

Embedded System Course 2019-20

C4µC

C/C++ Programming for Microcontrollers

Lecturers: Ing. Marco Santic, Ing. Walter Tiberti

marco.santic@univaq.it walter.tiberti@graduate.univaq.it

C4μC



- Introduction
 - Atmega328P
 - Toolchain
 - IDE
 - Examples
 - **C language recap**
- Basic functionalities
 - GPIO
 - Timers
 - Interrupt
 - Examples
- Comm. Interfaces
 - I2C
 - UART
 - SPI
 - Examples
- Other peripherals
 - ADC
 - GPIO bitbanging
 - Examples

C4μC - C basics

- Simple Data types and constants (`0_examples.c`)



```
/* machine-dependent types */

int integer_variable;

unsigned int unsigned_integer_variable;

short int small_integer;

long int long_integer;

unsigned short int small_unsigned;

/* machine-independent types (stdint.h) */

#include <stdint.h>

int8_t eight_bit_integer;

uint8_t unsigned_eight_bit; /* byte */

int16_t sixteen;

uint16_t unsigned_sixteen;

/* ... */
```

C4μC - C basics

- Simple Data types and constants (`0_examples.c`)

```
/* Other */

char character; /* default = signed */

float floating_point_32bit; /* IEEE 754 */

double fp_64bit;

/* pre-processor constants */

#define CONSTANT 42

#define NAME "jack"

/* small macro */

#define DOUBLE(x) ((x)+(x))
```



C4μC - C basics

- Functions (`0_examples.c`)

```
int sum(int a, int b); /* function prototype */\n\n/* Another function: main */\nint main(void)\n{\n    /* code of our application */\n\n    /* Function call */\n    int d = sum(1, 2);\n\n    return 0;\n}\n\n/* implementation of sum*/\nint sum(int a, int b)\n{\n    return a+b;\n}
```

C4μC - C basics

- Simple statements (`0_examples.c`)

```
int a, b, c;

a = 33; /* assignment */
a = b;
a++; /* post-increment */
++a; /* pre-increement */
a = a + 1; /* addition and assignement */
a += 1; /* inplace addition */
b = b - 1; /* subtraction */
b -= 1; /* inplace subtraction */
c = a * b; /* multiplication */
c *= 2; /* inplace mult. */
c = c / 2; /* division */
c /= 1; /* inplace division */
c = a % 2; /* modulo */
c %= 2; /* inplace modulo */
```



C4μC - C basics

- Simple statements (`0_examples.c`)

```
int a, b, c;

a == b; /* comparison (returns 0 or non-zero) */
a != b; /* negative comparison */
a > b;
a >= b;
a < b;
a <= b;
!a; /* logical negation */
a && b; /* logical AND */
a || b; /* logical OR */
```

C4μC - C basics

- Simple statements (`0_examples.c`)

```
int a, b, c;

~a; /* bitwise NOT */
a = b & 7; /* bitwise AND */
a &= 7;
a = b | c; /* bitwise OR */
a |= 1;
a = b ^ c; /* bitwise XOR */
a ^= 7;
b = a << 2; /* left shift */
b <<= 2;
c = b >> 2; /* right shift */
c >>= 2;
```



C4μC - C basics

- Conditions Constructs (`0_examples.c`)

```
/* condition: 0 = false, non-zero = true */

int condition = 1;
if (condition) {
    /* .... */
} else {
    /* .... */
}

if (a == 3) {
    b = 7;
} else {
    b = 8;
}

/* ternary form */
(a == 3) ? b = 7 : b = 8;
```



C4μC - C basics

- Loop Constructs (`0_examples.c`)

```
/* head check */
while (condition) {
    /* .... */
}

/* tail check */
do {
    /* ... */
} while (condition);

/* for loop
for (init; condition; advance) {
    ...
}
*/
```



C4μC - C basics

- Loop Constructs (`0_examples.c`)

```
while (a < 10) {  
    b += 3;  
}  
  
do {  
    a++;  
    c = a & 1;  
} while (c != 0);  
  
/* inline declaration of variable inside for statement is NOT VALID in C89 */  
/* int i = 0; */  
for (int i=0; i<100; ++i) {  
    a *= a;  
}
```

C4μC - C basics

- Compound data types (`0_examples.c`)

```
/* array */  
/* type name[size]; */  
int array[100];  
int array1[2] = {1, 2};
```

```
/* struct */  
struct structure {  
    int a;  
    uint8_t b;  
};  
  
struct struct2 {  
    uint8_t a : 7;  
    : 1;  
    uint16_t b;  
};
```

C4μC - C basics

- Compound data types (`0_examples.c`)

```
union union1 {
    int a;
    char b;
};

typedef int my_type_t;
typedef struct structure structure_t;

/* Equivalent */
struct structure s1;
structure_t s2;

int e = s1.a;
```

C4μC - C basics

- Pointers (`0_examples.c`)

```
int pointed = 4;
int *pointer = &pointed;

int array2[100];

/* Equivalent */
int *p2 = array2;
int *p3 = &array2[0];

/* Equivalent */
array2[5];
*(p2 + 5);

structure_t t[10];

structure_t *p = &t[3];

/* Equivalent */
int r1 = (*p).a;
int r2 = p->a;
```

C4μC - C basics

- Function pointers (`0_examples.c`)

```
int (*funct_ptr)(int a, int b);  
  
funct_ptr = sum;  
  
/* Function call */  
  
int c2 = sum(1, 2);  
  
/* Function call using funct pointer*/  
  
int d = funct_ptr(3, 4);
```



C4μC - C basics

- Casting: forcing a variable of a type to be of another type

```
/* Casting: operator "()" */

uint16_t a = 300;
uint8_t b = (uint8_t) a;

/* declaration and assignment of some variables */
int int_number;
int int_number2 = 10;
float float_number = 6.34;
float float_number2;
char a_char = 'A';

/* castings */
int_number = (int) float_number;           /* assigns value 6 (integer part) */
int_number = (int) a_char;                 /* assigns value 65 (ASCII code) */
float_number2 = (float) int_number2;        /* assigns value 10.0 (float value) */

/* useful in division */
/* this operation ensures we have a floating point result */
float_number = (float)int_number/(float)int_number2;

float_number = /*(float)*/ int_number/(float)int_number2;
```

C4μC - C basics

- Casting: how many "casting"?



```
uint16_t simple_integer = 1234;  
  
char c5 = '5';  
  
/* implicit extension */  
  
uint32_t long_integer = simple_integer;  
uint32_t long_integer2 = c5;  
  
/* implicit truncation */  
  
uint8_t small_integer = simple_integer;  
  
/* (explicit) cast */  
  
float floatingpoint = (float) simple_integer;  
int greekpi = (int) 3.14;  
  
float fp2 = 5.25f;  
simple_integer = (uint16_t) fp2; /* WARNING: see IEEE 754 standard */
```

C4μC - C basics

- Casting pointers

```
uint16_t pointer1 = &simple_integer;

uint8_t pointer2 = (uint8_t *) &simple_integer; /* pointer casting, value is
conserved */

/* Note that, although the pointers point to the same address... */
pointer1++; /* pointer gets incremented by sizeof(uint16_t) = 2 bytes */
pointer2++; /* pointer gets incremented by sizeof(uint8_t) = 1 byte */

uint8_t small_array[4] = {1,2,3,4};

/* In order to re-create an uint32_t value from the array above,
 * we first cast the array name to a uint32_t pointer, then we
 * deference the pointer to obtain the pointed value */
uint32_t reconstructed = *((uint32_t *)small_array);

/* another useful example */

uint8_t buffer1[128];

struct YourComplexStructure cs1 = *((struct YourComplexStructure *)buffer1);
```

C4μC - C basics



- Casting pointers, a little challenge for you...

```
/* ... */  
  
uint16_t array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
uint16_t *arr_u16_pt = array;  
  
uint8_t *arr_u8_pt = (uint8_t *)arr_u16_pt;
```

```
/* ... */  
  
arr_u16_pt[4] = ??  
  
arr_u8_pt[4] = ??
```

array[]	0	1	2	3	4	5	6	7	8	9	
value(dec)	1	2	3	4	5	6	7	8	9	10	
memory(MSB)	0 0	0 1	0 2	0 3	0 4	0 5	0 6	0 7	0 8	0 9	0 a

C4μC - C basics

- Casting pointers, a little challenge for you...

```
/* ... */  
  
uint16_t array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
uint16_t *arr_u16_pt = array;  
  
uint8_t *arr_u8_pt = (uint8_t *)arr_u16_pt;
```

```
/* ... */  
  
arr_u16_pt[4] =  
0x0005  
  
arr_u8_pt[4] =  
0x00
```

array[]	0	1	2	3	4	5	6	7	8	9
value(dec)	1	2	3	4	5	6	7	8	9	10
memory(MSB)	0 0	0 1	0 2	0 0	0 3	0 4	0 0	0 5	0 6	0 7

- Let's have a better understanding...

C4μC - C basics

- Pointer arithmetics & casting

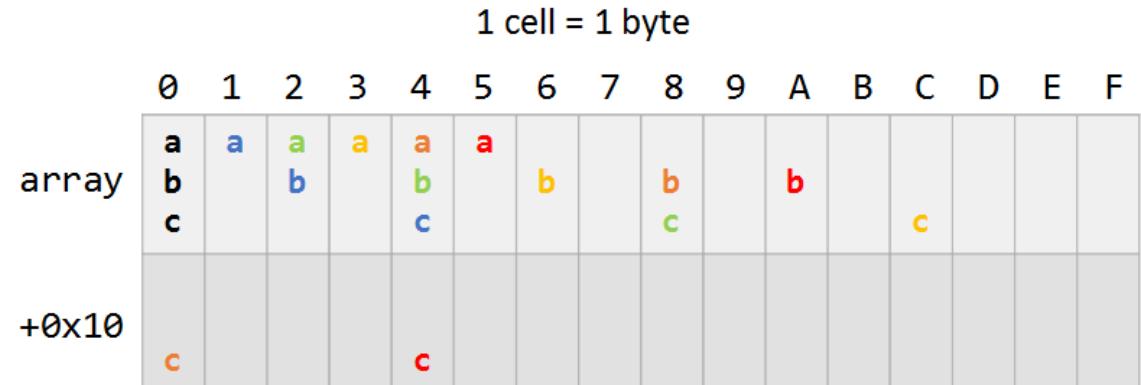


```
uint8_t array[32];
```

```
uint8_t *a = array;  
uint16_t *b = (uint16_t*)array;  
uint32_t *c = (uint32_t*)array;
```

```
for (int i=0; i<5; i++) {
```

```
    a++;  
    b++;  
    c++;
```



1 cell = 1 byte

i = 0 : a → 0x01, b → 0x02, c → 0x04
i = 1 : a → 0x02, b → 0x04, c → 0x08
i = 2 : a → 0x03, b → 0x06, c → 0x0C
i = 3 : a → 0x04, b → 0x08, c → 0x10
i = 4 : a → 0x05, b → 0x0A, c → 0x14

```
}
```

C4μC - C basics

- Let's suppose the area of memory which our program intend to use is the following 64 byte block:

address		values
.....	.	0..1..2..3..4..5..6..7..8..9..a..b..c..d..e..f.
0x0000	00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0010	00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0020	00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0030	00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- When we declare a (global) variable, the compiler allocates it inside the memory. Thus, after the following declarations:

```
uint8_t array1[5] = {8, 9, 10, 11, 16};  
uint16_t *ptr1 = NULL;  
uint32_t var1 = 0x12345678;  
uint16_t array2[6] = {6, 16, 26, 1000, 2048, 0xFFFF};  
uint8_t *ptr2 = NULL;
```

C4μC - C basics

- The block of memory becomes: (suppose 16bit-long pointers and BIG endian byte order)

address		values
.....		0..1..2..3..4..5..6..7..8..9..a..b..c..d..e..f..
0x0000		08 09 0A 0B 10 00 00 12 34 56 78 00 06 00 10 00
0x0010		1A 03 F8 08 00 FF FF 00 00 00 00 00 00 00 00 00 00
0x0020		00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0030		00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- By examining how the compiler layed out the variables, we see that:
 - The (base) address of array1 is 0x0000 and it occupies $5 * 1 = 5$ bytes
 - The address of ptr1 is 0x0005 and it occupies 2 bytes
 - The address of var1 is 0x0007 and it occupies 4 bytes
 - The (base) address of array2 is 0x000B and it occupies $6 * 2 = 12$ bytes
 - The address of ptr2 is 0x0017 and it occupies 2 bytes

C4μC - C basics

- Now, we do the following assignments:

```
ptr1 = &array2[2];  
ptr2 = &array[4];
```

address	values
.....	0..1..2..3..4..5..6..7..8..9..a..b..c..d..e..f.
0x0000	08 09 0A 0B 10 00 0F 12 34 56 78 00 06 00 10 00
0x0010	1A 03 F8 08 00 FF FF 00 04 00 00 00 00 00 00 00 00
0x0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- As you can see, now ptr1 = 0x000F and ptr2 = 0x0004.
- Now, we use pointer arithmetic to move pointers:

```
ptr1 += 1;  
ptr2 -= 2;
```

- Since ptr1 is a pointer to uint16_t (which is long 2 bytes), moving the pointer one element ahead means adding 2 to its value. In the same manner, since ptr2 is a pointer to uint8_t (which is long 1 byte), moving to 2 elements before means subtracting $2 * 1 = 2$ from its value.

C4μC - C basics



- So, after the movements:

- ptr1 becomes 0x11 and now has the address of array2[3] (0x3F8, which is 1000 in decimal)
- ptr2 becomes 0x02 and now points to array[2] (9)

address	values
.....	0..1..2..3..4..5..6..7..8..9..a..b..c..d..e..f.
0x0000	08 09 0A 0B 10 00 11 12 34 56 78 00 06 00 10 00
0x0010	1A 03 F8 08 00 FF FF 00 02 00 00 00 00 00 00 00
0x0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- Observe that, if we do:

```
ptr2 = (uint8_t *) ptr1;
```

- Note: the cast is needed since the types of the two pointer are different.

However, since the size of a `uint8_t*` and a `uint16_t*` is the same (2 bytes), the casted value is unaltered

C4μC - C basics



address	values
.....	0..1..2..3..4..5..6..7..8..9..a..b..c..d..e..f..
0x0000	08 09 0A 0B 10 00 11 12 34 56 78 00 06 00 10 00
0x0010	1A 03 F8 08 00 FF FF 00 11 00 00 00 00 00 00 00 00
0x0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- With ptr2 containing the address 0x11, it points to the first byte of array2[3].
- By doing the following:

```
*ptr1++ = 0xAAAA; /* first assign the pointer, then access and assign the value */  
*++ptr2 = 0x88; /* first increment, then access and assign the value */
```

C4μC - C basics

- We have:

address	values
.....	0..1..2..3..4..5..6..7..8..9..a..b..c..d..e..f.
0x0000	08 09 0A 0B 10 00 13 12 34 56 78 00 06 00 10 00
0x0010	1A AA 88 08 00 FF FF 00 12 00 00 00 00 00 00 00 00
0x0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- As you can notice, ptr2 moves to 0x12 while ptr1 moves to 0x13. This is due the fact that the increment of a pointer is dependent on the type of the variable to which it is pointing to.
- So
 - incrementing a uint8_t pointer or any other 8 bit type, adds 1 to the pointer
 - incrementing a uint16_t pointer or any other 16 bit type, adds 2 to the pointer
 - incrementing a uint32_t pointer or any other 32 bit type, adds 4 to the pointer
- Finally:
 - incrementing a pointer to a uint8_t pointer or any other pointer type, adds *X* to the pointer, where X is the size of the pointer type in the considered architecture. In our case it's 2 bytes.