Introduction

C is the most frequently used language to program microcontrollers. Its syntax is very similar to that of Java.
This document is not a course on C, its purpose is to highlight the principal differences between C and languages which are commonly taught to beginners (Python and Java).
Among the fundamental differences, let us quote:

- C is a compiled language, whereas Python is an interpreted language and JAVA is a mixed of both, since it is compiled for a virtual machine. It makes C programs non portable, which is in general not a constraint in embedded systems since the programs are in this case developed for a specific platform. On the other hand, the compilation produces a faster executable code
- C does not have dynamic memory management like a Garbage Collector. The use of dynamic variables is possible, but must be entirely managed by the program.
- C does not have any management of the exceptions (unlike the instruction *try* of Python). These tests must be managed by the program (divide by zero, overflow of variables…).

Variables

The C variables are typed; the types are almost the same as in Java (cf programmer's guides).
The variables must be explicitly defined before being used.
Variables defined outside any function are global.
Variables defined inside a function are local to the function.
If a local variable has the same name as a global variable, it "masks it"; the access to the global variable becomes impossible within this particular function.

The Boolean type

No Boolean type is defined in C; the conditional instructions (if, while, switch) test an integer variable (or an expression whose result is an integer), which is considered as *false* if it is 0 and *true* if not.

Similarly, logical operators (!, && et ||) and relational operators (==, !=, <, >, <= et >=) give an integer result (0 for *false* and 1 for *true*).

C is not an object-oriented language

Classes and their associated concepts (polymorphism, heritage, method, encapsulation) don't exist in C.
We can however define a data type called *structure* to group several data, like the fields of a class. For example, *the PORTBbits* variable (described in the programmer's guide) is a structure.

Functions

In C, all functions are global (no function can be defined within another function).
That implies that their name must be unique (the names are case-sensitive).
The concept of overriding functions does not exist.

Modularity

A program C can (and should) be arbitrarily cut out in several files by the programmer.
Each file is compiled separately. It frequently happens that the variables and functions defined in a file are used in another file. For example, let us take two files *main.c* and *math.c* containing each one a function:

```
main.c
# include "math.h"

int main() {
   int has = 2;
   int B = 3;

   C = pow(a, b);
        .
        .
        .
}
```

```
math.c
int C;

int  pow(int bases, int exp) {
   length result = 1;

   while (exp > 0) {
      result * = bases;
      exp --
  }
   return(result);
}
```

To compile the file *main.c*, the compiler must know the function *pow()*. However the complete definition of the function is not required, the *prototype* or the *declaration* of the function is enough:

```
length pow(int bases, int exp);
```

This declarations contains all informations required, namely the number of parameters and their type, as well as the type returned by the function.

The same mechanism also applies to variables shared by several files.

Those declarations are gathered in *Header files*, with a suffix h.
This header file must be included in all files using the functions declared in the header; in our example, the first line of *main.c* includes the file *math.h.*

# *include*, plays a role similar to the instruction *importation* of Python and JAVA.

The file *math.h* of our example will be thus:

```
// global variable defined in math.c and used in  main.c
extern int C;

// declaration of the function pow, which calculates base^exp
int pow(int bases, int exp);

//defines constant pi
# define pi 3.14
```

To differentiate the declaration from the definition of a variable, the keyword *extern* is added in front of the declaration, to specify that this variable is defined outside the file in which the header file is included.

The last line of *math.h* is a precompilation macro (like *#include*). It allows to define the constant *π* in a symbolic manner, without using of any memory storage.
Precompilation macros are dealt with during the first pass of the C compiler, before the compilation itself. It is in fact a command to edit the source code:

- the macro *#define* indicate to the compiler that it must seek its 1st^t argument (pi in this case) in the code and replace it by its 2nd argument (3.14), exactly like the command *replace* of a text editor.
- In the same way, the macro *#include* indicates that it should be replaced by the content of the file in parameter.
- There exists other precompilation macros, but we will not present them in this document.