAC1_IRQHandler()
{
    static int index = 0;
    //acknowledge

    /** application **

}
}
```

Exercise: signal generation

```
typedef struct {
    int size;
    int val[];
} seqType;

const seqType seq = {
    .size = 14,
    .val = {1,2,4,8,16,32,64,128,64,32,16,8,4,2}
};

void TIM6_DAC1_IRQHandler()
{
    static int index = 0;
    //acknowledge
    TIM6->SR &= ~TIM_SR_UIF;
    /** application **
    GPIOA->ODR &= ~0xFF; //clear
    GPIOA->ODR |= seq.val[index];
    //index
    index++;
    if(index >= seq.size) index = 0;
}
```

Another solution using **GPIOA->BSRR**?

Exercise: signal generation

main part:

- ▶ give the `main()` function that assembles the whole:

```
int main()
{

    while(1)
    {

    }

}
```

Exercise: signal generation

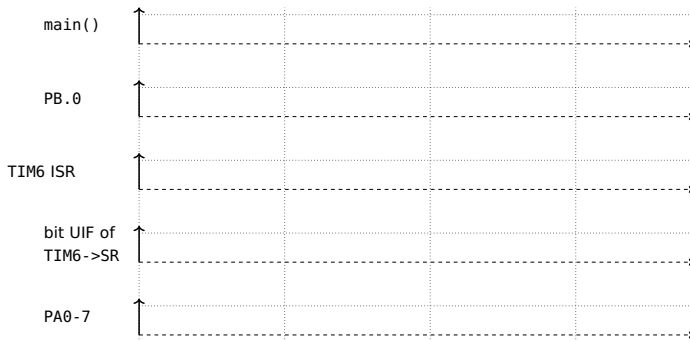
main part:

- ▷ give the main() function that assembles the whole:

```
int main()
{
    setup();
    configIT();
    while(1)
    {
        //nothing about chaser...
        pinToggle(PORTB,0);
    }
}
```

Exercise: signal generation

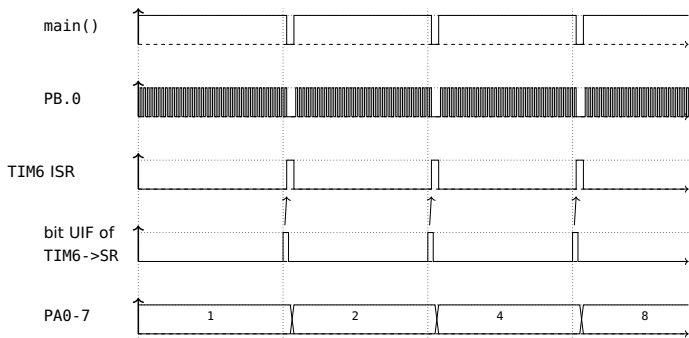
Chronogram



The processor only does *one thing at a time*. It is either in the main program (in the background task) or in the interrupt function. It is its high speed of execution that gives us the illusion of parallelism of execution.

Exercise: signal generation

Chronogram



The processor only does *one thing at a time*. It is either in the main program (in the background task) or in the interrupt function. It is its high speed of execution that gives us the illusion of parallelism of execution.

Functional architecture

An algorithm is used to describe sequential behaviour.

Objective

The purpose of the functional structure is to provide a *graphical* description to *show the links* between the different *entities* supposed to run in *parallel*, each performing a function in the system.

The functional structure does not replace the algorithm, it completes it by providing another level of description.

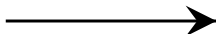
The *processes* supposed to run in parallel;

- » the hardware actions performed by peripherals;
- » the (temporary) software actions performed by interrupts;
- » the permanent software action (the background task);



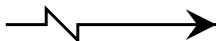
Links between entities may be:

) *Data Flow*



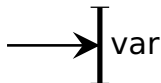
The arrow shows the direction of the data flow

) *Control Flow*



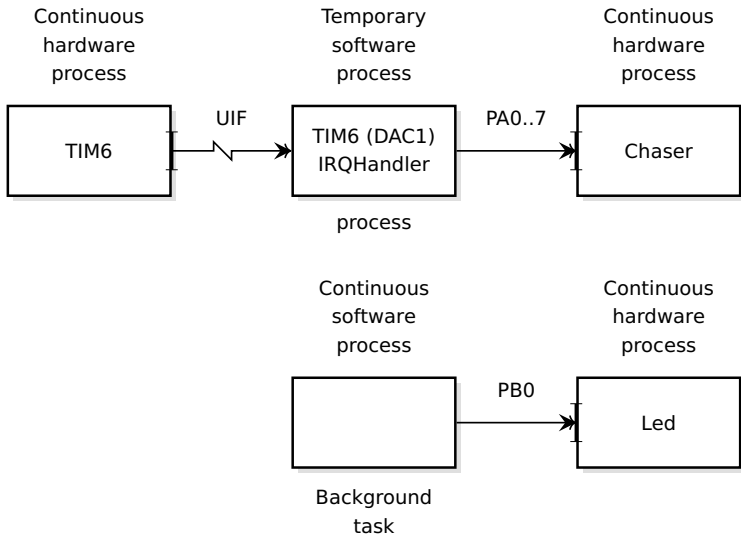
The arrow shows the direction of the control flow

A *global variable* should be defined, as it can be shared by different parallel tasks:



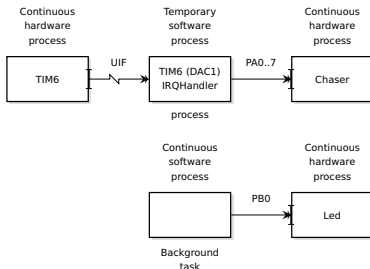
The arrow shows the direction of the data flow.
Here, the global variable `var` is written.

Functional architecture of the previous example



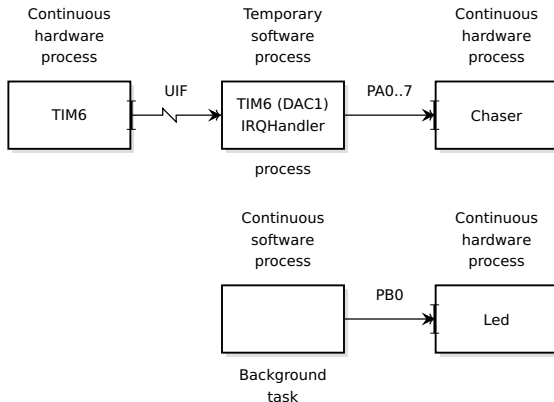
Functional architecture

- » the continuous hardware process (physical process) runs in *parallel* with the processor:
 - » timer and other peripherals;
 - » external peripheral, such as a LCD;
 - » sensors/actuators (external environment).
- » Continuous software process: The *background task* that runs all the time, except when it is interrupted. The algorithm of this action is the part of the program that is after the setup phase.



Functional architecture

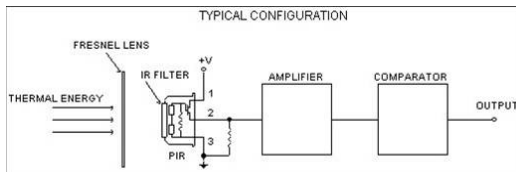
- » Temporary software process: The *process*, a procedure activated by the hardware, which *wakes up* as the result of hardware signal, and *interrupts* the background task, then *sleeps again* after doing its job.



- 1 Introduction
- 2 How to deal only with 0 and 1?
- 3 Specific C language operations
- 4 General Purpose I/O
- 5 Clock Sources
- 6 Pin muxing
- 7 Timer
- 8 Pulse Width Modulation
- 9 Interrupts
- 10 External Interrupt Handling**
- 11 Serial Comm. (UART/I2C/SPI)

Objective

- » extending the interrupt mechanism to detect a rising/falling edge on a pin
- » Example : PIR sensor (Pyroelectric (or Passive) InfraRed):

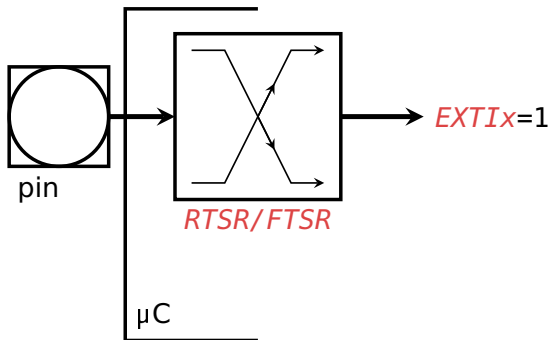


The output of the sensor is a pulse, that detect an infrared variation in the field of the sensor.

- » We have to dissociate:
 - » input level (low / high);
 - » input edge (rising/faling);

The EXternal Interrupt peripheral

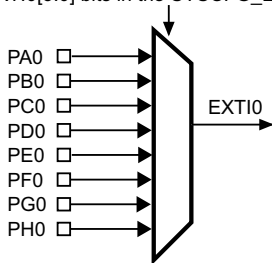
On the STM32, the peripheral is called *EXTI*: *EXT*ernal *I*nterrupt.
The principle is very basic:



The *EXT*ernal *I*nterrupt peripheral

The GPIOs are connected to 16 external interrupt lines:

EXTI0[3:0] bits in the SYSCFG_EXTICR1 register



The restriction is that if we have an external interrupt on pin PA3, no other external interrupt can be associated to P_x3, with $x \in [A, B, \dots, G]$

setting an external Interrupt

The following operations should be done:

- » the GPIO should be configured:
 - » clock for the GPIO port;
 - » input port mode, with possibly pull-up (push button, encoder, ...)
- » clock for the SYSCFG peripheral

```
RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;  
__asm("nop");
```

- » enable at least one external line (IMR register)
- » select the port associated to that interrupt (SYSCFG->EXTICR)
- » configure the edge detection (RTSR/FTSR registers)
- » enable the NVIC interrupt

Interrupt Mask Register

The interrupt mask register allows to enable an interrupt. For external interrupts, only the bit 0 to 15 are significant:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MR31	MR30	MR29	MR28	MR27	MR26	MR25	MR24	MR23	MR22	MR21	MR20	MR19	MR18	MR17	MR16
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

```
//interrupt for Px1 (x not yet defined)
```

```
EXTI->IMR |= EXTI_IMR_MR1; //Mask register 1
```

EXTI Configuration register

The EXTI Configuration register is defined in SYSCFG peripheral!! it defines the port chosen for external interrupt:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

4 registers are provided, for the $4 \times 4 = 16$ interrupts.

```
//select port B for exti1
```

```
SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PB;
```

Rising Trigger Selection register

The RTSR register defines if a rising edge may be detected. The FTSR (Falling Trigger Selection Register) behaves the same way

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TR31	TR30	TR29	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	TR20	TR19	TR18	TR17	TR16
rw	rw	rw							rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

```
//falling on exti1
```

```
EXTI->FTSR |= EXTI_FTSR_TR1;
```

Full example

Let's configure a push button (pull-up) connected to PB1, that toggles a led on PB0, under interrupt (1/2).

```
void setup()
{
    //config PB0 as output (Led)
    pinMode(GPIOB,0,OUTPUT);

    //config PB1 as input pull-up (push button)
    pinMode(GPIOB,1,INPUT_PULLUP);

    //config external interrupt on PB1
    //PBx associated to EXTIx => EXTI1 here
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
    __asm("nop");
    EXTI->IMR |= EXTI_IMR_MR1;           //Mask register 1
    EXTI->FTSR |= EXTI_FTSR_TR1;         //falling on exti1
    SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI1_PB; //port B for exti1
    NVIC_SetPriority(EXTI1_IRQn, 3); //NVIC config
    NVIC_EnableIRQ(EXTI1_IRQn);
}
```

Full example

Let's configure a push button (pull-up) connected to PB1, that toggles a led on PB0, under interrupt (2/2).

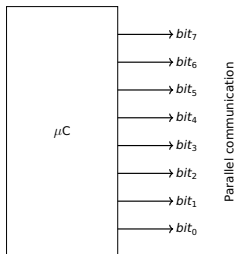
```
void EXTI1_IRQHandler()  
{  
    GPIOB->ODR ^= 1 ; //toggle led  
    EXTI->PR |= EXTI_PR_PR1; //it acknowledge  
}  
  
/* main function */  
int main(void)  
{  
    setup();  
    /* Infinite loop */  
    while (1);  
}
```


- 1 Introduction
- 2 How to deal only with 0 and 1?
- 3 Specific C language operations
- 4 General Purpose I/O
- 5 Clock Sources
- 6 Pin muxing
- 7 Timer
- 8 Pulse Width Modulation
- 9 Interrupts
- 10 External Interrupt Handling
- 11 Serial Comm. (UART/I2C/SPI)**

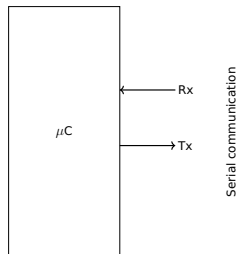
Introduction

Main methods to communicate between 2 devices:

-)) a *parallel* communication, in which each pin transmits one bit.



-)) a *serial* communication, where each bit is transmitted one after the other.



There are many serial communication protocols:

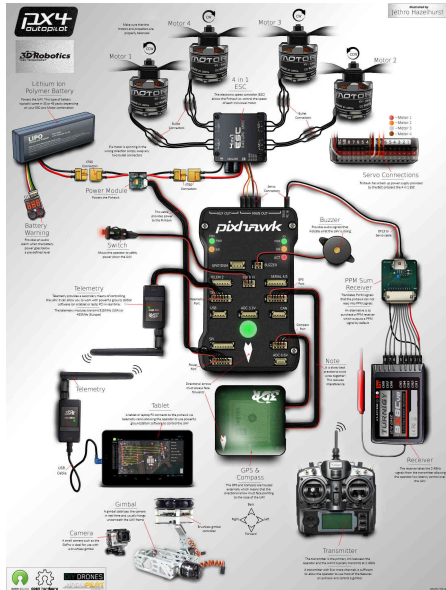
USB, UART, SPI, I2C, CAN, ...

Example: a drone

Global electronic architecture of a drone.

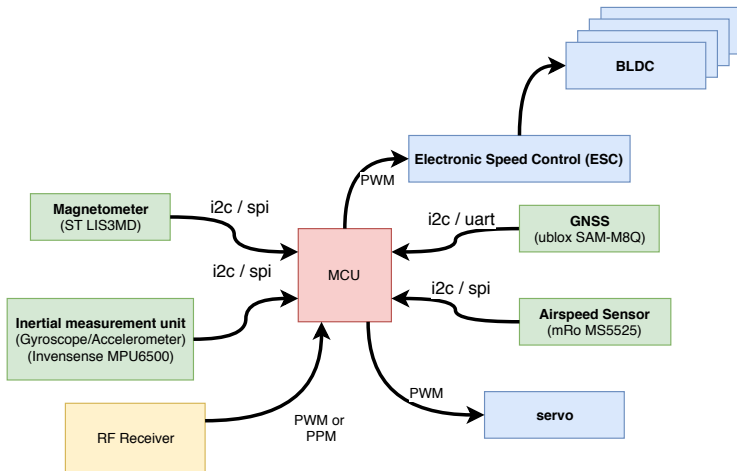
- » 4 brushless motors, with an ESC (Electronic Speed Control).
- » power supply
- » motherboard with embedded sensors
- » GNSS sensor
- » RF Receiver
- » telemetry
- » camera gimbal control

Image from <http://ardupilot.org>



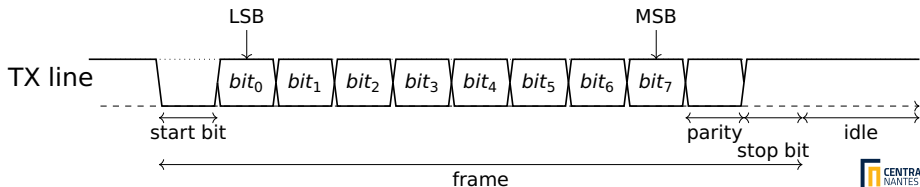
Example: a drone

Processor / device communication



Universal Asynchronous Receiver Transmitter (UART)

- » On a serial line, bits are transmitted one after the other (time multiplexing);
- » Transmission is point-to-point, full-duplex, with one RX line and one TX line.
- » Speed is expressed in bit per second (b/s) or *baud*;
- » When idle, the line is at high level;
- » The transmission starts with a *start* bit;
- » This *start* bit is used by the receiver to synchronize itself;
- » The first bit of the transmitted byte is the *LSB*.



UART : flow control

Flow control ensures that a signal is transmitted correctly.

The *parity check* is a simple way to *detect* a transmission error on a bit (an inversion). However, it does not allow this error to be corrected.

- » the sender and receiver count the number of high bits (1) in the sent frame.
- » With an even parity configuration, we will make sure that the number of bits of the frame is even. Thus, the parity bit will be set to:
 - » 0 if the number of bits in the frame is already even;
 - » 1 if the number of bits in the frame is odd, to make this number even.
- » With an odd parity, we make sure that the number of bits transmitted is odd (same approach).
- » This flow control allows to detect a transmission error on a bit (an inversion), but not to know which bit is wrong.

UART : characteristics

- » the *size* of the data: generally 8 bits, but 7 bits (ASCII) can be used;
- » the control of the transmission: a bit of *parity* even or odd, or no parity (*even, odd or none*)
- » *speed* of transmission (in bit/s): generally up to 115200 bits/s.
- » number of *stop* bits, generally programmable on 1, 1.5 or 2 bits.

We often use a configuration *115200 8N1*:

- » 115200 bauds;
- » 8 bits of data;
- » no parity (N);
- » 1 stop bit.

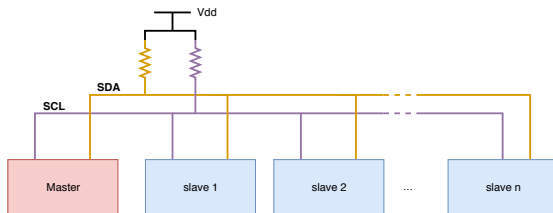
- » The I2C bus (Inter Integrated Circuit) is a local bus invented by Philips (now NXP).
- » It allows the connection between the components of the same system (μ C, RAM, real-time clock, EEPROM, LCD driver, remote I/O,...)
- » it's a *master/slave* bus, where the master initiates communication to 1 or more slaves;
- » the bus is synchronous bi-directional half-duplex:
 - » a line is reserved for the clock that is common for each device (*synchronous*);
 - » *bi-directional* (master \rightarrow slave or master \leftarrow slave);
 - » the communication is performed in only one direction at a time (*half-duplex*), because there is only one data line;

- » In the general case, there is only one master (μ C), and several slaves.
- » Each slave has an address (on 7 bits) \Rightarrow maximum of 128 slaves;
- » the transmission speeds are relatively low:
 - ≤ 100 kbits/s mode *standard*;
 - ≤ 400 kbits/s mode *fast*;
 - ≤ 1 Mbits/s mode *fast+*;
 - ≤ 3.4 Mbits/s mode *high speed*;
 - ≤ 5 Mbits/s mode *ultra fast* (unidirectional only);

I2C: Hardware structure

- » The communication uses 2 lines at high level when idle:
 - » *SDA*: Serial DAta line;
 - » *SCL*: Serial CLock line;
- » the outputs are *open drain* with only one pull-up resistor;
- » each circuit monitors the level on the bus lines;

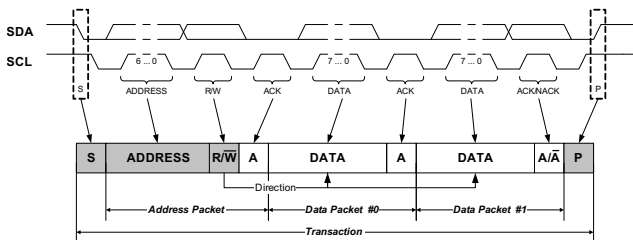
NOTE: A pull-up resistor is required on both lines! ($\sim 2K\Omega$ for a std mode)



- » *wired AND*: allows synchronization (several masters on the bus, and therefore several SCL), as well as arbitration (simultaneous data transmission on SDA).

I2C: bus transaction

- » the master always initiates the communication: it's the only device that manages the clock SCL



S Start of Frame

ADDRESS slave address (7 bits). MSB first.

R/W Read/write mode

A Acknowledge from the slave (ACK slot)

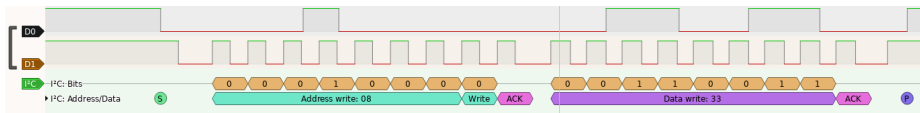
DATA 8 bits data (direction depends on previous bit R/W)

P End of Frame (stoP)

I2C: bus transaction

Exemple of a transaction, using a logic analyser:

- » line D0 is *SDA*
- » line D1 is *SCL*

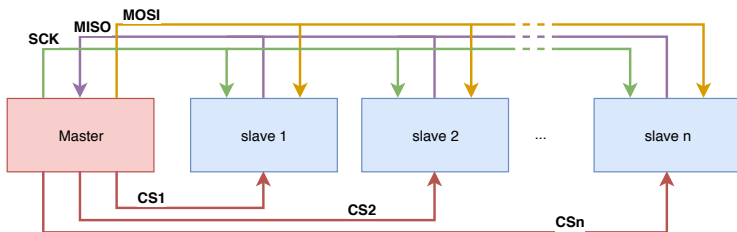


- » The SPI (*Serial Peripheral Interface*) was introduced by Motorola (become Freescale, bought by NXP in 2015) in the 80's.
- » like the I2C, it's a *master/slave* bus, where the master initiates communication to 1 or more slaves;
- » the bus is synchronous bi-directional full-duplex:
 - » a line is reserved for the clock that is common for each device (*synchronous*);
 - » *bi-directional* (master → slave or master ← slave);
 - » the communication may be done in two directions at the same time (*full-duplex*), because there is one data line for each direction;

- » In the general case, there is only one master (μ C), and several slaves.
- » each slave is selected using a specific I/O (*Chip Select*). This means one physical pin should be reserved for each slave.
- » the SPI allows *faster* transmissions than I2C ($> 10\text{MHz}$) because of the push/pull approach (vs open drain in I2C);
- » the SPI is a *de facto* standard. There are many adaptations.

SPI: Hardware structure

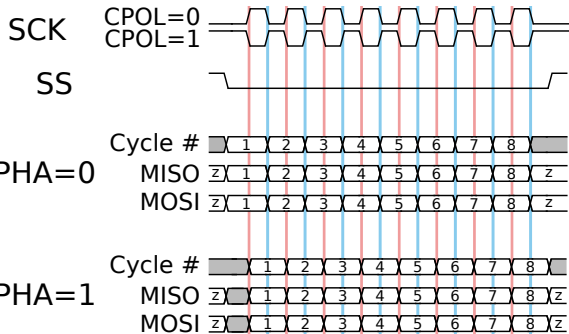
- » The communication uses 3 lines, and 1 Chip Select for each slave:
- » **MISO**: Data line *Master In, Slave Out* (or SO);
 - » **MOSI**: Data line *Master Out, Slave In* (or SI);
 - » **SCK**: *Serial Clock*;
 - » **CS_i**: *Chip Select* for slave *i* (low when the slave is selected)



SPI : Communication modes

» **CPOL** is the clock polarity. it gives the idle state.

» **CPHA** is the clock phase.



Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

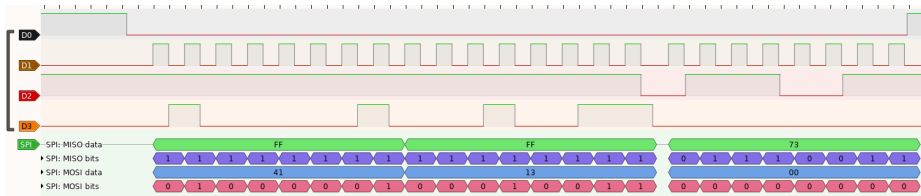
Modes 0 and 3 works in the same way, except in idle state (idem for modes 1 and 2).

image: Wikipedia

SPI: bus transaction

Exemple of a transaction, using a logic analyser (mode 0):

- » line D0 is *CS*
- » line D1 is *SCK*
- » line D2 is *MISO*
- » line D3 is *MOSI*



SPI: Hardware block in STM32

