

# Le C embarqué, par l'exemple.

Samuel Poiraud – 2020



















Ce document vous propose de découvrir les principaux concepts du langage C, par l'exemple, dans le contexte d'une cible embarquée (STM32F103).

Les notions sont présentées sous forme d'exemples indépendants. Chaque chapitre présente une notion, dans un but pédagogique précis.

L'évaluation des compétences se fera sous la forme d'un devoir surveillé (2h) en deux parties :

- 40 questions issue d'une liste connue, disponible à la fin de ce document (/20 points)
  - Bonne réponse : +1 pt
  - Mauvaise réponse : -2 pts
- 20 nouvelles questions (/20 points)

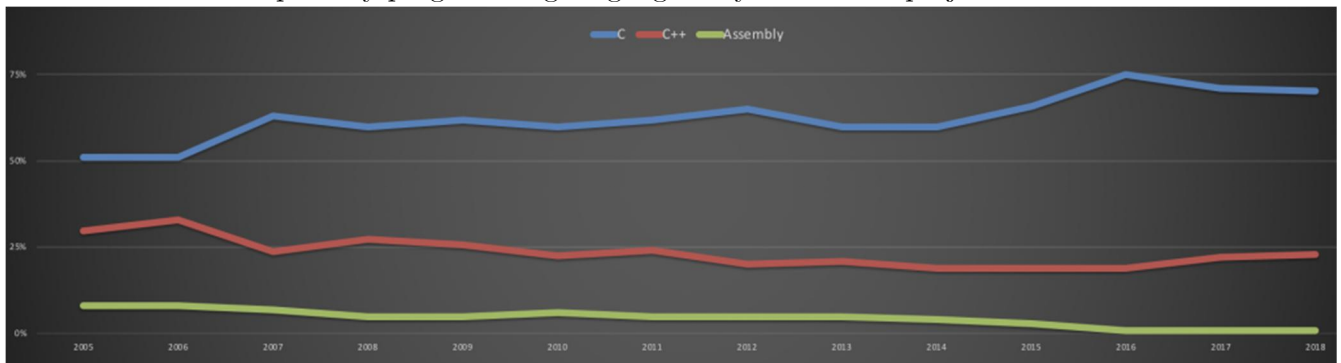
2017 : top 10 des langages informatiques :

| <div><div>Web</div><div>Mobile</div><div>Enterprise</div><div>Embedded</div></div> |   |   |                  |
|--|---|---|------------------|
| Language Rank  | Types   |   | Spectrum Ranking |
| 1. Python  |    |    | 100.0            |
| 2. C   |  |  | 99.7             |
| 3. Java  |  |  | 99.4             |
| 4. C++   |  |  | 97.2             |
| 5. C#  |  |  | 88.6             |
| 6. R   |  |   | 88.1             |
| 7. JavaScript  |  |  | 85.5             |
| 8. PHP   |  |   | 81.4             |
| 9. Go  |  |  | 76.1             |
| 10. Swift  |  |  | 75.3             |

Plus qu'un (ancien) langage informatique, le langage C est un langage incontournable dans le monde des systèmes embarqués.

Sondage à destination des développeurs de systèmes embarqués :

- "What is the primary programming language for your current project?".



Voici un comparatif de l'efficacité énergétique, de leur temps d'exécution et de l'occupation mémoire des programmes conçus selon ces différents langages. On comprend l'importance du C dans l'embarqué.

| Total          |        |                |       |                |       |
|----------------|--------|----------------|-------|----------------|-------|
|                | Energy |                | Time  |                | Mb    |
| (c) C          | 1.00   | (c) C          | 1.00  | (c) Pascal     | 1.00  |
| (c) Rust       | 1.03   | (c) Rust       | 1.04  | (c) Go         | 1.05  |
| (c) C++        | 1.34   | (c) C++        | 1.56  | (c) C          | 1.17  |
| (c) Ada        | 1.70   | (c) Ada        | 1.85  | (c) Fortran    | 1.24  |
| (v) Java       | 1.98   | (v) Java       | 1.89  | (c) C++        | 1.34  |
| (c) Pascal     | 2.14   | (c) Chapel     | 2.14  | (c) Ada        | 1.47  |
| (c) Chapel     | 2.18   | (c) Go         | 2.83  | (c) Rust       | 1.54  |
| (v) Lisp       | 2.27   | (c) Pascal     | 3.02  | (v) Lisp       | 1.92  |
| (c) Ocaml      | 2.40   | (c) Ocaml      | 3.09  | (c) Haskell    | 2.45  |
| (c) Fortran    | 2.52   | (v) C#         | 3.14  | (i) PHP        | 2.57  |
| (c) Swift      | 2.79   | (v) Lisp       | 3.40  | (c) Swift      | 2.71  |
| (c) Haskell    | 3.10   | (c) Haskell    | 3.55  | (i) Python     | 2.80  |
| (v) C#         | 3.14   | (c) Swift      | 4.20  | (c) Ocaml      | 2.82  |
| (c) Go         | 3.23   | (c) Fortran    | 4.20  | (v) C#         | 2.85  |
| (i) Dart       | 3.83   | (v) F#         | 6.30  | (i) Hack       | 3.34  |
| (v) F#         | 4.13   | (i) JavaScript | 6.52  | (v) Racket     | 3.52  |
| (i) JavaScript | 4.45   | (i) Dart       | 6.67  | (i) Ruby       | 3.97  |
| (v) Racket     | 7.91   | (v) Racket     | 11.27 | (c) Chapel     | 4.00  |
| (i) TypeScript | 21.50  | (i) Hack       | 26.99 | (v) F#         | 4.25  |
| (i) Hack       | 24.02  | (i) PHP        | 27.64 | (i) JavaScript | 4.59  |
| (i) PHP        | 29.30  | (v) Erlang     | 36.71 | (i) TypeScript | 4.69  |
| (v) Erlang     | 42.23  | (i) Jruby      | 43.44 | (v) Java       | 6.01  |
| (i) Lua        | 45.98  | (i) TypeScript | 46.20 | (i) Perl       | 6.62  |
| (i) Jruby      | 46.54  | (i) Ruby       | 59.34 | (i) Lua        | 6.72  |
| (i) Ruby       | 69.91  | (i) Perl       | 65.79 | (v) Erlang     | 7.20  |
| (i) Python     | 75.88  | (i) Python     | 71.90 | (i) Dart       | 8.64  |
| (i) Perl       | 79.58  | (i) Lua        | 82.91 | (i) Jruby      | 19.84 |

Source : <https://programmation.developpez.com/actu/253829/Programmation-une-etude-revele-les-langages-les-plus-voraces-en-energie-Perl-Python-et-Ruby-en-tete-C-Rust-et-Cplusplus-les-langages-les-plus-verts/>

La suite de ce document est un résumé volontairement incomplet et parfois imprécis, à l'usage du débutant, de ce qu'il faut savoir pour aborder le langage C dans un contexte 'embarqué'.

De nombreux détails sont sciemment omis afin de simplifier les choses. Des choix arbitraires ont notamment été fait concernant certaines notations... Ces choix ont été faits sur la base des « bonnes pratiques » couramment admises par l'état de l'art.

Ponctuellement, quelques liens vers des ressources externes sont présentés en gris, si vous souhaitez en savoir plus où si vous ne comprenez pas ce qui est expliqué.

Vous pouvez vous référer aux ressources suivantes :

[https://c.developpez.com/cours/20-heures/?page=page\\_1](https://c.developpez.com/cours/20-heures/?page=page_1)

<https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c>

<https://zestedesavoir.com/tutoriels/755/le-langage-c-1/>

On ne s'intéresse pas dans ce document aux outils de compilation et d'exécution.

Voici un exemple de programme (pour PC) qui affiche ceci sur un terminal :

Nous recommandons la lecture de ce guide à **tout étudiant en stage, ou tout ingénieur qui développe en langage C.**

Les notions abordées dans les exercices suivants sont listées ici :

## Table des matières

|  |    |
|--|----|
| 1. Fonctions, variables et paramètres .....              | 5  |
| 2. Valeur de retour .....                                | 6  |
| 3. main().....   | 6  |
| 4. Types.....  | 8  |
| 5. Transtypage .....                                     | 9  |
| 6. Portion conditionnelle de code. ....                  | 10 |
| 7. Opérateur ternaire .....                              | 11 |
| 8. Commentaires .....                                    | 12 |
| 9. Débogueur .....                                       | 13 |
| 10. Opérateurs arithmétiques et logiques bit à bit ..... | 14 |
| 11. Définition de tableau .....                          | 16 |
| 12. Boucles for .....                                    | 16 |
| 13. Boucles while .....                                  | 17 |
| 14. Boucles do...while().....                            | 17 |
| 15. Boucles imbriquées .....                             | 19 |
| 16. Chaines de caractères, code ASCII.....               | 20 |
| 17. Enumérations .....                                   | 22 |
| 18. Structures .....                                     | 23 |
| 19. Switch .....   | 24 |
| 20. Module logiciel .....                                | 26 |
| 21. #include "fichier.h" .....                           | 27 |
| 22. Public/Privé ?.....                                  | 28 |
| 23. static .....   | 30 |
| 24. Accesseurs et variables privées .....                | 31 |
| 25. #define .....  | 32 |
| 26. #ifdef et #if .....                                  | 33 |
| 27. printf .....   | 34 |
| 28. Float .....  | 35 |
| 29. Pointeurs.....                                       | 36 |
| 30. Pointeurs sur fonctions.....                         | 38 |
| 31. volatile .....                                       | 39 |
| 32. assert... ..   | 41 |
| 33. fonctions et librairies à connaître.....             | 42 |
| 34. compilateur .....                                    | 43 |
| 35. Allocation mémoire .....                             | 44 |

## 1. Fonctions, variables et paramètres

Le C est un langage séquentiel ; des instructions s'exécutent les unes après les autres.

C'est également un langage 'fonctionnel' : une fonction principale appelle des fonctions qui peuvent appeler à leur tour des fonctions...

Voici une fonction (qui ne fait rien !) :

```
void une_fonction_qui_ne_fait_rien(void)
{
}
```

Rien, c'est tout de même peu... demandons à cette fonction d'effectuer une somme de deux nombres. Il nous faut pour cela introduire des '**variables**'.

-----

```
void function_sum(void)
{
    int a = 4; //création d'un entier a contenant 4
    int b = 2; //création d'un entier b contenant 2
    int s;     //création d'un entier s
    s = a + b; //c reçoit la somme de a et b
}
```

Dans cet exemple, les variables a, b et s sont créées en début de fonction, et détruites après l'appel de la fonction.

On remarque que chaque '**instruction**' se termine par un point-virgule.

Par ailleurs, les données insérées dans a et b sont indiquées 'en dur' dans le code : s vaudra donc toujours 6.

-----

Heureusement, une fonction peut aussi admettre des paramètres qu'elle utilisera dans ses calculs.

```
void function_sum(int a, int b)
{
    int s; //création d'un entier s
    s = a + b; //c reçoit la somme de a et b
}
```

La variable s contiendra la somme des deux nombres '**passés en paramètres**' à la fonction.

Pour appeler cette fonction et lui fournir les paramètres souhaités, il faut écrire :

```
void une_fonction(void)
{
    function_sum(4, 2); //appel de fonction avec paramètres
}
```

La valeur de s sera ici bien calculée selon les paramètres donnés à la fonction, mais cette variable sera détruite à l'issue de l'appel.

## 2. Valeur de retour

Il est possible de récupérer une ‘**valeur de retour**’ fournie par une fonction appelée :

```
//fonction renvoyant un entier
int function_sum(int a, int b)
{
    int s;           //création d'un entier s
    s = a + b;       //c reçoit la somme de a et b
    return s;        //retourner la valeur de la variable s
}
```

```
void une_fonction(void)
{
    int sum;          //création d'un entier sum
    sum = function_sum(4, 2); //sum reçoit la valeur retournée
}
```

-----

Pour pouvoir appeler correctement une fonction, il faut en connaître l'existence (et donc le type de sa valeur de retour ainsi que les nombres et types de ses paramètres).

C'est le ‘**prototype**’ de la fonction qui renseigne celui qui l'appelle de ces informations. Ce prototype doit être défini en dehors de toute fonction, et AVANT tout l'appel...

```
int function_sum(int a, int b); //Prototype de la fonction
```

Généralement, on peut considérer que le prototype est une copie de la première ligne de la fonction, avec un point-virgule à la fin (sans le ‘**contenu**’ délimité par les accolades).

## 3. main()

Si les fonctions s'appellent les unes les autres, il en faut une pour être la première fonction appelée !

Le ‘**compilateur**’ (qui transforme votre code C en un code machine compréhensible par le processeur de la cible sur laquelle vous voulez exécuter le programme) considère que la première fonction appelée se nomme ‘**main**’. Cette fonction ‘principale’ est le point d'entrée du programme.

```
int main(void)
{
    //initialisations

    //boucle de tâche de fond
    while(1)
    {

    }
}
```

Dans l'écosystème Arduino que certains d'entre vous ont pratiqué, la partie initialisation correspond à la fonction nommée setup() ; et la partie boucle de tâche de fond correspond à la fonction loop().

## 4. Types

Arrêtons-nous un instant sur cette question : qu'est-ce qu'un 'int' ?

On peut définir des variables selon plusieurs types. Un type définit notamment la taille du contenant ; c'est-à-dire l'espace mémoire qui est réservé pour la variable que l'on déclare.

Nous voilà face à un joli problème : les types par défaut du langage C n'ont pas toujours la même taille selon le contexte !

A quelques détails près, le C impose seulement :

$$\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$$

On peut donc avoir des différences selon les microcontrôleurs (selon leurs compilateurs), par exemple :

| Type     | Taille (bits) |             |
|----------|---------------|-------------|
|          | Cible PIC18   | Cible STM32 |
| char     | 8             | 8           |
| short    | 16            | 16          |
| int      | <b>16</b>     | <b>32</b>   |
| long int | <b>32</b>     | <b>64</b>   |

En d'autres termes, on peut avoir un code qui se comporte différemment selon la cible pour laquelle il a été compilé !

Pour éviter cela, il est vivement conseillé de ne pas utiliser les types short, int et long int... mais de privilégier des équivalences dont la taille est maîtrisée :

| Type     | Taille (bits) | Signé     | Intervalle  |
|----------|---------------|-----------|---|
| uint8_t  | 8             | non signé | $[0 \dots 2^8 - 1]$ = $[0 \dots 255]$   |
| int8_t   | 8             | signé     | $[-2^7 \dots 2^7 - 1]$ = $[-128 \dots 127]$   |
| uint16_t | 16            | non signé | $[0 \dots 2^{16} - 1]$ = $[0 \dots 65535]$  |
| int16_t  | 16            | signé     | $[-2^{15} \dots 2^{15} - 1]$ = $[-32768 \dots 32767]$   |
| uint32_t | 32            | non signé | $[0 \dots 2^{32} - 1]$ $\approx$ $[0 \dots 4,3 \text{ milliards}]$                            |
| int32_t  | 32            | signé     | $[-2^{31} \dots 2^{31} - 1]$ $\approx$ $[-2,1 \text{ milliards} \dots 2,1 \text{ milliards}]$ |
| uint64_t | 64            | non signé | $[0 \dots 2^{64} - 1]$ $\approx$ $[0 \dots 1,8 * 10^{19}]$                                    |
| int64_t  | 64            | signé     | $[-2^{63} \dots 2^{63} - 1]$ $\approx$ $[-9,2 * 10^{18} \dots 9,2 * 10^{18}]$                 |

Pour pouvoir utiliser ces types, il suffit d'inclure `<stdint.h>` au début de votre fichier :

```
#include <stdint.h>
```

**A partir de maintenant, le type int sera banni de vos codes<sup>1</sup> !**

A chaque fois que vous faite un calcul avec une variable, vous devez vous poser la question des limites de votre calcul. (Risque-t-il de déborder ? Quelle valeur maximale puis-je obtenir ? ...)

*Des fusées ont pété pour moins que ça. Des avions se sont retournés.*

---

<sup>1</sup> Et de ce document... le but étant de bien maîtriser la taille de chaque variable manipulée.

## 5. Transtypage

On peut forcer affecter une donnée d'un type dans une variable d'un autre type.  
On parle alors de transtypage.

Quelques exemples :

```
uint8_t a = 250;
float b = 12.5;
uint32_t c = 1000000000;
float d = 1000.5;

//relativement indolore : on range une petite variable dans une grande case
c = (uint32_t)(a);

//explosif : on tronque une grande variable (équivalent ici à un modulo 256)
a = (uint8_t)(c);

//le nombre décimal stocké dans b sera arrondi à l'entier inférieur
c = (uint32_t)(b);

//deux étapes dans cet exemple ; où la partie entière de d est
temporairement écrasée (modulo 256) sur un entier 8 bits ; avant d'être
automatiquement transtypé en 32 bits dans c.
le nombre obtenu est 1000%256= 232
c = (uint8_t)(d);
```

Généralement, le compilateur lève un warning lorsque la case de destination est plus petite que la case du nombre source. Le fait de transtyper explicitement montre que l'on assume cela ; ce qui supprime le warning.

L'inverse (un petit nombre dans une grande case) ne provoque pas de warning.



## 6. Portion conditionnelle de code.

Lorsque l'on réalise des programmes, on a très vite besoin de conditionner l'exécution de certaines instructions. C'est le rôle du mot clé 'if' et des parenthèses qui le suivent.

Dans ce premier exemple, on voit un bloc exécuté si  $a > b$ , et un bloc exécuté dans le cas contraire.

```
//renvoie le maximum de deux nombres
uint8_t max(uint8_t a, uint8_t b)
{
    if(a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

Dans le cas où il n'y a qu'une seule instruction dans le bloc, on peut omettre les accolades.

```
//renvoie la valeur absolue
int16_t absolute(int16_t a)
{
    if(a < 0)
        return -a;
    else
        return a;
}
```

Attention, les indentations n'ont aucun pouvoir en C (par opposition au Python) :

Ce code là est (très) fourbe (en plus d'être inutile) :

```
//Cette fonction complètement stupide renvoie le maximum entre deux nombres, ou 0
si reset vaut 1
int32_t max_with_reset(int32_t a, int32_t b, int8_t reset)
{
    if(reset == 1)
        a = 0;
        b = 0;           //attention, b = 0 pas dans le if() même si son
//indentation pourrait le laisser croire
    return max(a, b);
}
```

Il serait **syntactiquement correct** d'écrire :

```
int32_t max_with_reset(int32_t a, int32_t b, int8_t reset)
{
    if(reset == 1){ a = 0; b = 0; }return max(a, b);
}
```

Mais c'est quand même très moche, et peu lisible !

⇒ L'indentation n'est pas syntaxiquement obligatoire...

**Mais on s'oblige à écrire un code indenté et lisible !**

## 7. Opérateur ternaire

Il existe une forme compacte équivalente au `if()` que l'on nomme « opérateur ternaire ».

Sa syntaxe est la suivante :

```
v = (condition)?(valeur_si_vraie):(valeur_si_fausse);
```

En voici un exemple.

```
max = (a>b)?a:b;

//équivalent à :
if(a>b)
    max = a;
else
    max = b;
```

Les parenthèses sont conseillées autour des valeurs si elles contiennent des calculs (notamment pour éviter des problèmes avec les priorités des opérateurs).

## 8. Commentaires

Un code bien construit (quel que soit le langage !) est surtout un code bien commenté !

Il existe deux façon d'insérer des commentaires en C :

```
/*  
    La méthode multilignes...  
*/  
  
int32_t abc;          //la méthode "fin de ligne". Du code peut se placer avant.  
  
//La mise en commentaire d'une ligne de code permet de la "désactiver" sans la  
supprimer complètement :  
//abc = 4;
```

Il est d'usage de commenter toute ligne présentant une difficulté particulière... afin d'aider le lecteur (parfois soi-même plus tard) à comprendre le code.

Le standard doxygen (<https://fr.wikipedia.org/wiki/Doxygen>) tend à s'imposer dans de nombreux contextes. Ce format standardisé permet notamment la génération automatique de documentation d'après les commentaires insérés directement dans le code.

Regardez les commentaires associés à cette fonction.

On y trouve :

- `/**` : le début d'un « commentaire au format doxygen » (avec deux étoiles)
- `@brief` : brève description de la fonction
- `@param` : description des paramètres à fournir
- `@retval` : description de la donnée retournée par la fonction
- `@pre` : précondition à respecter pour que la fonction fasse son job
- `@post` : postcondition que garanti la fonction

```
/**  
 * @brief Reads the specified input port pin.  
 * @param GPIOx: where x can be (A..G depending on device used) to select the GPIO  
 * @param GPIO_Pin: port bit to read : GPIO_PIN_x where x can be (0..15).  
 * @pre the corresponding GPIO must be initialized before calling this function  
 * @retval The input port pin value.  
 */  
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)  
{  
    GPIO_PinState bitstatus;  
  
    //masquage ET entre le registre IDR du GPIO demandé et la broche voulue  
    if ((GPIOx->IDR & GPIO_Pin) != (uint32_t)GPIO_PIN_RESET)  
    {  
        bitstatus = GPIO_PIN_SET;  
    }  
    else  
    {  
        bitstatus = GPIO_PIN_RESET;  
    }  
    return bitstatus;  
}
```

## 9. Débogueur

*« Le débogueur ? J'en ai pas besoin moi monsieur, je fais pas de bogue »  
Etudiant anonyme – paix à son âme.*

*Vous connaissez la différence entre un bon et un mauvais développeur... ?*

- *Un mauvais développeur croit qu'il doit construire des programmes sans bogues...*
- *Un bon développeur a compris qu'il doit être capable de trouver les bogues qui restent dans son programme.*

Le débogueur n'enlève pas de bogues.

Le débogueur est l'un des outils permettant de :

- Analyser l'exécution du programme, pas à pas
- Observer l'état des variables, de la mémoire, des registres du processeur, des périphériques...
- Arrêter un programme à un endroit donné (breakpoint)

Il faut pouvoir jongler dans le code en plaçant des breakpoints, en lançant le code, en modifiant si besoin le contenu des variables, ...

## 10. Opérateurs arithmétiques et logiques bit à bit

Voici une liste des opérateurs ‘arithmétiques’ et ‘logiques bit à bit’ en C :

|    |                       |
|----|-----------------------|
| +  | addition              |
| -  | soustraction          |
| *  | multiplication        |
| /  | division              |
| %  | Modulo algébrique     |
|    | ou logique bit à bit  |
| &  | et logique bit à bit  |
| ^  | ou exclusif bit à bit |
| ~  | non bit à bit         |
| << | décalage à gauche     |
| >> | décalage à droite     |

| Raccourcis          | Equivalent               |
|---------------------|--------------------------|
| <b>i++;</b>         | <b>i = i+1;</b>          |
| <b>i--;</b>         | <b>i = i-1;</b>          |
| <b>i+=4;</b>        | <b>i = i + 4;</b>        |
| <b>i-=4;</b>        | <b>i = i - 4;</b>        |
| <b>i*=4;</b>        | <b>i = i * 4;</b>        |
| <b>i/=4;</b>        | <b>i = i / 4;</b>        |
| <b>i%=4;</b>        | <b>i = i % 4;</b>        |
| <b>i&lt;&lt;=4;</b> | <b>i = i &lt;&lt; 4;</b> |
| <b>i&gt;&gt;=4;</b> | <b>i = i &gt;&gt; 4;</b> |
| <b>i&amp;=4;</b>    | <b>i = i &amp; 4;</b>    |
| <b>i^=4;</b>        | <b>i = i ^ 4;</b>        |

Certains de ces opérateurs nécessitent quelques exemples :

|          |   |   |
|----------|---|---|
| tab[i++] | Accède à tab[i], puis incrémente i  | Compte tenu du risque d’ambiguïté, ces notations sont déconseillées |
| tab[++i] | Incrémente i, puis accède à i+1   |   |
| (-4)%3   | -1 (donc ce n’est pas tout à fait un modulo, mais bien le reste de la division entière) |   |
| 42/0     | Mauvaise idée... → levée d’une interruption, plantage du microcontrôleur... !           |   |

### Vrai/Faux

Une variable est considérée comme fausse si elle vaut 0.

Elle est vraie sinon.

‘Et pis c’est tout’...

### Opérateurs logiques

|    |             |
|----|-------------|
|    | ou logique  |
| && | et logique  |
| !  | non logique |

### Opérateurs de comparaisons

|    |                   |
|----|-------------------|
| >  | supérieur         |
| <  | inférieur         |
| >= | supérieur ou égal |
| <= | inférieur ou égal |
| == | égal              |
| != | différent         |

```
//Attention, il ne faut pas confondre l'opérateur d'affectation = avec
l'opérateur de comparaison d'égalité ==
int16_t a = 12;           //a vaut 12
if(a = 13)                 //a vaut-il 13 ?
    printf("perdu !\n");   //bah mince... !
```

Dans cet exemple qui est une « erreur courante du débutant », on ne teste pas si a vaut 13 : on affecte la valeur 13 à la variable a. Le **if** s’applique alors au résultat de cette affectation (qui en c vaut la valeur affectée elle-même, soit ici 13, qui est vrai car non nul).

## 11. Définition de tableau

Très vite naît le besoin de regrouper des informations similaires en tableau. Un tableau est une suite d'informations rangées consécutivement dans la mémoire.

Par exemple : les notes des élèves d'une classe... Ou les caractères d'une chaîne.

On utilise alors la notation suivante : `type nom_du_tableau[taille];`

```
int32_t tableau[15];    //création d'un tableau de 15 entiers (de 0 à 14 !)  
  
tableau[0] = 123; //on remplit la première case du tableau  
tableau[1] = 456; //on remplit la deuxième case du tableau  
//...  
tableau[14] = 4;  //on remplit la dernière case du tableau  
  
int8_t i = 5;  
tableau[i] = 98;  //cela marche aussi avec un index variable  
  
int32_t tableau_initialise[15] = {5, 6, 4, 1, 2, 0, 9};  
//création d'un tableau de 15 cases initialisées  
//on peut indiquer au plus un nombre de valeurs correspondant à la taille du  
tableau. Si on en indique moins, la suite est complétée par des zéros.
```

## 12. Boucles for

Nul besoin de vous faire un dessin. Remplir un tableau de 1000 cases risque d'être fastidieux s'il faut écrire 1000 lignes...

On utilise :

```
for(instruction initiale ; condition ; action de fin de boucle)  
{  
    //corps de la boucle  
}
```

L'instruction 'for' permet une itération de la boucle limitée par les accolades tant que la 'condition' est vraie. L'instruction initiale est réalisée une fois au début. L'action de fin de boucle est réalisée **à la fin de chaque boucle**.

Dans cet exemple, on initialise les 1000 cases du tableau.

```
int32_t t[1000];  
int16_t i;  
for(i = 0; i<1000; i++)  
{  
    t[i] = 0;  //initialisation à 0 de toutes les cases du tableau  
}
```

Attention, deux 'erreurs graves du débutant' se cachent dans ce code (à vous de les trouver !) :

```
int32_t t[1000];  
int8_t i;  
for(i = 0; i<=1000; i++)  
    t[i] = 0;
```

### 13. Boucles while

A peu de choses près, une boucle while est similaire à la boucle for. (En tout cas, on peut toujours syntaxiquement utiliser l'une ou l'autre.)

Dans la pratique, on préfère les utiliser ainsi :

- **for** est plutôt utilisé lorsqu'on connaît le nombre d'itération de la boucle
- **while** est plutôt utilisé lorsqu'on ne connaît pas le nombre d'itération de la boucle

Syntaxe :

```
while(condition)
{
    //corps de la boucle
}
```

```
int32_t t[1000];
int16_t i;
i = 0;
while(i<1000)
{
    t[i] = 0;
    i++;
}
```

Parfois, une boucle for ou while ne présente aucun contenu, on peut alors se contenter d'un point-virgule pour clore l'instruction à la place des accolades :

```
while(!condition);

for(i = 0; i<=1000; i++);

for(;;);
```

Les notations **for(;;)** et **while(1)** sont des boucles infinies... donc on ne sortira jamais... à moins que...

- **break;** permet de sortir d'une boucle avant la fin

### 14. Boucles do...while()

A la différence de la boucle while qui n'exécute pas la boucle si la condition est fausse dès le début ; la boucle do...while() exécute au moins une fois la boucle avant de consulter la condition.

```
do{
    //corps de la boucle
}while(condition); //sans oublier le ;
```

## 15. Boucles imbriquées

Il est bien sûr possible d'imbriquer plusieurs boucles.

Par exemple pour ce tableau à doubles dimensions où l'affectation dans ce tableau sera exécutée 2000 fois (100\*20) !

```
int32_t t[100][20];
int16_t i;
int16_t j;
for(i = 0; i<100; i++)
{
    for(j = 0; j<20; j++)
    {
        t[i][j] = i*j;
    }
}
```



## 16. Chaines de caractères, code ASCII

*S'il n'y avait pas d'humains, la vie n'en serait que plus simple pour les machines... Ces primates compliquent nettement les choses avec leur façon de parler : des mots, des phrases, quelle idée !*

Afin de stocker de façon simple et pratique des informations textuelles, on utilise :

- une table de conversion (ASCII) dans laquelle chaque caractère est associé à un nombre
- un tableau de caractères (i.e. un tableau de nombres !) pour les rassembler en « chaîne ».

Dans la table ASCII, on utilise des entiers sur 8 bits (nombres de 0 à 255).

En C, tout caractère est noté entre quotes : 'x'.

Une chaîne de plusieurs caractères peut se noter en guillemets : "chaîne".

On remarque que les chiffres '0' à '9' se suivent dans l'ordre, ce qui permet de passer facilement de '4' à 4 en soustrayant '0'.

De la même façon, les lettres se suivent... ce qui permet par exemple de comparer deux mots dans l'ordre alphabétiques.

Attention à ne pas confondre :

**Le caractère nul :**

'\0' = 0 = 0x00

avec :

**Le caractère zéro :**

'0' = 48 = 0x30

**Par convention, une fin de chaîne se détermine par un caractère nul.**

**Il faut donc prévoir sa place en mémoire : la chaîne "toto" occupe 5 octets en mémoire !**

| Nombre |    | Car | Code en base |    | Car | Code en base |    | Car | Code en base |    | Car |
|--------|----|-----|--------------|----|-----|--------------|----|-----|--------------|----|-----|
| 10     | 16 |     | 10           | 16 |     | 10           | 16 |     | 10           | 16 |     |
| 0      | 0  | NUL | 32           | 20 | SP  | 64           | 40 | @   | 96           | 60 | `   |
| 1      | 1  | SOH | 33           | 21 | !   | 65           | 41 | A   | 97           | 61 | a   |
| 2      | 2  | STX | 34           | 22 | "   | 66           | 42 | B   | 98           | 62 | b   |
| 3      | 3  | ETX | 35           | 23 | #   | 67           | 43 | C   | 99           | 63 | c   |
| 4      | 4  | EOT | 36           | 24 | \$  | 68           | 44 | D   | 100          | 64 | d   |
| 5      | 5  | ENQ | 37           | 25 | %   | 69           | 45 | E   | 101          | 65 | e   |
| 6      | 6  | ACK | 38           | 26 | &   | 70           | 46 | F   | 102          | 66 | f   |
| 7      | 7  | BEL | 39           | 27 | '   | 71           | 47 | G   | 103          | 67 | g   |
| 8      | 8  | BS  | 40           | 28 | (   | 72           | 48 | H   | 104          | 68 | h   |
| 9      | 9  | HT  | 41           | 29 | )   | 73           | 49 | I   | 105          | 69 | i   |
| 10     | 0A | LF  | 42           | 2A | *   | 74           | 4A | J   | 106          | 6A | j   |
| 11     | 0B | VT  | 43           | 2B | +   | 75           | 4B | K   | 107          | 6B | k   |
| 12     | 0C | FF  | 44           | 2C | ,   | 76           | 4C | L   | 108          | 6C | l   |
| 13     | 0D | CR  | 45           | 2D | -   | 77           | 4D | M   | 109          | 6D | m   |
| 14     | 0E | SO  | 46           | 2E | .   | 78           | 4E | N   | 110          | 6E | n   |
| 15     | 0F | SI  | 47           | 2F | /   | 79           | 4F | O   | 111          | 6F | o   |
| 16     | 10 | DLE | 48           | 30 | 0   | 80           | 50 | P   | 112          | 70 | p   |
| 17     | 11 | DC1 | 49           | 31 | 1   | 81           | 51 | Q   | 113          | 71 | q   |
| 18     | 12 | DC2 | 50           | 32 | 2   | 82           | 52 | R   | 114          | 72 | r   |
| 19     | 13 | DC3 | 51           | 33 | 3   | 83           | 53 | S   | 115          | 73 | s   |
| 20     | 14 | DC4 | 52           | 34 | 4   | 84           | 54 | T   | 116          | 74 | t   |
| 21     | 15 | NAK | 53           | 35 | 5   | 85           | 55 | U   | 117          | 75 | u   |
| 22     | 16 | SYN | 54           | 36 | 6   | 86           | 56 | V   | 118          | 76 | v   |
| 23     | 17 | ETB | 55           | 37 | 7   | 87           | 57 | W   | 119          | 77 | w   |
| 24     | 18 | CAN | 56           | 38 | 8   | 88           | 58 | X   | 120          | 78 | x   |
| 25     | 19 | EM  | 57           | 39 | 9   | 89           | 59 | Y   | 121          | 79 | y   |
| 26     | 1A | SUB | 58           | 3A | :   | 90           | 5A | Z   | 122          | 7A | z   |
| 27     | 1B | ESC | 59           | 3B | ;   | 91           | 5B | [   | 123          | 7B | {   |
| 28     | 1C | FS  | 60           | 3C | <   | 92           | 5C | \   | 124          | 7C |     |
| 29     | 1D | GS  | 61           | 3D | =   | 93           | 5D | ]   | 125          | 7D | }   |
| 30     | 1E | RS  | 62           | 3E | >   | 94           | 5E | ^   | 126          | 7E | ~   |
| 31     | 1F | US  | 63           | 3F | ?   | 95           | 5F | _   | 127          | 7F | DEL |

```
//Par convention, chacune de ces chaines se termine par un caractère nul
//Nous verrons plus tard le rôle de l'étoile : *
char * chaine1 = "ceci est une chaine de caractères";
char * chaine2 = "voici un saut de ligne : \n";
char * chaine3 = "voici un saut de ligne avec retour chariot : \r\n";
char * chaine4 = "voici un antislash \\ un guillemet \" et une tabulation : \t";
char * chaine5 = "exemple"; //cette chaine occupe 8 octets (7 lettres + 1 zéro)
char chainevariable[20]; //chaine variable de 20 caractères, non initialisée
```

Attention à ne pas confondre une chaîne de caractères avec un chêne de caractère :



<https://t.co/dfiX5zLK4K>

## 17. Enumérations

Afin d'organiser les informations stockées, il devient vite utile de faire appel aux concepts de structures, d'énumération et d'union.

Une énumération est une suite de mots auxquels on associe des nombres constants.

Le principal objectif est d'utiliser dans le code **des mots explicites** plutôt que des numéros neutres et sans saveur.

```
typedef enum{
    COLOR_BLACK = 0,           //le premier élément vaut 0 (on peut l'indiquer)
    COLOR_BLUE,                //chaque élément porte le nombre suivant...ici 1
    COLOR_GREEN,               //le mot COLOR_GREEN équivaut à 2
    COLOR_CYAN,                //...
    COLOR_RED,
    COLOR_PURPLE,
    COLOR_YELLOW,
    COLOR_WHITE                //COLOR_WHITE équivaut à 7
}color_e;

color_e background_color = COLOR_CYAN;    //on utilise le type color_e...
color_e beach_color = COLOR_YELLOW;
color_e see_color = COLOR_BLUE;
```

Comparez la lisibilité de ces deux appels de fonctions, avec et sans utilisation d'énumérations.

```
//avec énumérations
try_going(1350, COLOR_Y(100), state, NEXT_STATE, GET_OUT, FAST, FORWARD,
NO_DODGE_AND_WAIT, END_AT_BRAKE);

//sans énumérations
try_going(1350, COLOR_Y(100), state, 8, 15, 0, 2, 2, 1);
```

## 18. Structures

Une structure est un type contenant plusieurs champs. Les informations s'y trouvant sont alors regroupées.

Pour accéder aux champs d'une structure qui nous appartient, on utilise le point : '.'

Pour accéder aux champs d'une structure dont on ne connaît qu'un pointeur, on utilise la flèche : '->'

```
//définition d'une structure point_t
typedef struct{
    int32_t x;
    int32_t y;
    color_e color;
    char nom[20];
}point_t;           //par convention, on suffixe le nom des types avec _t

point_t p1;          //déclaration d'un point initialisé
p1.x = 120;          //remplissage du champ x
p1.y = 45;           //remplissage du champ y
p1.color = COLOR_GREEN; //remplissage du champ color
p1.nom[0] = 'p';      //remplissage du champ nom
p1.nom[1] = 'l';
p1.nom[2] = '\0';

//déclaration d'un point initialisé
point_t p2 = (point_t){28, 42, COLOR_PURPLE, {'p','2','\0'}};

point_t * pointeur_vers_p; //création d'un pointeur vers une structure
pointeur_vers_p = &p1;     //le pointeur vaut "l'adresse de p1"
pointeur_vers_p->x = 140 ;  //accès par adresse au champ de la structure

//appel de fonction et passage de structure par pointeur
fonction(&p1);             //très économique en temps d'exécution (1 adresse transmise)
```

## 19. Switch

Quand on a la flemme d'écrire ceci :

```
if(variable == 0)
{
    led_write(LED_1, 1);
}
else if(variable == 1)
{
    p1.x = 20;
}
else if(variable == 2)
{
    p2.y++;
}
else if(variable == 3 || variable == 4)
{
    beach_color = COLOR_BLACK;
}
else
{
    led_write(LED_1, 0);
}
```

Alors on peut écrire ceci :

```
switch(variable)
{
    case 0:    //ce cas sera exécuté si la variable vaut 0
        led_write(LED_1, 1);
        break;
    case 1:
        p1.x = 20;
        break;
    case 2:
        p2.y++;
        break;
    case 3:
        //no break;    --> must be stuck with case 4
        //un commentaire fait explicitement apparaître qu'on a choisi de ne
        pas vouloir de break, ce 'case 3' équivaudra donc au 'case 4'.
    case 4:
        beach_color = COLOR_BLACK; //ligne exécutée si variable vaut 3 ou 4
        break;
    default:
        led_write(LED_1, 0);
        break;
}
```

Attention à ne pas oublier les '**break**'. Sans cela, on continue l'exécution sur le **case** suivant !  
Et on vous jure que c'est vachement mieux.

## 20. Module logiciel

**1 module logiciel = 1 fichier source et son header**

Dès que le programme s'enrichit et que l'on fait apparaître de nombreuses fonctions, il n'est plus possible de se contenter de tout placer dans le fichier `main.c` : il est nécessaire d'organiser ces fonctions et de les répartir en plusieurs fichiers.

Les objectifs de cette structuration sont multiples :

- y voir plus clair ! (un fichier de 5000 lignes avec 200 fonctions, c'est mal)
- permettre la réutilisabilité des fonctions créées (dans d'autres projets par exemple)
- favoriser la durée de recompilation (seuls les fichiers modifiés sont recompilés)
- lisibilité++ : un module logiciel est un ensemble cohérent qui dispose de variables et de fonctionnalités. Son périmètre est clairement défini. Parfois en dehors de tout contexte.

L'ensemble des briques proposées dans la librairie que nous mettons à votre disposition ont été conçues de façon génériques en dehors d'un contexte précis. On peut les utiliser dans de nombreux projets.

### *Que met-t-on dans un header ?*

D'une façon générale : « **le header contient ce qui est public** ». On n'y trouve pas de 'contenu de fonction', mais bien uniquement leurs prototypes. (Et ce qui est nécessaire pour les comprendre).

```
/*    gpio.h                                //nom du fichier
 *    Created on: 21 july 1969              //date de création du fichier
 *    Author: Niel Armstrong                //auteur
 */
#ifndef GPIO_H_ //ces lignes permettent d'éviter les inclusions récursives.
#define GPIO_H_ // voir l'encadré spécifique à ce sujet

//inclusions nécessaire à l'interprétation des prototypes publics ci-dessous
#include "stm32f1xx.h"
#include "macro_types.h"

//Types publics (énumération, structures...)
typedef enum
{
    GPIO_DIRECTION_INPUT,
    GPIO_DIRECTION_OUTPUT
}GPIO_direction_e;

typedef struct
{
    uint8_t port;
    uint8_t pin;
    bool_e state;
    GPIO_direction_e direction;
}GPIO_port_t;

//Prototypes des fonctions publiques
void GPIO_test(void);
void GPIO_init(GPIO_port_t gpio);
void GPIO_set(GPIO_port_t gpio, bool_e state);
bool_e GPIO_get(GPIO_port_t gpio);

#endif /* GPIO_H_ */
```

## 21. `#include "fichier.h"`

Pour pouvoir utiliser les fonctions public d'un module depuis un autre fichier, il faut « inclure » son header.

La pseudo-instruction `#include` est équivalente à une sorte de « copier-coller ».

Quelques règles :

- Lorsque l'on inclue un header issu d'une librairie externe, on utilise cette notation :
  - `#include <stdlib.h>`
- Lorsque l'on inclue un header issu de nos sources, on utilise cette notation :
  - `#include "f1.h"`
- Attention : **on n'inclue jamais un .c** :
  - Rappelons-nous que seuls les .c sont compilés → cela revient à dupliquer du code !
- On doit se contenter d'inclure seulement les headers dont on a besoin...
  - Il est inutile de lister tous les .h et de les inclure partout
- Dans un header : on inclut d'éventuels autres header définissant notamment des types utilisées dans les prototypes publics qui suivent.
- On évite les inclusions récursives (cf ci-dessous)

On remarque les lignes suivantes qui évitent les inclusions récursives infinies des fichiers header lors de la compilation :

```
#ifndef F1_H_      //si ceci n'est pas déjà défini
#define F1_H_      //Je le défini
                  //Et ce contenu est compilé

#endif /* F1_H_ */
```

Exemple de boucle infinie lors de la compilation de `f1.c`

(`f1.c` inclut `f1.h` qui inclut `f2.h` qui inclut `f1.h` qui inclut `f2.h` qui inclut `f1.h` qui inclut `f2.h` qui inclut `f1.h` qui inclut `f2.h`...)

*f1.c*

```
#include "f1.h"
//...
```

*f1.h*

```
#include "f2.h"
//...
```

*f2.h*

```
#include "f1.h"
//...
```

## 22. Public/Privé ?

Dans de nombreux langages, on s'impose de rendre privé ce qui ne doit pas être public.

Si vous avez créé une fonction qui s'appelle `affiche()` ; et qu'une fonction portant le même nom existe déjà dans un autre code source d'une librairie dont vous avez besoin... l'éditeur des liens (appelé lors de la compilation) est face à un conflit gênant ! Si ces fonctions ont pour seul but d'être appelées « en local » seulement dans les fichiers où elles sont définies, alors le fait de les rendre privées résoudra le conflit.

Le mot clé '**static**' est utilisé pour rendre privée une fonction où une variable.

```
                                //CRYPTUART.h
#ifndef CRYPTUART_H_
#define CRYPTUART_H_
#include <stdint.h>
//Fonctions publiques :
// on connaît leur existence depuis l'extérieur
void CRYPTUART_init(void);
void CRYPTUART_puts(char * chaine);
uint8_t CRYPTUART_gets(char * chaine, uint8_t max_size);

#endif /* CRYPTUART_H_ */
```

```
                                //CRYPTUART.c
#include CRYPTUART.h    //inclusion du header

//prototypes des fonctions privée
static void CRYPTUART_encrypt(char * encrypted_string, char * clear_string);
static void CRYPTUART_decrypt(char * clear_string, char * encrypted_string);
static void CRYPTUART_putc(char c);

//////////Fonctions publiques//////////
// on connaît leur existence depuis l'extérieur

void CRYPTUART_init(void){
//contenu de la fonction qui initialise ce module logiciel...
}

void CRYPTUART_puts(char * chaine){
//contenu de la fonction qui envoie une chaîne après l'avoir chiffrée
}

uint8_t CRYPTUART_gets(char * chaine, uint8_t max_size){
//contenu de la fonction qui récupère une chaîne, la déchiffre et la retourne
}

//////////Fonctions privées//////////
// appelées seulement depuis ce fichier
static void CRYPTUART_encrypt(char * encrypted_string, char * clear_string){
//contenu de la fonction qui chiffre une chaîne
}

static void CRYPTUART_decrypt(char * clear_string, char * encrypted_string){
//contenu de la fonction qui déchiffre une chaîne
}

static void CRYPTUART_putc(char c){
//contenu de la fonction envoie un caractère
}
```



## 23. static

En voilà un beau mot clé...

Le mot clé 'static' s'applique sur une fonction ou une variable pour lui conférer un caractère « **privé** » et « **immobile dans la mémoire** » (cela ne veut pas dire constant !).

Appliquons sur des exemples pour mieux comprendre :

```
//cette variable est accessible dans ce fichier seulement
//elle est initialisée à 1 avant même l'exécution du programme
static uint8_t variable_privée = 1;

//cette variable est accessible dans ce fichier, ou à partir des autres fichiers
s'ils la définissent avec 'extern'
uint8_t variable_publicue;

//fait référence à une variable publique d'un autre fichier !
//Attention, cela ne déclare pas une variable, mais seulement le fait qu'elle
est sensée exister ailleurs.
//ce concept de variable globale est une mauvaise pratique. Il est conseillé
d'utiliser des accesseurs. (voir le chapitre concerné)
extern uint8_t variable_publicue_declaree_ailleurs;

//cette fonction est privée ; son prototype sera recopié au début du fichier .c
//elle sera accessible seulement depuis ce fichier !
static void fonction_privée(void)
{
    //...
}

//cette fonction est publique ; son prototype doit être recopié dans le .h
//elle est accessible depuis les autres fichiers
void fonction_publicue(void)
{
    //cette variable est recréée à chaque appel de la fonction
    //elle est supprimée en fin de fonction !
    //elle vaut donc 4 à chaque appel de cette fonction !
    //elle est accessible dans cette fonction
    uint8_t variable_locale_temporaire = 4;

    //cette variable existe en permanence au même endroit de la mémoire.
    //elle est donc initialisée une seule fois à 5, avant tout appel !
    //elle garde donc sa valeur d'un appel à l'autre de la fonction
    //elle est accessible dans cette fonction
    static uint8_t variable_locale_permanente = 5;

    //...
}
```

Rien ne sert de rendre publique une fonction ou une variable qui n'a pas besoin de l'être !

Rien ne sert de déclarer une variable en début de fichier si seule une fonction en a besoin !

## 24. Accesseurs et variables privées

Comment accéder aux variables appartenant à un autre module logiciel ?

Il existe une mauvaise méthode que nous ne présenterons pas : la variable globale. C'est peu propre. Nous recommandons plutôt cette méthode dont l'exemple est tiré d'une fonctionnalité implémentée sur les robots du club de robotique de l'ESEO.

Si vous inversez les cartes des deux robots, sans les reprogrammer, le fonctionnement est identique.

Cela est rendu possible parce que chaque carte identifie au démarrage le robot sur lequel elle se trouve. (en lisant simplement l'état d'une broche d'entrée imposé par une résistance soudée différemment).

```
#ifndef WHO_AM_I_H_
#define WHO_AM_I_H_

typedef enum
{
    BIG_ROBOT,
    SMALL_ROBOT
}robot_e;

void WHO_AM_I_init(void);

robot_e WHO_AM_I_get(void);

void WHO_AM_I_set(robot_e robot);

#endif /* WHO_AM_I_H_ */
```

```
#include "WHO_AM_I.h"

//variable privée !
static robot_e i_am = BIG_ROBOT;    //valeur par défaut au reset

void WHO_AM_I_init(void)
{
    //lecture de l'état de la broche et remplissage de la variable locale i_am
    if(GPIO_read(WHO_AM_I_PORT, WHO_AM_I_PIN) == 1)
        i_am = BIG_ROBOT;
    else
        i_am = SMALL_ROBOT;
}

robot_e WHO_AM_I_get(void)
{
    return i_am; //fournit une copie du contenu de la variable locale i_am
}

void WHO_AM_I_set(robot_e new_robot)
{
    i_am = new_robot;
    //modification de la variable locale i_am (pour la simulation par exemple)
}
```

Les fonctions xxx\_get() et xxx\_set() se nomment des accesseurs. Elles fournissent des accès en lecture ou en écriture aux variables privées du module logiciel concerné.

## 25. #define

Dans certains projets, il est utile de pouvoir configurer le code selon certaines constantes.

Par exemple, vous avez conçu une carte électronique pouvant accueillir de 1 à 5 capteurs, et vous voulez un code différent selon le nombre de capteurs :

```
#include <stdbool.h>
#define NB_OF_SENSORS 4

#if NB_OF_SENSORS > 5
    #error "Il ne peut y avoir plus de 5 capteurs"
#endif

_Bool sensor_initialized[NB_OF_SENSORS];

void SENSORS_init(void)
{
    uint8_t n;
    for(n=0; n < NB_OF_SENSORS; n++)
        sensor_initialized[n] = true;
}
```

La macro #define permet de définir une équivalence symbolique entre un ‘mot’ et une valeur.

Cela n’a rien à voir avec la définition d’une variable (qui aura une existence en mémoire). Cette équivalence est considérée au début de la compilation (phase dite de ‘préprocesseur’).

Dans l’exemple ci-dessus, on évite de devoir modifier de très nombreuses lignes à chaque fois que l’on change le nombre de capteurs !

Cette méthode est très utile pour les tailles de tableaux, qui sont généralement reprises dans chaque boucle parcourant le tableau !

## 26. #ifdef et #if

Parfois, on veut aller plus loin, et activer certaines portions de code selon des modes différents :

```
//Décommenter cette ligne permet d'activer le mode 'fake sensors'.
//La mettre en commentaire permet de désactiver ce mode.
#define MODE_FAKE_SENSORS

uint16_t SENSORS_get_value(sensor_id_e id)
{
    #ifdef MODE_FAKE_SENSORS
        return 1234;
    #else
        return sensor_value[id];
    #endif
}
```

Une variante en affectant 1 ou 0 à la macro et en utilisant #if

```
//mettre 1 pour activer le mode 'fake sensors' ; 0 pour le désactiver.
#define MODE_FAKE_SENSORS 1

uint16_t SENSORS_get_value(sensor_id_e id)
{
    #if MODE_FAKE_SENSORS
        return 1234;
    #else
        return sensor_value[id];
    #endif
}
```

Sur la plupart des IDE modernes (Eclipse...), la portion de code ‘désactivée’ est grisée. C’est comme si elle n’existait pas.

Cette fonctionnalité est notamment utilisée pour activer/désactiver les modules logiciels que nous vous proposons dans le cadre du projet de système embarqué. Un fichier ‘config.h’ recense un ensemble de macro USE\_xxx où chaque ‘xxx’ concerne un module logiciel que l’on peut activer.

## 27. printf

Une liaison série ou un écran sont des moyens permettant à notre programme de sortir des informations ; soit pour l'utilisateur final ; soit pour le développeur.

Sortir des informations de notre programme est l'un des moyens très utiles pour déboguer.

Dès lors qu'on dispose d'une fonction permettant de sortir « un caractère », on peut assez vite concevoir des fonctions plus riches (une boucle for et c'est réglé) permettant de sortir des chaînes complètes, ou des chaînes contenant des variables !

C'est le fondement de la famille de fonction qui entourent printf.

Quelques exemples :

```
uint32_t a = 1024;
float pi = 3.14159265359;
char c = '8';
const char * chaine = "un chêne vert";

printf("une chaine avec un entier : %d\n", i);
printf("une autre avec un entier sur 32 bits : %d\n", a);
printf("ou bien avec un float : %f\n", pi);
printf("voici un caractère : %c\n", c);
printf("et même une chaine complète : %s !\n", chaine);
printf("on peut même en mettre plusieurs : %02d/%02d/%04d\n", 1, 4, 2021);
printf("et limiter le nombre de chiffre après la virgule : %.3f\n", pi);
printf("ou afficher des nombres en hexadecimal : %lx - %x\n", a, i);
printf("un même nombre peut prendre plusieurs forme : %x = %c = %d\n", c, c, c);
printf("il y a les tabulations \t, les antislachs \\ et les guillemets \" \n");

char tab[40];
sprintf(tab, "sprintf écrit dans un tableau. %s est un arbre!\n", chaine);
//et pourquoi pas ensuite :
printf(tab);
```

```
int printf (const char *__restrict, ...);
```

On remarque que printf est une fonction dont le nombre d'arguments est variable. Il dépend du nombre de %● dans la 'chaîne de format'. Le ● indique la nature des arguments successifs.

Voici une liste des possibilités :

(visitez <https://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/format.html>)

|       |                 |  |
|-------|-----------------|--|
| %d %i | Decimal/Integer | Entier décimal signé                             |
| %u    | Unsigned        | Entier non signé                                 |
| %o    | Octat           | Entier octal                                     |
| %x %X | heXa            | Hex integer (lettres en minuscule ou majuscules) |
| %c    | Char            | Caractère  |
| %f    | Float           | Flottant   |
| %s    | String          | Chaîne   |
| %%    |                 | Le symbole pourcent : %                          |

TODO : précision / taille du champ / ...

## 28. Float

On a compris qu'il existait des types d'entiers différents par leur taille.

Avec ces types, on peut exprimer des nombres positifs ou négatifs, jusqu'à plusieurs milliards (32 bits). Mais comment exprimer des nombres décimaux ?

Il existe deux types nommés **float** et **double** qui permettent cela.

Voici comment sont stockés ces nombres :

[extraits choisis de l'article Wikipedia : [https://fr.wikipedia.org/wiki/IEEE\\_754](https://fr.wikipedia.org/wiki/IEEE_754)]

L'IEEE 754 est une norme sur l'arithmétique à virgule flottante.



Un float est constitué de : 1 bit de signe, 8 bits d'exposant (-126 à 127), 24 bits de mantisse dont 1 implicite  
Un double est constitué de : 1 bit de signe, 11 bits d'exposant (-1022 à 1023), 53 bits de mantisse dont 1 imp.

$$\text{valeur} = \text{signe} \times \text{mantisse} \times 2^{(\text{exposant} - \text{décalage})}$$

avec :  $\text{signe} = 1 \text{ ou } -1$

et :  $\text{décalage} = 2^{e-1} - 1$  (e est le nombre de bits de l'exposant)

Quelques exceptions permettent de stocker des nombres remarquables :

0, +/-∞, NaN (Not a Number), nombres dénormalisés

Autant dire qu'il est plutôt difficile d'interpréter la valeur d'un float en regardant son contenu binaire !

Ce stockage présente une limite, notamment si l'on veut tester l'égalité de plusieurs nombre décimaux. En effet, si l'on stocke un nombre sur 32 bits, qu'il soit entier ou décimal, on a toujours 4,3 milliards de valeurs possibles. Il faut donc bien admettre que ces 4 milliards de valeurs ne pourront donc représenter qu'une partie seulement de l'infinité des nombres décimaux.

On a donc parfois affaire à des nombres arrondis peu souhaitables !

En pratique, regardez cet exemple troublant :

```
float c, d, e;
c = 0.2;
d = 0.5;
e = c+d;
if(e != 0.7)
    printf("croyez le ou pas, ceci sera affiché !\n");
```

| Name | Type  | Value       |
|------|-------|-------------|
| c    | float | 0.200000003 |
| d    | float | 0.5         |
| e    | float | 0.699999988 |

Ci-dessus ce que l'on peut voir avec le débogueur. On voit clairement que e ne vaut pas 0.7 ! Il ne vaut pas non plus c+d ! Il faut donc être vigilant et toujours privilégier les opérateurs '>' ou '<'.

Il est parfois bien plus malin d'exprimer certaines grandeurs avec des nombres entiers, simplement en changeant l'unité !

```
float distance = 0.35;           //On voudrait écrire ceci
uint32_t distance_mm = 350;      //On peut préférer cela en changeant l'unité !

//Pour afficher ce nombre à virgule stocké et manipulé en tant qu'entier :
printf("%ld,%03ld", distance_mm/1000, distance_mm%1000);

//Et pourquoi pas... pour une résolution plus fine
uint32_t d_mm4096 = 1433600;    //350*4096
//Dans cette unité finement résolue, le nombre maximal autorisé est
// (2^32-1 = 4294967295). Il correspond à 1048576 mm, soit environ 1 km.
```

## 29. Pointeurs

Lorsque l'on manipule un tableau, on le fait par son adresse en mémoire.

De la même façon, on peut accéder à l'adresse mémoire de chaque variable. Cela peut notamment permettre de donner à une fonction la possibilité d'écrire à l'emplacement d'une de nos variables...

Avant de s'intéresser aux nombreux usages possibles des pointeurs, il faut d'abord en comprendre les éléments de syntaxe.

Dans les exemples suivants, on considère qu'on travaille sur un microcontrôleur sur lequel toutes les adresses en mémoire ont une taille de 32 bits. (Par analogie, la taille d'une adresse postale est la même sur l'enveloppe, que celle-ci 'pointe' vers une maisonnette ou un immeuble de 40 étages !)

**'\*' signifie : « la valeur située à l'emplacement »**

**'&' signifie : « l'adresse de »**

```
//Lisez attentivement au moins 2 fois le code ci-dessous.

int a;           //je définis un entier a.
int b;           //je définis un entier b.
int * pa;        //je définis une adresse pa
int * pb;        //je définis une adresse pb

//Pour l'instant, les pointeurs pa et pb ne pointent vers rien de cohérent.
//Peut-être qu'ils pointent vers l'adresse 0... ou peut être ailleurs... cela dépend
//du contenu de la RAM aux emplacements pa et pb). Même chose pour a et b qui
//contiennent quelque chose, mais quoi ?!

pa = &a ;        //pa vaut désormais l'adresse de a. (pa « pointe vers » a)
pb = &b ;        //pb vaut désormais l'adresse de b. (pb « pointe vers » b)

//pa et pb étant des adresses, on peut écrire :
if(*pb != 1)      //si "la valeur située à l'emplacement pointé par pb" !=1
    *pa = 4 ;     //la valeur située à l'emplacement pointé par pa vaut 4.

printf("%lx\n",&pa); //&pa est l'adresse du pointeur pa. Mais on s'en fiche !
printf("%lx\n",pa);  // pa est l'adresse de a. On s'en fiche un peu aussi...
printf("%ld\n",a);   // a... (on verra '4')

int tab[10];       //je définis un tableau de 10 entiers.
int * ptab;        //je définis une adresse ptab. (qui ne vaut rien)
ptab = &tab[4];    //je mets dans ptab l'adresse de la 5ème case du tableau
*ptab = 0xB16B00B4; //je remplis cette 5ème case du tableau
tab[4] = 0xB16B00B4; //j'aurais pu faire la même chose comme cela
*ptab++;           //j'incréménte cette donnée (->0xB16B00B5)
ptab++;            //j'incréménte l'adresse (ptab pointe vers la 6ème case)
fonction1(tab);     //je communique à la fonction l'adresse du tableau
fonction1(&tab[0]) ; //idem...
fonction1(tab[0]) ;  //je communique à la fonction une copie de la 1ère case
```

## 30. Pointeurs sur fonctions

Lors de la compilation, le nom d'une fonction est associée à une adresse mémoire de l'emplacement où est rangée cette fonction.

Il est tout à fait possible d'appeler une fonction si l'on ne connaît que son emplacement :

```
typedef void(*pfun_t) (void); //Type pointeur sur fonction

pfun_t func;                  //définition d'un pointeur sur fonction
func = 0x08123456;           //remplissage du pointeur avec l'adresse d'un emplacement
func();                       //appel de la fonction
```

L'adresse d'une fonction peut se manipuler ainsi :

```
//Type pointeur sur fonction à 1 paramètre entier sur 32 bits
typedef void(*pfun_lint32_t) (uint32_t a);

//prototype d'une fonction ayant comme paramètre un pointeur sur fonction
void SysTick_add_callback_function(pfun_lint32_t func);

//une simple fonction (dont le prototype est conforme au type ci-dessus)
void process_ms(uint32_t param)
{
}

void init(void)
{
    //appel d'une fonction ayant comme paramètre un pointeur sur fonction
    SysTick_add_callback_function(&process_ms);
}
```

Dans cet exemple, on manipule des fonctions ayant un paramètre entier. On peut ainsi décliner des types de pointeurs pour chaque prototype différent.

En savoir + : <https://openclassrooms.com/fr/courses/1252476-les-pointeurs-sur-fonctions>



## 31. volatile

Dans de nombreux systèmes embarqués temps réel, il est impossible de se contenter d'une simple « tâche de fond ». C'est-à-dire d'une simple boucle dans laquelle on scrute l'ensemble des évènements qui pourraient se produire. Il est essentiel de faire appel à la notion d'interruption.

Cette notion qui est expliquée plus en détails dans une autre activité se résume ainsi :

- un programme en cours d'exécution peut être interrompu par une demande d'interruption
- une fonction d'interruption est alors exécutée
- on retourne ensuite poursuivre le programme initial

On utilise alors couramment des 'flags d'interruptions' qui permettent à un programme de savoir que l'interruption a eu lieu.

Typiquement :

```
static volatile uint32_t t = 0;    //temps [ms]

//cette fonction est appelée automatiquement en interruption à chaque ms.
void process_ms(void)
{
    if(t)        //si t est vraie, on le décrémente d'1ms.
        t--;
}

void main(void)
{
    while(1)
    {
        if(t == 0) //si la variable t est retombée à 0, on entre dans le if
        {
            t = 20;                //on 'recharge' t à 20ms.
            blink_led();           //on fait clignoter la led
        }
    }
}
```

Mais mettons-nous un l'instant à la place du compilateur. Lorsqu'on demande au compilateur de produire un code machine 'optimisé', ce dernier cherche des raccourcis afin de rendre le code plus rapide. Si on retire le mot clé 'volatile' dans cet exemple, le fait d'avoir vérifié une fois que t ne vaut pas 0 pourrait suffire pour déduire que puisqu'on y touche pas, il ne vaudra plus jamais zéro. Le compilateur s'autorise alors une optimisation fatale consistant à considérer qu'il est inutile de relire t !

Pour éviter cela, on peut imposer au compilateur que chaque utilisation de la variable t soit effectué en mémoire sans prendre de raccourci. C'est le rôle du mot clé volatile.

Autre exemple avec ce « délai » fabriqué artificiellement avec une boucle for.

```
void blink_led(void)    //faire clignoter la led durant un certina délai
{
    static uint32_t i;

    for(i=0; i<1000000; i++);           //délai
    HAL_GPIO_WritePin(LED_PORT, LED_PIN, 1); //on allume la led
    for(i=0; i<1000000; i++);           //délai
    HAL_GPIO_WritePin(LED_PORT, LED_PIN, 0); //on éteint la led
}
```

Lorsqu'on demande au compilateur de produire un code machine 'optimisé', ce dernier cherche des raccourcis afin de rendre le code plus rapide. Il remplace la boucle par un simple 'i = 1000000' !

Il faut impérativement définir i ainsi : `static volatile uint32_t i;`

## **32.      assert...**

(chapitre en cours de rédaction)

### 33. fonctions et librairies à connaître

Quelques généralités...

| Librairie                | Description brève  |
|--------------------------|--|
| <a href="#">stdio.h</a>  | Librairie standard d'entrées/sorties.<br>Fourni notamment un accès à printf.<br>Création, lecture et écriture de fichiers. |
| <a href="#">math.h</a>   | Fourni des fonctions mathématiques courantes (trigo, arrondis, exponentielles, racine                                      |
| <a href="#">stdlib.h</a> | Conversion de chaîne de caractère en nombre, tri de tableau, random (dépend du système cible)                              |
| <a href="#">string.h</a> | Manipulation de chaînes de caractères (comparaison, calcul de taille, recherche)   |
| <a href="#">ctype.h</a>  | Classification des caractères, mise en majuscule/minuscule...  |
|                          |  |

sizeof(x) est une macro qui renvoie la taille en mémoire du paramètre x. Il s'applique sur une variable, une constante, ou même un type !

## **34.      compilateur**

(chapitre en cours de rédaction)

## 35. Allocation mémoire

Lorsque l'on définit une variable, le compilateur dispose de plusieurs emplacements en mémoire :

| Type de déclaration de variable  | Durée d'existence               | Emplacement mémoire retenu  |
|----------------------------------|---------------------------------|---|
| Dans une fonction, avec 'static' | Permanente                      | Emplacement permanent en mémoire RAM  |
| Dans un fichier (hors fonction)  | Permanente                      | Emplacement permanent en mémoire RAM  |
| Dans une fonction, sans 'static' | Durant l'appel de la fonction   | Emplacement temporaire, dans un registre interne du processeur ou dans la RAM, en zone de pile (au choix du compilateur)    |
| Paramètre d'une fonction         | Durant l'appel de la fonction   | Emplacement temporaire, dans un registre interne du processeur ou dans la RAM, en zone de pile (au choix du compilateur)    |
| Avec le mot clé 'register'       | Selon le lieu de la déclaration | Dans un registre interne du processeur<br><i>Ne jamais utiliser ce mot clé dont vous continuerez d'ignorer l'existence.</i> |
| malloc() / free()                | Gérée par le programme          | Dans la zone de tas, gérée dynamiquement  |

La fonction malloc() permet d'allouer un emplacement en mémoire pour une variable (ou généralement un tableau). Le 'tas' est une zone de mémoire qui sert à cela.

L'allocation dynamique de variable est peu utilisée sur de tout petits systèmes embarqués.

En cas de développement sur un OS ou plusieurs tâches disposent d'une durée de vie limitée, cette approche est plus courante.

Il est à noter un risque de segmentation de la mémoire en cas de création/destruction de ces variables de façon désordonnée.

Et n'oubliez pas : « pour chaque malloc, il y a un free » !

**Utilisation / + d'infos :**

<https://fr.wikipedia.org/wiki/Malloc>