

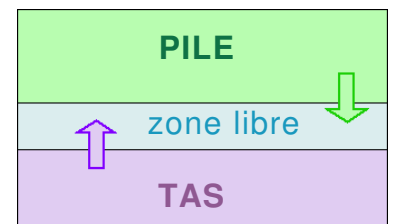
Collision entre la PILE et le TAS.

Par Nulentout le jeudi 27 Septembre 2024.

Bien que précisé à la fin de la liste des conseils pour programmer avec méthode, ce serait dommage dans notre cheminement expérimental de ne pas aborder ce sujet, car forcément un jour ou l'autre vous risquerez d'être la victime d'un tel phénomène. L'un des problèmes les plus vicieux qui puisse survenir lors du développement d'un programme, c'est la collision entre la **PILE** et le **TAS**. Il n'est pas question dans cet exposé d'analyser en profondeur le fonctionnement interne d'un microcontrôleur. On va se contenter du minimum minimorum.

Lorsque le programme fonctionne, il entasse dans une zone mémoire spéciale nommé le **TAS** les variables temporaires, comme les variables locales à une procédure, les paramètres passés par valeur etc. Simultanément, un pointeur d'adresse de retour des procédures et des interruptions entasse les adresses dans une autre zone nommée la **PILE**. La zone mémoire dédiée à ces deux fonctions est commune, et pour ne pas interférer elles sont situées aux "extrémités" de la RAM dédiée. Du coup le **TAS** se crée du bas vers le haut. La **PILE** au contraire ajoute ses valeurs du haut vers le bas. Plus il y a de données dans le **TAS**, plus il y a d'appels à procédures sans retour, et plus la **zone libre** (En bleu clair sur la Fig.1) qui sépare les deux antagonistes devient exigüe. Si vraiment la dynamique du programme exige trop de place simultanément dans ces deux zones, elles finissent par se superposer et il y a **écrasement de données vitales**. On désigne ce phénomène par **COLLISION DE PILE**. C'est un problème particulièrement sournois car brusquement le logiciel se met à avoir un comportement totalement imprévu, alors que l'on a à peine modifié un fifrelin le code SOURCE et que rien dans ce que l'on a changé ne peut justifier le marasme constaté. Par exemple on fait afficher "Bonjour". Puis on modifie par "Bonjour." en n'ajoutant qu'un point final à la chaîne de caractères. On relance le programme et PAFFFFFFFffffff, c'est du n'importe quoi. Soit le programme se met à afficher des incohérences, soit il se fige et on perd la main.

Fig.1



- **Brusquement un programme présente un comportement anormal,**
- **La séquence qui diverge n'a rien à voir avec les dernières modifications effectuées.**
- **Quand un tel comportement étrange se produit avec la certitude qu'il n'y a aucune relation entre la séquence anormale et les derniers correctifs apportés au logiciel, il faut diagnostiquer une collision de PILE, c'est la cause la plus probable du problème.**

➤ Prendre une assurance contre les collisions de PILE.

Par mesure de précaution, le programmeur devrait toujours faire un test de vérification du non risque de collision de **PILE** lorsque le développement de son programme s'achève et qu'il décide qu'il est pleinement opérationnel. La technique est la suivante :

1) On ajoute à la liste des procédures de servitude la fonction suivante :

```
//@@@@@@@@@ Ci-dessous fonction pour afficher la place disponible @@@@@@@@@@
int SRAM_LIBRE() { // Fonction qui retourne la taille de SRAM disponible.
    extern int __heap_start, *__brkval; // Déclaration des deux pointeurs dédiés.
    byte BIDON; // On ajoute une dernière variable allouée qui occupe le "haut" du TAS.
    if (__brkval == 0) {return (int) &BIDON -(int) &__heap_start;}
    else {return (int) &BIDON -(int) __brkval;}}
//@@@@@@@@@@@@@@@@***** @@@@@@@@@@
```

Noter au passage que tant que si l'étape ② qui suit n'est pas validée dans le programme SOURCE, la fonction **SRAM_LIBRE()** ne consommera aucun OCTET dans le programme OBJET. Cette fonction **SRAM_LIBRE** qui utilise les pointeurs a pour but de retourner la valeur de la place qui reste entre le **TAS** et la **PILE** c'est à dire dans la **zone libre** au moment de son appel.

2) On place en fin de la procédure **void loop()** la séquence :

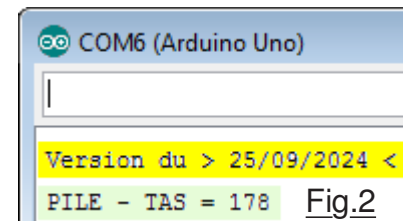
```
//@@@@@@@@ Ci-dessous code ajouté pour afficher la place disponible. @@@@@@@@@  
Serial.print("PILE - TAS = "); Serial.println(SRAM_LIBRE()); Serial.println();  
//@@@@@@@@ ***** @@@@@@@@@
```

Cette petite séquence fait afficher sur la voie série du **Moniteur** la valeur de **SRAM_LIBRE**. Si l'application est autonome, le résultat sera présenté sur l'afficheur utilisé par cette dernière.

NOTE : Je vous recommande très très fortement de ***toujours effectuer ce test quand vous venez de terminer un programme***. Pour un sketch qui consomme la presque totalité des 30720 OCTETS disponibles, on peut manquer de place pour loger ce test qui consomme environ 450 OCTETS de programme. Dans ce cas il importe alors de supprimer un ou deux appels à procédures (*Ce qui ne modifie pas l'évaluation*) pour faire provisoirement de la place. Une fois la marge de sécurité vérifiée, vous repassez la ligne qui invoque **SRAM_LIBRE** en remarque et rétablissez l'appel aux deux procédures. Affaire classée !

Invoquer du préventif dans ce domaine n'est pas une arme absolue. Ce n'est pas parce-que l'on a contracté une assurance que l'on n'aura jamais d'accident. C'est exactement pareil dans notre cas virtuel. En effet, quand le programme va fonctionner, il va entasser des données sur le **TAS** (*Variables locales, tableaux de données etc.*) et invoquer des procédures qui vont placer sur la **PILE** les adresses de retour. Si ses données dynamiques sont boulimiques, on peut fort bien saturer et dépasser la **zone libre** avec pour conséquence le PFFFFFFFFfffff ! Observons la Fig.2 qui présente l'aspect de ce que l'on obtient avec les ajouts ① et ② au logiciel. Dans la zone jaune le programme en cours de développement se présente. Puis dans la zone vert pastel le résultat est affiché. On peut donc conclure :

Sauf dans le cas particulier d'un programme qui utiliserait un nombre considérable de procédures qui s'invoquent les unes à la suite des autres sans retour, entassant un grand nombre d'adresses de retour dans la **PILE** et de données locales sur le **TAS**, *si* cet espace **zone libre dépasse les 150 OCTETS**, une grande expérience notamment en assembleur m'a montré que *le risque de collision est dérisoire*. Il ne faut pas oublier que si une collision de **PILE** doit avoir lieu, en principe elle surviendra durant le développement.



➤ Configurations qui produisent la collision de **PILE** avec le **TAS**.

Concrètement on oublie royalement qu'un nombre important d'interruptions se produisent en tâche de fond. Quelquefois un codeur rotatif génère des interruptions, sans compter les procédures **delay()**, les fonctions telles que **millis()**, la PWM ... une foule de ressources internes déclenche des interruptions. C'est transparent car c'est le compilateur C++ qui sur ces instructions fait sa cuisine interne. Arrive un moment, ou trop de données sont empilée sur le **TAS** et viennent écraser les adresses empilées. Puis le délimiteur '}' de fin d'une procédure ou d'une fonction demande au processeur de dépiler une adresse de retour. Comme cette dernière contient les résidus de la variable qui a "écrasé" les octets, le programme se "branche" strictement n'importe où. Étant alors sur du code objet incohérent, le comportement du logiciel devient totalement aléatoire.

PRÉVENTIF : Aucun programmeur n'est à l'abri d'une telle "catastrophe". Aussi, pour minimiser les risques il faut placer le minimum de chaînes de caractères dans le programme car en réalité elles sont placées sur le **TAS**. *Il faut également (Et surtout.) minimiser les tableaux.*

CURATIF : Quand se produit le **Scratchhh prouitchhh bom bring protchhhh !** c'est qu'il est trop tard. Nous avons placé plein plein plein de bavardages pour le dialogue Homme/Machine par exemple, alors que nous savons que ces "bla bla bla" sont entassés dans la mémoire dynamique. Notre démonstrateur comporte une foule de procédures et de fonctions qui passent des paramètres, sans compter les **for (byte l=1, ...)** qui ne sont pas gratuits. Les variables locales des boucles **for** doivent aussi être logées en RAM. *Enfin les tableaux sont gourmands en octets raison pour laquelle dès que l'on envisage des tableaux à plusieurs dimensions il faut rester prudent sur le nombre de leurs cellules et la taille des données dans ces dernières.*

REMÈDE : Dégager impérativement de la place sur le **TAS**, ce qui est plus facile à énoncer qu'à réaliser : S'aider des conseils donnés dans **CURATIF** ...

➤ *Pour celles et ceux qui désirent en savoir plus.*

Bien que la séquence `int SRAM_LIBRE()` soit compacte, elle n'est pas très évidente à comprendre car elle utilise la notion de pointeurs et fait appel largement à ces derniers. Ces pointeurs ont des identificateurs particuliers, la Fig.3 nous permet de les situer. Il nous faut ici "pénétrer dans la vie intime du fonctionnement des microcontrôleurs". Il serait hors de propos dans ces lignes d'étudier à la loupe l'agencement matériel de l'ATmega328 et d'en détailler finement le fonctionnement interne. Nous allons dans ce chapitre nous en tenir vraiment au strict minimum vital.

Fonctionnement de la mémoire vive **SRAM**.

La mémoire vive ($256 + 2Ko$) est globalement divisée en quatre zones bien distinctes :

- Les 256 premiers octets pour les registres généraux du microcontrôleur (*Représentée en jaune sur la Fig.3*) occupent "le bas" de la SRAM. (*En "assembleur" c'était la Page zéro.*)
- La zone nommée **BSS** qui contient toutes les variables globales, allouées statiquement au moment de l'édition de lien lors de la compilation. La **BSS** est utilisée par de nombreux compilateurs pour désigner une zone de données contenant les variables statiques définies dans les initialisations, et les déclarations avant **void loop()**.
- Le **TAS** sur lequel on entasse du bas vers le haut est destiné aux **allocations dynamiques** dans lequel on peut attribuer et libérer des blocs de mémoire. (*Nommé **HEAP***) Le **TAS** se fragmente généralement au cours de l'évolution du programme, (*Car une variable locale libérant de la place laisse "un trou" libre.*) avec un risque notable de le rendre inutilisable.

Défragmenter **HEAP** par une séquence de code de type "Ramasse miettes" est faisable mais relativement dangereux, car si l'on déplace une variable en cours d'utilisation, les conséquences peuvent s'avérer ingérables.

- La **PILE** nommée **STACK** mémorise temporairement :
 - * Les paramètres associés à l'appel des fonctions et procédures,
 - * **Les adresses de retour des fonctions et procédures,**
 - * Les variables locales aux fonctions et procédures.

La **PILE** est une zone de mémoire commençant en haut de la SRAM qui se charge vers le bas de façon linéaire et continue lors des appels des fonctions ou des procédures. Elle se réduit vers le haut lors des retours. Chaque appel à une procédure empile l'adresse de retour, (*Sauf cas particulier des procédures récursives.*) chaque retour la dépile et libère la place. Au cours du programme, si un grand nombre de données sont sur le **TAS** qui est "très haut", et que l'on enchaîne un grand nombre d'appels à procédures sans retour, (*Cas des sous-routines itératives par exemple*) il peut arriver que l'espace entre **PILE** et **TAS** devienne nul. C'est la collision et l'écrasement mutuel des OCTETS engendre un fonctionnement totalement imprévisible du microcontrôleur. Aussi, ***avant de considérer que notre programme est fiable, il faut impérativement vérifier que le risque de collision de PILE est dérisoire.*** L'expérience montre que lorsque toutes les initialisations de **void setup()** sont terminées, il est recommandé de ne pas avoir ***moins de 100 octets***, car le risque de collision par fragmentation de la zone devient exagéré.

(@) : Sur la Fig.3 les noms des divers pointeurs sont imposés par le compilateur de l'IDE.

