Eskimon



Zeste de savoir

# Gestion de la mémoire sur Arduino

22 mars 2019

# Table des matières



## 1. Introduction

Vous le savez peut-être, l'ordinateur (ou le téléphone, ou la tablette ou le minitel...) avec lequel vous êtes en train de consulter ce tutoriel dispose de plusieurs types de mémoire. Eh bien, de la même façon un microcontrôleur, lui aussi, en embarque plusieurs que nous allons découvrir ensemble.

# 2. Une histoire de mémoire


FIGURE 2. – Barrettes de mémoires RAM (source wikipedia)

Comme je vous l'expliquais dans l'introduction, il y a de fortes chances que le support avec lequel vous consultez ce tutoriel utilise différents types de mémoire. Par exemple, un ordinateur dispose d'un disque dur pour sauvegarder les données sur le long terme et il possède aussi de la mémoire vive (RAM) pour sauvegarder les données d'un programme qui est en train de fonctionner (les variables qu'il manipule par exemple). De la même façon, un être humain possède une mémoire dite à court terme, qui vous permet de vous rappeler d'acheter du lait lorsque vous allez faire vos courses et une mémoire à long terme qui vous permet de vous souvenir des informations qui vous ont marqué et/ou vous sont utiles au quotidien. Pour Arduino la situation est très similaire. On retrouve au total trois mémoires distinctes qui ont chacune un rôle précis : RAM , ROM et Flash .

## 2.1. Les Mémoires

### 2.1.1. La RAM

Tout d'abord la plus simple : la **RAM** qui est la mémoire vive du composant (comme sur votre ordinateur). Mémoire vive, car elle est très rapide et doit gérer beaucoup d'informations très vite. Vous le savez peut être : il existe différents types de RAM, dans notre cas ce sera de la **SRAM** (pour **S**tatic **R**andom **A**ccess **M**emory), qui est plus rapide mais aussi plus consommatrice en énergie que la RAM dynamique de vos ordinateurs et aussi plus encombrante (mais tout ça est bien sûr à relativiser à l'échelle électronique). Elle servira à stocker les variables du programme. Chaque fois que vous faites une nouvelle déclaration de variables, cette dernière se retrouvera dans cette mémoire. Une caractéristique à ne pas négliger, cette mémoire est entièrement effacée lorsque l'alimentation de l'Arduino cesse (on dit qu'elle est « volatile »).

### 2.1.2. L'EEPROM

Ensuite, on trouve une mémoire dite morte, l'**EEPROM** (**E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory). Alors non, n'allez pas croire à la lettre qu'elle est vraiment morte. En fait, on la nomme ainsi car elle est capable de stocker des informations même lorsqu'elle n'est plus alimentée. Cette dernière est similaire au disque dur de votre ordinateur par son comportement et ses caractéristiques. La vitesse d'accès est moins élevée que la RAM et sa durée de vie (nombre de cycle d'écritures possible) est plus faible aussi.

### 2.1.3. La FLASH

Enfin, une dernière mémoire de l'Arduino est la **Flash**. Elle a un rôle un peu particulier, elle sert à stocker le programme que vous téléchargez dans le microcontrôleur. Elle retient donc les informations même lorsque l'alimentation est coupée. Comme dit plus tôt, c'est ici que sont stockées toutes les instructions de votre programme, ainsi que le *bootloader*, un petit bout de code qui agit comme le BIOS de votre PC. Il détecte au démarrage de l'Arduino si on tente de programmer la carte via la liaison série et le cas échéant copiera les données dans la mémoire FLASH. On n'y stocke pas de données pendant l'exécution du programme. En revanche, on peut y stocker des constantes (comme des chaînes de caractères pour votre écran LCD par exemple) afin de gagner un peu de place dans la RAM. D'une manière générale, essayez de la voir comme une mémoire en lecture seule. Mais nous verrons cela plus tard.

## 2.2. Rappel sur les variables

Si vous utilisez déjà Arduino, vous devez le savoir depuis longtemps maintenant, toutes les données d'un programme informatique peuvent être stockées dans des variables. Ces dernières peuvent représenter différentes choses et donc on trouve différents types de variables qui occupent chacun une taille particulière dans la mémoire. Voici une liste exhaustive des différents types de données utilisables classés selon la taille occupée en mémoire :

— 1 octet : `char`, `byte` ( = `unsigned char`), `Boolean`
— 2 octets : `int`, `word` (= `unsigned int`), `short` (= `signed int`)
— 4 octets : `long`, `float`, `double` (= `float` chez Arduino)

Sur la nouvelle Arduino DUE, le `word` reste sur 2 octets.

## 2.3. Résumé

### 2.3.1. Caractéristiques

Voici un petit tableau résumant les caractéristiques des différentes mémoires :

| Nom | Taille (Uno) | Vitesse lecture/écriture | Écriture durant exécution | Simplicité d'utilisation |
|---|---|---|---|---|
| **SRAM** | 2 Ko | Très rapide | OUI | +++ |
| **EEPROM** | 1 Ko | Lent | OUI | ++ |
| **FLASH** | 32 Ko | Rapide | Lecture seulement | + |

### 2.3.2. Les cartes Arduino

Les tailles sont toutes exprimées en kilo-octets (et entre parenthèses se trouve la taille occupée par le bootloader).

| Carte | SRAM | EEPROM | Flash |
|---|---|---|---|
| **Uno** | 2 | 1 | 32 (0.5) |
| **Leonardo** | 2.5 | 1 | 32 (4) |
| **Mega 2560** | 8 | 4 | 256 (8) |
| **DUE** | 96 | 0[1] | 512 (0) |
| **Mini** | 2 | 1 | 32 (2) |
| **Micro** | 2.5 | 1 | 32 (4) |

TABLE 2. – Taille des mémoires dans les différentes cartes Arduino

### 2.3.3. Ajouter de la mémoire

Il est normalement possible de rajouter de la mémoire externe via l'utilisation de composant comme un circuit intégré d'EEPROM ou l'utilisation d'une carte FLASH (une carte SD) que l'on retrouve partout dans les appareils photo et téléphones portables. Cependant, toutes ces solutions reposent sur l'utilisation d'un protocole de communication différent à chaque fois ou d'adaptation électronique ou d'autres contraintes. Nous ne traiterons donc pas ces différentes mémoires dans ce chapitre pour nous concentrer uniquement sur ce qui est disponible au sein de l'Arduino (et qui sera amplement suffisant pour commencer tous vos premiers projets).

1. Avec la DUE il est en théorie possible d'écrire dans la mémoire Flash pendant l'exécution du programme, mais aucune bibliothèque dédiée n'existe pour le moment.

# 3. La SRAM ou mémoire vive

Commençons les choses tranquillement avec ce qui se fait de plus facile : la SRAM. Pour rappel, cette mémoire est équivalente à la mémoire vive de votre ordinateur. Dans le cas de l'Arduino UNO, nous disposons de 2 Kilo-Octets (2 KB), ce qui représente un total de 2048 octets. En terme de quantité de variables, cela représente au choix :

— 2048 `char`
— 1024 `int`
— 512 `float`

Bien entendu, vous pouvez y stocker tous les types de données que vous souhaitez, du plus simple au plus farfelu. Par exemple, vous pouvez y mettre quelques `char` pour définir les broches à utiliser en entrées/sorties et une chaîne de caractères que vous utiliserez pour un message pour votre écran LCD. Si l'on part de cette liste, on pourrait obtenir :

— Trois `char` pour définir trois entrées/sorties (Led1, Led2, Bouton)
— Un tableau de caractère "Salut les gens !" (donc 17 `char` avec le caractère '\0' de fin de chaîne)

Cela nous fait un total de 20 octets en RAM. Avec Arduino (tout comme avec votre ordinateur), aucune complication pour lire et écrire des variables dans la RAM. C'est le cœur de la machine qui s'en occupe. C'est totalement transparent pour vous. Par contre, contrairement à votre ordinateur, Arduino ne possède pas plusieurs Giga-octet de RAM. C'est pourquoi il est souvent judicieux de réfléchir au type de la variable à déclarer lorsqu'on en crée une.

Par exemple, pour stocker l'état d'un bouton ou un âge, inutile de prendre un `int`, un simple `char` suffit et vous économiserez alors 1 octet par variable. Cela peut sembler trivial, mais on n'y pense pas forcément lorsqu'on arrive d'un milieu où la mémoire est souvent le cadet des soucis.

Cet aspect est d'autant plus important qu'il est assez difficile de déceler une incohérence de comportement du programme à cause de la mémoire manquante.

En effet, sur votre ordinateur le système d'exploitation (OS, Operating System) possède un certain contrôle sur la quantité de mémoire maximale autorisée par programme. Sur Arduino, pas d'OS donc pas de message d'erreur lorsque la mémoire est saturée. Le microcontrôleur essaiera tant que possible de faire tenir les variables en mémoire, mais s'il ne peut pas le comportement peut devenir imprévisible et les problèmes de RAM sont souvent la dernière chose à laquelle on pense. Donc un conseil : méfiez-vous lorsque vous déclarez vos variables ! (surtout si vous déclarez de nombreuses chaînes de caractères qui prennent rapidement de la place).

? Et c'est tout ?

Eh oui. La RAM est une mémoire vraiment simple à utiliser puisque c'est complètement transparent ! (tant que vous ne déclarez pas des variables à tort et à travers )

# 4. L'EEPROM une mémoire "morte"

Comme nous le disions plus tôt, cette mémoire est un peu le "disque dur" de votre carte à la différence qu'il n'y a pas de partie mécanique (donc pas de casse possible). En revanche, la durée de vie de cette mémoire possède un nombre de lectures/écritures limité (environ 100 000 lectures/écritures pour chaque octet). Comme pour tout système de mémoire, elle fonctionne à partir d'un mécanisme d'adresse. Un peu comme si vous rangiez des informations dans un livre, avec une information par page. La taille d'une information est ici d'un octet, et le nombre de cases dans lequel on peut stocker ces infos est de 1024 (sur une Arduino Uno). Vous pouvez donc stocker 1024 octets au total. Vous pouvez aussi stocker 512 `int` par exemple (1024/2) ou fait un mix des deux. Pour pouvoir manipuler l'EEPROM, il vous faudra dans un premier temps inclure une bibliothèque bien nommée : `EEPROM.h`.

```
1 ##include "EEPROM.h"
```

*i* Une case mémoire qui n'a jamais été utilisée possède une valeur initiale de 255.

## 4.1. Enregistrer des données

La mémoire EEPROM est donc divisée en 1024 blocs de 8 bits. Pour écrire une donnée, il va falloir décider dans quel bloc on veut l'enregistrer, c'est ce qu'on appelle **l'adresse d'écriture**. Comme la mémoire est initialement vide quand vous achetez la carte, vous pouvez choisir comme bon vous semble où vous voulez mettre les informations, à quelle adresse, entre 0 et 1023. Par contre lorsque vous voudrez les récupérer il faudra vous souvenir où elles sont ! Pour enregistrer une donnée c'est très simple, il suffit simplement d'utiliser une seule fonction : `write()`. Comme elle appartient à la librairie EEPROM, et pour qu'elle ne soit pas confondue avec une autre, elle est déclarée dans un ensemble (un *namespace*) qui s'appelle EEPROM. Pour utiliser la fonction il faudra donc écrire :

```
1 EEPROM.write()
```

Cette fonction prend deux arguments :

— L'adresse où écrire (un `int` entre 0 et 1023)
— L'octet à enregistrer (un `unsigned char`)

Par exemple pour enregistrer la valeur 42 à l'adresse 600 on fera :

```
1 EEPROM.write(600, 42); //adresse = 600, valeur = 42
```

## 4.2. Lire des données

Je suis intiment persuadé que vous n'aurez aucun mal à deviner comment lire des données depuis la mémoire... Vous avez trouvé ? En effet, il suffit d'utiliser une autre fonction avec un nom très explicite : read() ! Comme pour sa cousine l'écriture, il faudra la faire précéder de "EEPROM" pour y accéder. Cette fonction ne prendra qu'un seul argument qui sera l'adresse à laquelle on veut aller chercher notre donnée. Exemple, je vais lire la donnée à l'adresse 600 :

```
1 unsigned char donnee = EEPROM.read(600); //adresse = 600
```

La variable `donnee` prendra donc la valeur stockée à l'adresse mémoire numéro 600. Soit 42. Et voilà, vous savez tout sur l'écriture et la lecture dans l'EEPROM.

Hep hep hep, minute papillon. C'est sympa tout ça mais je fais comment si je veux stocker un `float` par exemple ?

Excellente question, nous allons voir comment nous allons y répondre !

## 4.3. Limite de ces fonctions

Comme vous l'avez vu, les deux fonctions présentées ci-dessus sont fait pour lire/écrire un seul octet à la fois. Ce qui veut dire qu'on ne peut pas les utiliser pour enregistrer un `int, float, double`... C'est très embêtant... Mais bien entendu, chaque problème à sa solution. Par contre il va falloir mettre les mains dans le cambouis et faire nos propres fonctions d'enregistrement pour pouvoir conserver des types de variable plus grand qu'un simple octet.

### 4.3.1. Rappel sur l'opérateur de décalage binaires et les masques

Cette partie parle de masquage. Si vous ne connaissez pas ce terme, je vous invite a lire cette partie de tutoriel sur l'utilisation des registres à décalage et fait ainsi la découverte de l'*opérateur de décalage binaire.*

Pour rappel, les opérateurs de décalage permettent de décaler tous les bits d'une variable vers la droite ou vers la gauche. Ils s'écrivent avec des chevrons : `<` pour décaler à gauche et `>` pour décaler à droite. L'écriture se fait de la manière suivante :

```

```

Par exemple, pour décaler de 3 bits vers la gauche :

```
int variable = 42 ;
variable = variable << 3 ;
```

Les **masques**, qui vont être utiles pour la suite, sont réalisés grâce à deux opérateurs logiques bits-à-bits, le OU ( `|` ) et le ET ( `&` ). Un OU permettra d'imposer un bit à 1 tandis que le ET permettra d'imposer un bit à 0. Par exemple, pour mettre les 4 derniers bits d'un octet à 1 on fera :

```
unsigned char monOctet = 42 ; // en binaire : 0010 1010
monOctet = monOctet | 0x0F ; //équivaut à 0010 1010 | 0000 1111 =
    0010 1111
```

De même pour mettre les quatre derniers à 0 on fera :

```
unsigned char monOctet = 42 ; // en binaire : 0010 1010
monOctet = monOctet & 0xF0 ; //équivaut à 0010 1010 & 1111 0000 =
    0010 0000
```

Encore une fois, pour plus de détails sur les opérateurs de décalage ou de masquage, référez-vous à ce tutoriel . Maintenant que les rappels sont faits, nous allons voir comment faire pour lire et écrire dans l'EEPROM des variables qui font plus d'un octet. Pour cela nous allons réaliser deux fonctions qui prendront pour exemple l'écriture/lecture d'un `int` (mais vous allez comprendre le principe et serez donc capable de faire sans problème la même chose pour tous les types de données )

### 4.3.2. Écrire un `int` dans l'EEPROM

Un `int` qui représentera la variable à mettre en mémoire. Afin de garder les choses simples, on enregistrera les valeurs à la suite, dans des cases mémoires consécutives. Pour pouvoir mettre notre `int` dans les deux cases, il va falloir le découper en deux pour obtenir deux octets. On va d'abord commencer par isoler les 8 bits les plus à droite (bits de poids faible) grâce à un simple masque. Ensuite, on va faire évoluer le masque en le décalant 8 fois vers la gauche et ainsi isoler les bits de poids fort.L'enregistrement se fera alors de la manière la plus simple du monde, en faisant deux enregistrements successifs à l'adresse n et n+1.

*i* Je vous invite à essayer par vous-même avant de regarder le code suivant, cela vous fera un bon exercice.

```
1  //on veut sauvegarder par exemple le nombre décimal 55084, en
       binaire : 1101 0111 0010 1100
2
3  //fonction d'écriture d'un type int en mémoire EEPROM
4  void sauverInt(int adresse, int val)
5  {
6      //découpage de la variable val qui contient la valeur à
           sauvegarder en mémoire
7      unsigned char faible = val & 0x00FF; //récupère les 8 bits de
           droite (poids faible) -> 0010 1100
8      //calcul : 1101 0111 0010 1100 & 0000 0000 1111 1111 = 0010
           1100
9
10     unsigned char fort = (val >> 8) & 0x00FF;  //décale puis
           récupère les 8 bits de gauche (poids fort) -> 1101 0111
11     //calcul : 1101 0111 0010 1100 >> 8 = 0000 0000 1101 0111 puis
           le même & qu'avant
12
13     //puis on enregistre les deux variables obtenues en mémoire
14     EEPROM.write(adresse, fort) ; //on écrit les bits de poids fort
           en premier
15     EEPROM.write(adresse+1, faible) ; //puis on écrit les bits de
           poids faible à la case suivante
16 }
```

### 4.3.3. Lire un *int* depuis l'EEPROM

Je ne sais pas si vous avez trouvé le code précédent simple, mais si c'est le cas alors pas d'inquiétude car on reste sur le même concept. De même que précédemment, je vous invite à lire ce que l'on va faire, essayer, puis regarder la solution après. Le principe est le suivant. Nous allons tout d'abord récupérer l'octet de poids fort puis celui de poids faible qui composait la variable de type `int` reconstituée ! Puis on va reconstruire notre `int` à partir de ces deux morceaux.

```
1  //lecture de la variable de type int enregistrée précédemment par
       la fonction que l'on a créée
2
3  int lireInt(int adresse)
4  {
5      int val = 0 ; //variable de type int, vide, qui va contenir le
           résultat de la lecture
6
7      unsigned char fort = EEPROM.read(adresse);     //récupère les 8
           bits de gauche (poids fort) -> 1101 0111
```

```
8       unsigned char faible = EEPROM.read(adresse+1); //récupère les 8
            bits de droite (poids faible) -> 0010 1100
9
10      //assemblage des deux variable précédentes
11      val = fort ;          // val vaut alors 0000 0000 1101 0111
12      val = val << 8 ;      // val vaut maintenant 1101 0111 0000 0000
            (décalage)
13      val = val | faible ; // utilisation du masque
14      // calcul : 1101 0111 0000 0000 | 0010 1100 = 1101 0111 0010
            1100
15
16      return val ; //on n'oublie pas de retourner la valeur lue !
17  }
```

## 5. La Flash, mémoire de programme, morte et vive à la fois !

Maintenant que vous avez tout compris aux différents types de mémoires et que l'on a vu ensemble comment manipuler les plus simples, nous allons pouvoir passer à la dernière, la plus compliquée, la mémoire **Flash** dite "de programme". Cette mémoire, appelée plus communément "mémoire de programme" (ou encore "Progmem") sert d'ordinaire à stocker le code que vous avez créé puis compilé, le programme en somme. En effet, lorsque vous "téléversez" votre programme (beurk cette traduction) vers le microcontrôleur, c'est ici qu'il sera envoyé. Comme toutes les mémoires flash, sa durée de vie (exprimée en nombres de lectures/écritures) n'est pas infinie. L'utilisation de cette flash est un peu particulière. En effet, on ne peut enregistrer des données dedans qu'au moment du téléchargement du programme. Une fois le programme chargé, elle agit en lecture seule, si bien que vous ne pourrez que récupérer des données injectées plus tôt mais pas en rajouter de nouvelles au moment du fonctionnement normal (par exemple pour y stocker des valeurs de variables).

? Alors, ça ne sert à rien une mémoire en lecture seule !?

Détrompez-vous, c'est en fait très utile pour mettre des données qui ne varient pas et qui risqueraient d'encombrer votre RAM par exemple. Le premier des usages est souvent le stockage de chaîne de caractères, que l'on va plus tard envoyer sur un écran LCD. Par exemple je veux afficher à chaque démarrage du programme le message "Salut les gens !" sur mon écran. La première méthode serait donc de faire un truc du genre :

```
1  const char message[] = "Salut les gens !" ;
2  //const pas obligatoire mais c'est plus rigoureux avec
3
4  ... //du code
5
6  monlcd.print(message) ;
```

Cette ligne de code va être interprétée de façon à enregistrer la chaîne dans la mémoire RAM. Ce message qui fait 16 caractères prendra donc 16 octets dans votre RAM (en fait 17 avec le caractères de fin de chaîne '\0'). Ça peut paraître insignifiant, mais si votre programme contient plusieurs chaînes pour afficher du texte de manière dynamique ça peut vite devenir serré. Comme ce message n'a pas besoin d'être modifié pour être ensuite ré-enregistré, la meilleure des solutions reste de le stocker dans la mémoire de programme qui est beaucoup plus grande que la mémoire RAM. Il ne prend ainsi pas de place en RAM et on pourra toujours le récupérer à chaque fois que l'on en aura besoin .

## 5.1. Sauvegarder/charger des variables

Maintenant que le concept est posé, passons un peu à la pratique . Nous allons commencer par enregistrer et recharger des variables "simples" (un `int` par exemple) puis ensuite nous chercherons à stocker des variables plus compliquées comme un tableau de caractères.

L'utilisation de la mémoire de programme repose sur un mot-clé qui nous sert d'attribut modificateur de variables. En simple, c'est ce mot-clé qui dira au compilateur "Cette variable il faut la mettre en mémoire flash de programme". Ce mot-clé est `PROGMEM` (tout en majuscules). Pour pouvoir l'utiliser, il vous faudra aussi intégrer la librairie de gestion de cette mémoire :

```
1 ##include <avr/pgmspace.h>
```

### 5.1.1. Des variables simples

Pour enregistrer une variable, il vous suffira simplement de la déclarer comme d'habitude, à la seule différence qu'il faut rajouter le modificateur `PROGMEM` pour qu'elle soit enregistrée au bon endroit, et la déclarer comme constante avec `const`. Par exemple pour enregistrer un `int` :

```
1 ##include <avr/pgmspace.h> //on n'oublie pas d'intégrer la
      bibliothèque de gestion de mémoire
2
3 const int unInt PROGMEM = 42; //ce int est enregistré en mémoire
      flash
4
5 int unAutreInt = 42; //celui-ci sera mis en RAM
```

*i* Le mot-clé PROGMEM peut aussi s'écrire avant le type, mais pas entre le type et le nom de la variable : `const PROGMEM int unInt = 42 ;`

Il semble, à ce jour, qu'il y ait un problème avec la sauvegarde des nombres flottants. Les seuls types à utiliser sont donc `char`, `int` et `long` (unsigned ou signed)

Une fois que les variables sont enregistrées, il ne nous reste plus qu'à les récupérer pour les utiliser dans notre programme. Pour cela, il existe des fonctions pour chaque type de variable. Elles sont toutes plutôt simples à retenir :

- `pgm_read_byte()` -> pour lire un `char`
- `pgm_read_word()` -> pour lire un `int`
- `pgm_read_dword()` -> pour lire un `long`

Pour chacune de ces fonctions, il vous faudra mettre en argument la variable créée, précédée par le symbole '&'. Les habitués des pointeurs doivent savoir pourquoi. Pour les autres voici une petite explication. Lorsque vous déclarez votre variable, celle-ci va sagement se mettre dans une case mémoire. Cette case possède une adresse pour retrouver la variable plus tard (comme avec l'EEPROM souvenez-vous). Lorsque vous faites appelle à la fonction `pgm_read_byte()`, vous devez passer l'adresse de la variable plutôt que sa valeur (passer la valeur n'a en effet aucun intérêt ici). C'est ce que permet l'opérateur '&'.

Je comprends que cela soit flou, et le cours n'est pas là pour faire une explication exhaustive des pointeurs. Je vous demanderais juste de ne pas oublier de mettre le symbole '&' pour utiliser la variable en flash.

Un petit exemple :

```
1  ##include <avr/pgmspace.h> //on n'oublie pas d'intégrer la
       bibliothèque de gestion de mémoire FLASH
2
3  const unsigned char unChar PROGMEM = 42;      //ce char est
       enregistré en mémoire flash
4  const unsigned int unInt PROGMEM = 1324;      //ce int est
       enregistré en mémoire flash
5  const unsigned long unLong PROGMEM = 987654; //ce double est
       enregistré en mémoire flash
6
7  void setup()
8  {
9      Serial.begin(9600);
10     Serial.print("Mon char : ");
11     Serial.println(pgm_read_byte(&unChar));
12     Serial.print("Mon int : ");
13     Serial.println(pgm_read_word(&unInt));
14     Serial.print("Mon long : ");
15     Serial.println(pgm_read_dword(&unLong));
16 }
17
```

```
18 void loop()
19 {
20
21 }
```

### 5.1.2. Une chaîne de caractères

Un usage très fréquent de l'utilisation de la mémoire de programme est le stockage de chaînes de caractères vouées à être affichées plus tard. En effet, une chaîne de caractères qui sert uniquement à indiquer un menu ou un message d'accueil ne sera pas modifiée et a donc pleinement sa place dans une mémoire en lecture seule. On va commencer par déclarer un tableau comme on le ferait normalement, puis comme précédemment on va lui ajouter le modificateur `PROGMEM` :

```
1 ##include <avr/pgmspace.h> //on n'oublie pas d'intégrer la
      bibliothèque de gestion de mémoire
2
3 const char message[] PROGMEM = "Salut les gens !";
4 //écriture équivalente :
5 const PROGMEM char message[] = "Salut les gens !";
```

Maintenant que les données sont enregistrées, l'étape de lecture arrive et c'est plus délicat... En effet, les seules fonctions permettant de lire des données dans la Flash sont celles que nous avons vues juste avant, et elles ne permettent donc que de récupérer qu'un `char`... Je sens qu'on va s'amuser ! Il va donc falloir créer une boucle pour tout récupérer ! Comme ici le contenu stocké est une chaîne de caractères, nous allons détecter le caractère de fin de chaîne pour arrêter la boucle. Il n'y a pas de solutions magiques, chaque cas doit avoir son traitement (si vous ne stockez pas que des chaînes de caractères). Cette fois-ci, par contre, nous n'allons pas mettre le symbole '&' devant le nom de la variable dans la fonction `pgm_read_byte()`. En effet, un tableau représente déjà une adresse mémoire et il n'est donc pas nécessaire d'utiliser le '&' pour l'indiquer.

```
1 ##include <avr/pgmspace.h> //on n'oublie pas d'intégrer la
      bibliothèque de gestion de mémoire
2
3 const char message[] PROGMEM = "Salut les gens !"; //chaîne de
      caractères enregistrée dans la mémoire FLASH
4
5 void setup()
6 {
7     Serial.begin(9600);
8     char temp = pgm_read_byte(message); //on récupère le premier
          caractère
9     char i=0; //compte le nombre de déplacement
```

```
10      while(temp != '\0') //tant que le caractère récupéré est
            différent du caractère de fin de chaîne
11      {
12          Serial.print(temp); //on affiche le caractère lu
13          i++; //on incrémente le déplacement
14          temp = pgm_read_byte(message + i); //on récupère le
                caractère suivant
15      }
16      Serial.println();
17  }
```

Une autre solution existe cependant **pour les chaînes de caractères uniquement**. En effet, si vous voulez utiliser une chaîne sans vous fatiguer, vous pouvez simplement utiliser :

```
1  F("Chaine completement en flash !")
```

Lors de la compilation, tout le mécanisme de stockage et de lecture sera ainsi mis en place de manière transparente. Par exemple vous pourriez afficher la même chose que ci-dessus en faisant :

```
1  ##include <avr/pgmspace.h> //on n'oublie pas d'intégrer la
       bibliothèque de gestion de mémoire
2
3  void setup()
4  {
5      Serial.begin(9600);
6      Serial.println(F("Salut les gens !"));
7  }
```

Cela dit, si vous devez utiliser plusieurs fois la même chaîne de caractères cette solution n'est pas idéale puisque l'espace de stockage utilisé sera différent pour chaque appel de cette fonction même si le contenu ne change pas, ce qui vous fera donc consommer de la mémoire pour rien... enfin si, pour ne pas se fatiguer avec le code !

## 6. Conclusion

Vous en savez maintenant un peu plus sur les différents types de mémoires présentes au sein d'Arduino.

Cependant, si vous avez toujours un besoin plus important de mémoire, vous pouvez essayer de vous tourner vers des composants tels que des EEPROM externes.