Renaud Pinon
Développeur
Créateur de Tiny Code Studio

Arduino : on fait joujou avec la SRAM

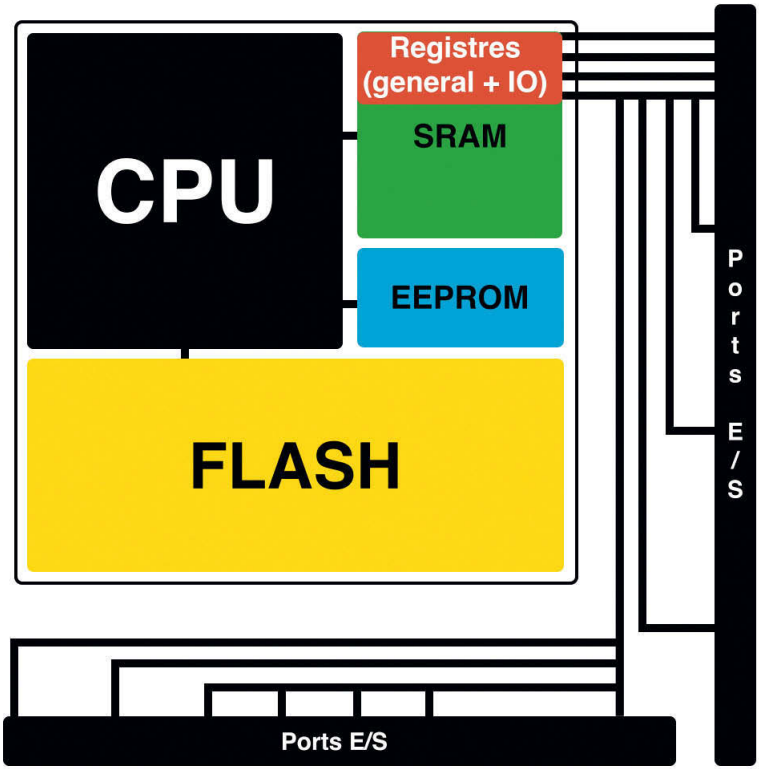
Je vous propose aujourd'hui de plonger dans les tréfonds de la mémoire d'une carte Arduino. J'expliquerai donc quels sont les différents types de mémoires, de quelles façons elles sont architecturées et comment obtenir, via la connexion série USB, une partie du contenu de la SRAM. Nous verrons ensuite dans un prochain article comment utiliser ces informations pour nous fabriquer un mini debugger.

niveau 200

C'est l'instant publicité ;) Le sujet de cet article s'est imposé à moi suite au développement de mon application Tiny Code Studio, un IDE permettant de programmer les cartes Arduino (https://www.tinycodestudio.com). Il est pour le moment disponible en version beta pour Mac et la version Windows est en cours de développement.

Je souhaitais une application qui puisse déboguer les programmes Arduino sans avoir à utiliser de boîtier matériel. D'une part car la plupart d'entre eux se révèlent assez coûteux, mais aussi car la majorité d'entre eux sont très pénibles à configurer dans un environnement de développement. L'application devait avoir la possibilité d'afficher le contenu des variables et donc d'obtenir leur valeur à partir de leur adresse, quel que soit leur type (variable standard de type int par exemple ou pointeur vers un objet ou une structure). Il fallait également qu'elle puisse obtenir le numéro de l'instruction en cours ainsi que celui de l'instruction de retour des fonctions.

Toutefois, cette quête ne s'est pas faite sans douleur tant la documentation sur des notions telles que le Program Counter ou les Stack Frames, que l'on verra dans un prochain article, sont très limitées en ce qui concerne les puces Atmel. J'espère ainsi pouvoir partager quelques informations intéressantes qui, même si elles ne vous serviront pas tous les jours, vous permettront de mieux comprendre le fonctionnement de ces petites cartes. ;)



Détails techniques

Une carte Arduino est en réalité très simple : il s'agit principalement d'une puce (un microcontrôleur ou MCU) de type Atmel dont les entrées et sorties sont reliées à des connecteurs (les fameuses entrées/sorties).

Le microcontrôleur est subdivisé en un microprocesseur (l'unité qui va exécuter les instructions du programme), la mémoire Flash (qui contient les instructions du programme) et la SRAM (mémoire volatile réinitialisée à chaque démarrage). Reste la mémoire EEPROM que nous n'aborderons pas ici (mémoire non volatile qui peut être modifiée directement par le programme, par exemple pour stocker des paramètres après extinction).

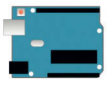
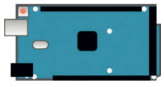
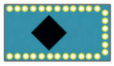
Voici le détail :

- Microprocesseur : architecture AVR 8bits de type RISC. Même s'ils sont 8bits, ces microprocesseurs peuvent parfaitement effectuer des opérations sur 2 ou 4 octets. Pour compiler le programme en AVR, on peut utiliser la suite GCC et ses exécutables de type "avr" (avr-gcc, avr-asm, avr-objdump, …). Ce sont d'ailleurs les exécutables qu'utilise Arduino IDE pour compiler.
- Mémoire Flash : on pourrait la comparer à un CD-ROM pour un ordinateur. C'est une mémoire non modifiable qui contient le programme que vous avez envoyé à la carte (un fichier .hex ou .elf). Elle comprend également certaines données telles que les variables définies avec l'attribut PROGMEM.
- Mémoire SRAM (Static Random Access Memory) : l'équivalent de la RAM d'un ordinateur. C'est une mémoire réinitialisée à chaque démarrage de la carte. Elle contient toutes les données que le programme utilise au cours de son exécution : variables globales, locales, données de registres "poussées" sur la pile (instruction "PUSH" en Assembler), paramètres passés aux fonctions, … Autre fait un peu moins connu : sur les puces Atmel, les valeurs courantes des registres du microprocesseur sont directement accessibles à partir de l'adresse 0x0000 de la SRAM ! Les processeurs Atmel ayant tous en commun au moins 32 registres 8 bits (R0 à R31), vous saurez que vous pouvez obtenir la valeur de chacun d'entre eux en lisant la valeur allant de 0x0000 à 0x001F. Et à la suite se trouvent les registres d'entrées/sorties dont le nombre est variable suivant le modèle de puce.

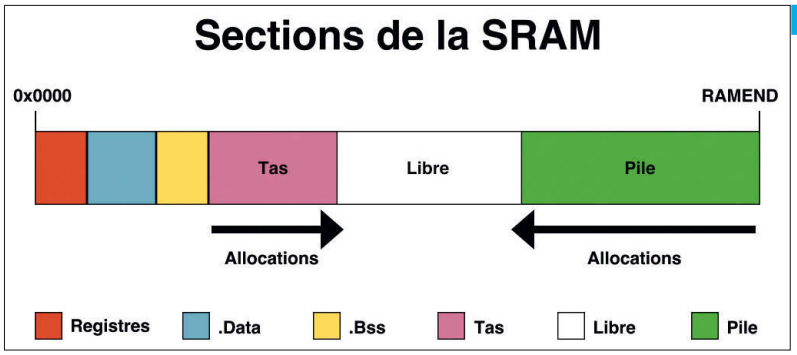
Un point sur la notation mémoire

La notation mémoire que j'utiliserai dans cet article est évidemment l'hexadécimal (base 16 de 0 à F). Grâce à elle on peut représenter plusieurs octets, par exemple la valeur 0x2F9A qui est une valeur sur 2 octets. Attention : si on récupère l'octet 0x2F suivi de l'octet 0x9A, alors pris ensemble ils ne donnent pas la valeur 0x2F9A mais 0x9A2F (=39471 en décimal), le premier octet en mémoire étant celui de poids faible.

La mémoire Flash est très limitée : tous les modèles actuellement en vente embarquent au moins 32 ko, et jusqu'à 256 ko pour l'Arduino Mega. Une instruction Assembleur nécessitant de 2 à 4 octets, il faudra être concis. Il faut de plus retrancher l'espace utilisé par le bootloader, un morceau de code nécessaire pour l'envoi du programme via l'USB sans avoir à utiliser de boîtier matériel externe. La taille du bootloader change suivant le modèle de carte.
La SRAM, quant à elle, est encore plus limitée : de 1 à 8 ko suivant les modèles. Oui, vous avez bien lu : 1ko, c'est à peine l'espace pour stocker 512 variables de type int. Il faudra donc bien dimensionner le type de carte suivant le projet que l'on veut réaliser ! Nous verrons aujourd'hui particulièrement la SRAM, la mémoire Flash étant davantage étudiée dans le prochain article.
Ci-dessous un descriptif de quelques modèles de cartes : 1

	UNO	MEGA	Nano	Pro Mini
				
Flash	32ko	256 ko	32 ko	32 ko
(BL = BootLoader)	(BL = 512 o)	(BL = 8 ko)	(BL = 2ko)	(BL = 2 ko)
SRAM	2 ko	8 ko	2 ko	2 ko
EEPROM	1 ko	4 ko	1 ko	1 ko
Entrées/Sorties (D = digital, A = analog)	14D + 6A	54D + 16A	14D + 8A	14D + 8A

1



2

Fonctionnement de la SRAM

La SRAM est décomposée en plusieurs sections : l'une est de taille fixe (les registres), les autres de tailles variables. Voici la représentation schématique d'une SRAM : 2

- Registres : valeurs des différents registres du processeur : R0 à R31 qui ont une taille fixe ainsi que les registres E/S, qui sont de taille variable suivant le modèle de carte. Pour vulgariser, ces derniers correspondent à l'état des différentes broches de votre Arduino (même si la lecture n'est pas aussi simple que ça puisque l'état d'une seule broche dépend de plusieurs registres à la fois).
- .Data : contient les variables globales initialisées + les variables locales "static" initialisées.
- .Bss : contient les variables globales non initialisées + variables locales "static" non initialisées.
- Tas (*Heap*) : blocs de données alloués par malloc() ou l'opérateur new. Les nouvelles allocations ont lieu de manière croissante (la valeur de l'adresse augmente petit à petit). Contrairement aux variables classiques, les variables déclarées par *malloc()* ou *new* doivent obligatoirement être libérées, respectivement par *free()* et *delete*, sous peine de perdurer en mémoire même si la fonction dans laquelle elles ont été déclarées est terminée.
- Pile (*Stack*) : contient les variables locales qui ne sont pas allouées par *malloc()* ou *new*. Il s'agit des variables dites "classiques" (int, float, structures, …). Elles sont automatiquement créées à l'entrée d'une fonction et sont automatiquement supprimées de la mémoire lors de la sortie de cette fonction. A savoir que les variables de la pile sont déclarées dans le sens inverse du tas : la section débute à l'adresse *RAMEND* et décroit petit à petit. Si jamais le tas ou la pile grandit de façon trop importante, vous risquez la collision !

Obtenir l'adresse d'une variable

Les adresses mémoires sont codées sur 2 octets. Pour obtenir l'adresse mémoire d'une variable on utilise l'opérateur de référencement "&" et on caste le tout en *int* ou *uint16_t* (les SRAM internes ne dépassent jamais 8 ko donc ça tient sur 2 octets). On teste :

```
void setup()
{
  Serial.begin(9600);
}

void loop()
{
  int x = 0, y = 0, z = 0;

  uint16_t varAddress1 = (uint16_t)&x;
  uint16_t varAddress2 = (uint16_t)&y;
  uint16_t varAddress3 = (uint16_t)&z;

  Serial.println("Addr1: 0x" + String(varAddress1, HEX));
  Serial.println("Addr2: 0x" + String(varAddress2, HEX));
  Serial.println("Addr3: 0x" + String(varAddress3, HEX));

  // Valeur fin de RAM:
  Serial.println("RAMEND: 0x" + String(RAMEND, HEX));

  delay(10000);
}
```

Nous initialisons 3 variables dans la fonction *loop()*. Le type int étant un type "classique", les variables devraient se retrouver dans la pile et donc avoir des adresses décroissantes. Sur un *Arduino Mega*, j'obtiens le résultat suivant :

Addr1: Ox21ec
Addr2: Ox21ea
Addr3: Ox21e8
RAMEND: Ox21ff

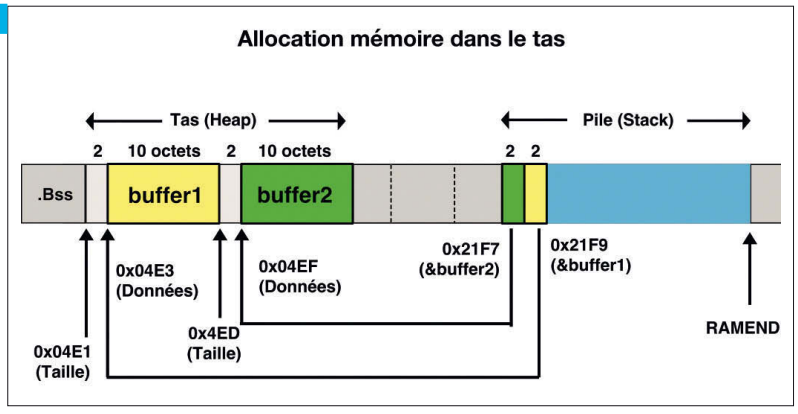
On a bien des adresses décroissantes et espacées de 2 octets, ce qui est la taille du type *int*.
Pour obtenir la taille d'une variable en mémoire, on utilise la fonction sizeof(). Par exemple, et ce, quelle que soit la machine, sizeof(uint16_t) donnera toujours "2", sizeof(uint32_t) donnera toujours "4". int ou long sont par contre dépendants de la machine sur laquelle s'exécute le code (2 et 4 octets pour les Arduino).
On peut passer en argument de sizeof() soit le nom d'un type, soit le nom d'une variable, ce qui rend la fonction particulièrement versatile. Cela fonctionne également pour les tableaux : si vous déclarez un tableau de 10 éléments de type uint32_t, alors sizeof(tableau) vous renverra bien la valeur 40. Cela fonctionne aussi avec les structures et objets du moment qui ne sont pas des pointeurs (donc pas déclarés avec l'opérateur new). Sizeof(), utilisée sur un pointeur, renverra toujours 2.

Allocation dans le tas

Essayons maintenant d'allouer deux buffers de 10 octets chacun. Ils devraient logiquement se trouver dans le tas :

```
// void setup() éludée…
void loop()
{
  uint8_t* buffer1 = (uint8_t*)malloc(10);
  uint8_t* buffer2 = (uint8_t*)malloc(10);

  Serial.println("Addr1: 0x" + String((uint16_t)buffer1, HEX));
  Serial.println("Addr2: 0x" + String((uint16_t)buffer2, HEX));

  free(buffer1);
  free(buffer2);

  delay(10000);
}
```

3



On obtient le résultat:

Addr1: Ox4e3
Addr2: Ox4ef

Si on regarde la documentation de l'Arduino Mega, les registres se terminent à l'octet 0x200 (512 en décimal) de la SRAM. Se trouvent ensuite les sections .data et .bss, puis le tas. On peut donc supposer que 0x04E3 se trouve bien au début du tas. Mais surprise ! 0x4EF moins 0x4E3 donne 0x0C, soit 12 en décimal et non pas 10. C'est en réalité parce qu'un bloc mémoire alloué dans le tas est toujours précédé de 2 octets décrivant sa longueur. Cela veut dire que si on veut connaître la taille d'un bloc inconnu, il suffit de lire la valeur à pointeur - 2. Nice ! 3

Obtenir les données d'une variable

Pour obtenir la valeur d'une variable, nous allons regarder l'adresse de cette dernière, puis convertir les octets à cette position en hexadécimal. Comme les types de variables peuvent avoir une longueur de données différente, nous allons créer une fonction qui lit la valeur à l'adresse voulue sur une taille d'octets déterminée, puis l'imprime en hexadécimal sur le port série de l'ordinateur :

```
void PrintVariable(uint16_t address, int len)
{
  // On place un pointeur à l'adresse:
  uint8_t* pointer = (uint8_t*)address;

  Serial.print("0x"); // On commence l'écriture.
  for (int i = 0 ; i < len ; i++)
  {
    uint8_t val = *(pointer+i); // valeur d'1 octet.
    Serial.print((val < 16 ? "0" : "") + String(val, HEX));
  }
  Serial.println(""); // CRLF

}
```

Comme dit précédemment, il faut lire les octets "à l'envers" si on veut obtenir la valeur d'une variable sur plusieurs octets (int, long, uint16_t, …). Le premier octet étant celui de poids faible, si on a un uint32_t dont les octets sont 01 00 00 00, alors il faut les inverser ce qui donne la valeur 0x00000001. En revanche, si on imprime un tableau ou une variable chaîne de type char*, alors les données seront dans le bon ordre.
On peut appeler la fonction de la façon suivante :

```
void loop()
{
  long maVar = 32;
  // Obtention de l'adresse avec '&'
  // et de la taille avec sizeof():
  PrintVariable((uint16_t)&maVar, sizeof(maVar));

  delay(10000);
}
```

Par simplicité je n'ai pas mis le prototype ni les fonctions PrintVariable() et setup(). N'oubliez pas Serial.begin(vitesse) dans

cette dernière. A l'exécution, on obtient le résultat suivant dans le moniteur série (toutes les 10 secondes) :

0x20000000

soit 0X20 suivi de trois fois 0X00, ce qui, lu à l'envers et en décimal donne bien 32.
On peut également utiliser la fonction sur un tableau d'éléments (un tableau est stocké dans la pile et se libère automatiquement une fois la fonction terminée). Voici un exemple :

```
void loop()
{
  byte tableau[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  PrintVariable((uint16_t)&tableau, sizeof(tableau));

  delay(10000);
}
```

Le résultat donne:

0x0001020304050607080 9

Imprimer le contenu d'un buffer

Imaginons maintenant que vous voulions imprimer le contenu d'un buffer et que nous ne sachions pas quelle taille il a. Nous allons utiliser la technique vue précédemment dans le schéma *Allocation mémoire dans le tas*, à savoir reculer de 2 octets pour obtenir la taille. Je propose ceci :

```
void loop()
{
  // On crée un buffer:
  uint8_t* buffer = (uint8_t*)malloc(10);
  // On initialise tous les octets à 0xa0:
  memset(buffer, 0xa0, 10);

  // Imaginons maintenant qu'on ne
  // connaisse pas la taille du buffer:

  // Adresse du buffer dans le tas:
  uint16_t address = (uint16_t)buffer;

  uint8_t* pointer = (uint8_t*)(address - 2);

  // Taille:
  uint16_t len = 0;
  // On copie la valeur à l'adresse -2 dans len:
  memcpy(&len, pointer, sizeof(uint16_t));

  pointer += 2;
  PrintVariable((uint16_t)pointer, len);

  free(buffer);    // On libère !

  delay(10000);
}
```

Si tout va bien, on devrait obtenir en résultat 10 fois 0xA0 ; *memset()* a initialisé chaque octet de notre buffer à cette valeur. Et effectivement, à l'exécution on obtient :

0xa0a0a0a0a0a0a0a0a0a0

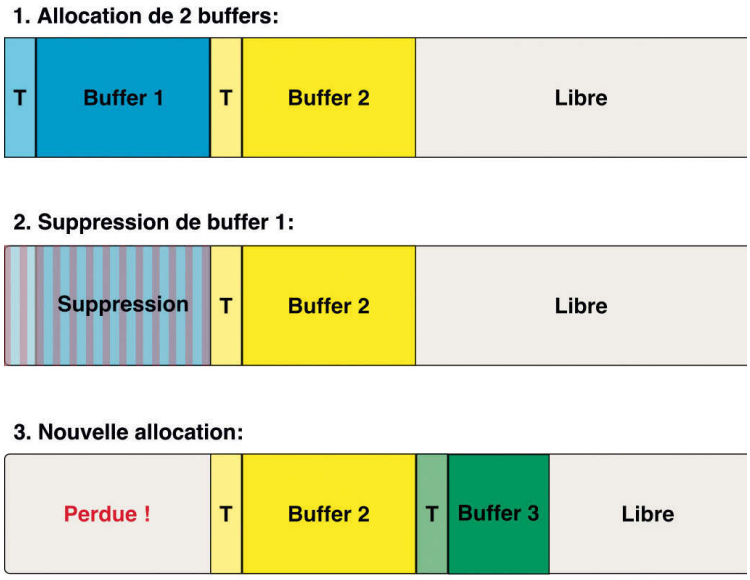
Obtenir la taille des différentes sections

Même si ce n'est pas très connu, on peut obtenir assez facilement la quantité de mémoire utilisée par chacune des sections en SRAM. Il existe des variables, à déclarer avec le mot-clé "extern", qui se chargent de nous donner les débuts et fins de chaque section. Avec juste un petit twist pour le tas : la variable *__heap_end* existe bel et bien mais sa valeur n'est pas fiable. Je propose donc la fonction suivante :

```
// Ecrit les taille de chaque section:
void PrintMemusage()
{
  extern char* __data_start;
  extern char* __data_end;
  extern char* __bss_start;
  extern char* __bss_end;
  extern char* __heap_start;
  // prochaine allocation du tas disponible:
  extern char* __brkval;

  uint16_t memReg = (uint16_t)&__data_start;
  uint16_t memData = (uint16_t)&__data_end - (uint16_t)&__data_start;
  uint16_t memBss = (uint16_t)&__bss_end - (uint16_t)&__bss_start;
  uint16_t memHeapTtl = ((uint16_t)__brkval == 0) ? 0 : (__brkval - __malloc_heap_start);

  uint16_t memStack = RAMEND - SP; // SP = pointeur de pile.
  uint16_t freeRam = SP - ((int)&__heap_start + memHeapTtl);

  Serial.println("Registres: " + String(memReg) + " octets");
  Serial.println(".Data: " + String(memData) + " octets");
  Serial.println(".Bss: " + String(memBss) + " octets");
  Serial.println("Tas1: " + String(memHeapTtl) + " octets");
  Serial.println("Pile: " + String(memStack) + " octets");
  Serial.println("Libre: " + String(freeRam) + " octets");
}
```

On voit que ces variables spéciales sont des pointeurs vers un type char (notez qu'elles commencent par 2 tirets bas et pas un seul). Leurs noms sont assez parlants, sauf pour __brkval qui est un pointeur vers le prochain emplacement disponible dans le tas : s'il vaut 0, alors il n'y a aucun octet alloué dans le tas. Sinon, il est forcément supérieur à __malloc_heap_start et on obtient la taille en lui retranchant cette valeur.
Pour info l'Arduino n'a pas de Garbage Collector : il écrit toujours la prochaine allocation à la fin du tas. Si on déclare 2 buffers, qu'on libère le premier mais pas le deuxième puis qu'on en crée un troisième, ce dernier n'ira pas dans l'espacement libéré par buffer 1, même si la place est suffisante. Cet espace est alors perdu ! 4
Enfin parlons de la macro *SP* : elle signifie *Stack Pointer* et renvoie l'adresse du dernier octet de la pile (donc le moins élevé). On s'en servira beaucoup plus dans le prochain article car on peut faire plein de choses sympathiques avec.
Testons tout de suite notre fonction avec le programme suivant qui

4



imprime, toutes les 10 secondes, les quantités de mémoires utilisées :

```
void loop()
{
  uint8_t* buff1 = (uint8_t*)malloc(10);
  uint8_t* buff2 = (uint8_t*)malloc(20);

  PrintMemusage();

  free(buff2);    // Libérée !
  free(buff1); // Délivrée !

  delay(10000);
}
```

Normalement nous devrions avoir un tas égal à la quantité de nos deux buffers + 2 octets chacun (pour la taille allouée), ce qui nous donne 34 octets. Voici le résultat :

Registres: 512 octets
.Data: 100 octets
.Bss: 677 octets
Tas: 0 octets
Pile: 17 octets
Libre: 7397 octets

Allô Houston, on a un problème ! Le tas est à 0 octets au lieu de 34. On a pourtant des valeurs plutôt crédibles pour le reste. Alors pourquoi ?
En fait la raison est très simple : nous avons été victimes des optimisations lors de la compilation. Nous avons alloué deux buffers, mais ne les avons pas utilisés puis les avons libérés. Le compilateur les a donc tout logiquement supprimés car ils n'avaient aucune utilité dans le programme !
Pour avoir le bon résultat, nous devons donc les utiliser d'une façon ou d'une autre avant qu'ils ne soient libérés. Je propose d'utiliser un simple *memset()* qui affecte une valeur quelconque à chaque octet du buffer :

```
void loop()
{
  uint8_t* buff1 = (uint8_t*)malloc(10);
  uint8_t* buff2 = (uint8_t*)malloc(20);

  PrintMemusage();

  memset(buff1, 0xe4, 10);
  memset(buff2, 0xe4, 20);

  free(buff2);    // Libérée !
  free(buff1); // Délivrée !
  delay(10000);
}
```

Résultat :

Registres: 512 octets
.Data: 100 octets
.Bss: 677 octets
Tas : 34 octets
Pile: 17 octets
Libre : 7363 octets

Beaucoup mieux ;) A noter que les optimisations du compilateur ne se limitent pas à cela : par exemple, s'il voit qu'une fonction n'est pas utilisée, il la supprime. Ou si elle n'est pas assez utilisée il peut carrément la transformer en fonction "inline", c'est à dire qu'il copiera le code de la fonction tel quel à l'endroit où elle est appelée (pas de branchement ni d'arguments ni de valeur de retour). Pour information il est possible, à la compilation par avr-gcc, de préciser le niveau d'optimisation (-Og, -O0 à -O3 ou -Os, classées de la moins agressive à la plus agressive). La configuration de ces options n'est pas directement disponible dans Arduino IDE, mais nous verrons un moyen de contourner cela dans le prochain article.

Conclusion

Nous avons vu comment manipuler la SRAM sur un Arduino : son fonctionnement, très similaire à celui d'un ordinateur classique, nous permet à l'aide d'une simple adresse en mémoire de récupérer les données que l'on souhaite. Nous avons également pu bidouiller pour obtenir la taille d'un buffer dans le tas, la taille des sections en mémoire et comprendre un peu mieux leur fonctionnement.
Nous verrons dans le prochain article comment obtenir les numéros d'instructions qui sont stockées en mémoire Flash, comment déterminer le numéro de l'instruction en cours et comment créer un mini debugger logiciel qui nous permettra d'arrêter l'exécution d'un programme à une instruction donnée.

•