

Petit bras manipulateur : L'aspect logiciel.

Par Nulentout : Jeudi 12 Décembre 2012.

Toutes celles et ceux qui vont s'égarer dans cette présentation ne sont pas forcément des "fanas" de la programmation, raison pour laquelle le didacticiel sur la réalisation de l'ensemble n'est pas encombré par les développements informatiques. Toutefois, il n'est pas exclu que certaines et certains des visiteurs soient patinés par la programmation des produits Arduino et désirent en savoir plus pour éventuellement glaner des pistes nouvelles, voir modifier le programme que je vous propose. Les chapitres qui suivent vont présenter certains aspects du codage en langage C++ rencontrés lors de cette réalisation.

Probablement que les lecteurs concernés par l'aspect logiciel le seront pour le programme d'exploitation sur Arduino, mais également pour tout ce qui est relatif à la programmation en général, et désireront à ce titre utiliser l'intégralité des possibilités de leur imprimante 3D. Comme personnellement j'ai suivi ce chemin, j'ai étudié tout ce qui me tombait sous la main concernant le **gcode** que l'on soumet à nos machines pour imprimer. Pour que les opérateurs puissent intervenir directement sur le "code machine", des programmes tels que **Repetier-Host** mettent à disposition des utilisateurs un éditeur de texte agissant directement sur le "programme machine". Ayant globalement cerné les éléments d'interprétation du **gcode**, j'ai résumé l'ensemble des informations dans un petit livret nommé  **Le G_CODE des imprimantes 3D.pdf**. Pour réaliser ce petit document papier, vous imprimez les feuilles Recto/Verso. Vous les pliez en deux et vous agrafez le tout en quatre endroits sur la tranche. Vous vous enrichissez ainsi d'un petit manuel de douze pages qui encombre peu l'environnement de l'imprimante. Non seulement ce manuel explique en page **P6** comment utiliser l'éditeur de **Repetier-Host**, et surtout il propose en **P10** une application pratique pour démontrer la pertinence de consacrer un peu de temps à cet aspect de l'emploi de votre machine.

Détailler les 1200 lignes du code source de **P07_Programme_Complet.ino** ne serait pas pertinent dans ce projet, car le but n'est pas la programmation. Par ailleurs, le "Sketch" est très documenté par d'innombrables remarques, il se lit presque comme un roman. Par ailleurs, on trouve déjà une très grande source d'informations dans les diverses fiches d'utilisation du bras manipulateur, avec présentation des divers écrans, liste des commandes et des erreurs répertoriées, ainsi que les divers protocoles de mise en service et d'apprentissage. Et surtout, dans les fiches de maintenance nous avons *Occupation de la mémoire EEPROM*, *Détermination logicielle de la température*, associée à la fiche *Comportement du capteur de mesure thermique*, sans compter la fiche *Paramètres d'adaptation du logiciel au matériel*. Et pour finir *Problème de la boucle d'affichage sur OLED 1,3'*. Aussi, dans ce didacticiel de complément je me suis contenté d'aborder *quelques aspects du traitement graphique imposant l'usage d'un peu de trigonométrie pour coder le programme*.



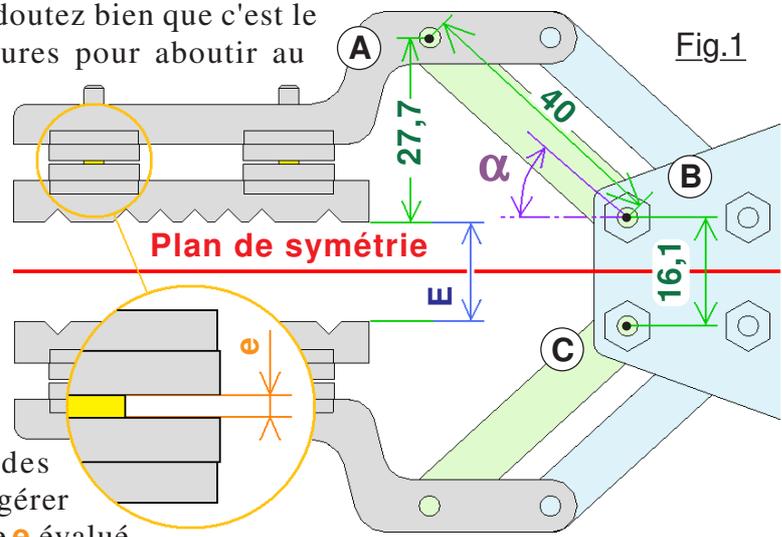
Ben Môa môa, je préfère de loin ma vie en rose naturelle, je suis bien plus souple. Dans cette version FLEXtruxPLA je me sens très vieille, et mes articulations craquent dès que je cherche à bouger un chtipeu !

Ouverture de la pince : Études "informatiques".

Commande inspirée des autres consignes moteur, 's' est prévue pour indiquer l'angle de positionnement de la bielle motrice par rapport au plan médian du corps de la PINCE. Il s'agit d'un paramètre "moteur" qui n'est pas spécialement convivial pour l'opérateur. Ce dernier préférerait indiquer l'épaisseur de la pièce à saisir par le bras manipulateur. C'est précisément le facteur déclenchant qui a motivé l'introduction de la commande 'S', le caractère étant suivi d'une valeur numérique précisant l'épaisseur de la pièce exprimée en mm. *Compte tenu du fait que l'on pilote les moteurs degré angulaire par degré angulaire*, il devient inutile d'indiquer l'épaisseur de l'objet saisi en précisant les fractions de millimètres.

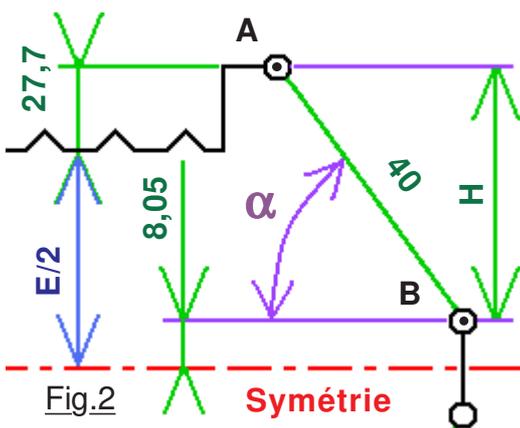
➤ Approche géométrique.

Avant même de tenter d'établir des équations et des formules propices à un codage en C++ il importe de déterminer les paramètres pertinents qui serviront à établir la représentation mathématique de la pince. Il y a quarante années, l'étude du bras manipulateur aurait été conduite sur une planche à dessin et au pantographe. (*Nostalgie ... Hips !*) Puis, pour développer cette étude nous aurions procédé un traçant des épures. Nous sommes en 2019 et le bras manipulateur, tant que l'on n'a pas mis à contribution l'imprimante 3D ne réside que dans les cellules binaires d'une machine informatique, alias l'ordinateur. Vous vous doutez bien que c'est le logiciel de **D.A.O.** qui a remplacé les épures pour aboutir au

croquis de la  sur laquelle figurent les paramètres géométriques qui vont conditionner les équations du modèle mathématique. (*D.A.O : Dessin Assisté par Ordinateur.*) En bleu clair on peut oublier, ces organes ne font que maintenir les mors en translation. Pour ne pas surcharger les servomoteurs, nous avons introduit de l'élasticité entre les mors par le truchement de ressorts de compression. Encore faut-il s'assurer que les bossages de centrage des ressorts ne viennent pas en contact. On doit gérer la machine de façon à ménager le petit espace **e** évalué à 1mm dans les études. Compte tenu des formes et des dimensions des mâchoires et des doigts, la distance entre l'articulation **A** et la surface active de la mâchoire est de 27,7mm. (*Cote déterminée par le logiciel de dessin.*) La distance entre les axes d'articulation **A** et **B** sur les biellettes est imposée à 40mm. Enfin la distance entre les deux axes **B** et **C** est de 16,1mm. (*Cette cote peut sembler étrange, un 16,0 aurait été plus "scolaire". Les diverses optimisations qui se sont enchaînées durant le développement conduisent à cette valeur ... que l'on conserve naturellement.*)

➤ Transposition mathématique.

L'étude qui va suivre consiste à "transformer" le visuel de la Fig.1 en équations mathématiques qui seront aisées à coder en langage C++. Résultant des conséquences engendrées par la compilation et la représentation des données en binaire, nous sommes contraints à traiter par "moitié"



en prenant en compte le plan de symétrie. Nous allons donc établir une représentation géométrique montrée sur la Fig.2 respectant ce critère. On y retrouve les dimensions du dessin précédent sauf qu'ici c'est l'épaisseur **E** divisée par deux qui figure sur le croquis. Concrètement, l'opérateur fournira en saisie la valeur désirée de **E** exprimée en mm. Le programme doit en déduire l'angle α de fermeture de la PINCE. C'est cet angle évalué en degrés angulaires qui sera imposé en consigne au moteur de la bielle motrice. La base géométrique pour effectuer les calculs est facile à traduire :

$$E = 2 * (8,05 + H - 27,7) = 2 * (H - 19,65)$$

➤ Un tantinet de trigonométrie.

L'équation encadrée est assez évidente, mais il nous reste à déterminer la valeur de **H** qui est fonction de la longueur de la bielle soit 40mm, et de l'angle α de positionnement du moteur. Heureusement pour nous, il y a belle lurette que les mathématiciens ont résolu ce problème qui relève de la trigonométrie qui comme son nom le précise s'occupe des figures géométriques à trois cotés, donc des triangles. Pour celles et ceux qui ont eu la chance d'effectuer des études en terminale, on nous y apprend que dans notre cas, **H** est égal à la longueur du coté multipliée par le *sinus* de l'angle α exprimé en radians. On peut donc le formuler par l'équation :

$$H = 40 * \sin(\alpha)$$

Dans cette formule **H** sera obtenu en mm à condition d'exprimer l'angle en radians. Ne nous prenons pas la tête pour passer des angles exprimés en degrés aux équivalents précisés en radians. La formule qui est bien connue se trouve déjà intégrée dans le compilateur d'Arduino. Exprimée en langage C++ cette relation serait codée : $H = 40 * \sin(\text{radians}(\alpha))$. Comme les limites du compilateur n'acceptent pas des lettres comme α , on utilisera à la place l'identificateur `Angle_Serrage_Pince` et le tour est joué. La formule initiale devient :

$$E = 2 * ((40 * \sin(\text{radians}(\text{Angle_Serrage_Pince})) - 19,65)$$

Nous avons bien progressé. Toutefois il reste un petit yatus. En effet, l'opérateur va indiquer la valeur de **E**, et le logiciel devra en déduire la valeur de `Angle_Serrage_Pince`. Il faut donc "triturer" cette équation pour que `Angle_Serrage_Pince` en devienne l'inconnue à calculer.

➤ On retourne la chaussette.

Transposer une formule de ce type relève du niveau de terminale, voir de première. Bref, on utilise nos savoirs dans ce domaine relativement banal, on fait passer des termes d'un côté, on repousse d'autres à droite, à gauche, on transpire un max, est couci-couça, très fier de notre performance, on arrive enfin à la formule sympathique suivante :

$$40 * \sin(\text{radians}(\text{Angle_Serrage_Pince})) = (E / 2) + 19,65 \text{ soit encore}$$

$$\sin(\text{radians}(\text{Angle_Serrage_Pince})) = (E / 80) + 0,49125 \text{ (Si, si, je vous l'assure !)}$$

- *Narf narf narf*, c'est pas ce que l'on veut. Le *sinus* de l'angle on s'en tamponne le coquillard avec un marteau à bomber le vitres. Ce que l'on veut c'est l'angle qui correspond à ce *sinus*.

C'est là que notre Professeur de mathématique se réjouit :

- *Et oui Dudule, si tu avais bien suivi mes cours tu saurais que c'est la fonction réciproque !*

- *Heueueueu, oui m'sieu ... comment on la trouve cette fonction réciproque s'il vous plait ?*

- *Ben elle se nomme arc sinus tout simplement et le compilateur sait la calculer ... lui !*

- *Hé bé ... on en a de la chance, il nous sauve la mise le compilateur en question.*

Pour obtenir l'arc correspondant à un *sinus* on utilisera `asin()` la fonction dédiée. Elle retourne la valeur de l'angle en radians, il faut donc encore retourner la chaussette et *transposer* en inverse *des radians en degrés angulaires* ce que fait à notre place la fonction `degrees()` du compilateur C++.

Codée pour Arduino nous avons enfin l'instruction idoine :

$$\textcircled{1} \quad \text{Sinus} = (\text{Valeur_Numerique} / 80) + 0.49125;$$

$$\textcircled{2} \quad \text{Angle_Serrage_Pince} = \text{degrees}(\text{asin}(\text{Sinus}));$$

Dans l'instruction $\textcircled{1}$ on commence par calculer dans la variable de type `float` le *Sinus* de l'angle dont ensuite on déterminera en $\textcircled{2}$ la valeur de l'*arc* correspondant. Dans l'instruction $\textcircled{1}$ l'identificateur `Valeur_Numerique` contient la valeur numérique qui accompagne la commande '**S**' que nous avons donné au programme par l'entremise du moniteur série USB de l'**IDE**. Si vous consultez le programme complet qui anime le bras manipulateur, vous pourrez vérifier que certaines précautions viennent compléter ces deux instructions. On commence par vérifier que l'écart désiré entre les deux mors soit bien compatible avec l'architecture de la pince, c'est à dire compris entre 1mm et 42mm. Ensuite, la valeur déterminée pour *Sinus*, compte tenu des contraintes résultant des divers traitement binaires peut conduire à une grandeur supérieure à un. Un test corrige ce cas ennuyeux. Dans l'équation $\textcircled{1}$ on oblige le compilateur à retourner une donnée codée sous forme d'un `float`. Enfin, en $\textcircled{2}$ on impose de prendre la valeur absolue du résultat du calcul pour avoir toujours un angle positif. Moyennant d'avoir contourné toutes ces petites tracasseries, notre programme fonctionne bien.

➤ **Enfin les résultats.**

P réambule au développement de ce petit projet, il a été décidé tout en amont d'exprimer les consignes moteur en leur imposant des positions angulaires exprimées en degrés. Pour diverses raisons, tant du point de vue pratiques, que pour simplifier le logiciel, les positions angulaires exprimées le sont par des entiers. Donc *on ne précisera pas des fractions de degrés*. Du coup, nous allons exprimer l'écart entre les mors en mm "entiers" également. De cette valeur le programme calcule l'angle qui en résulte. On se doute qu'exprimé en degrés entiers, la valeur réelle de la distance séparant les mors sera différente de celle que l'on désire. Le tableau Fig.3 le montre bien :

E souhaité	Angle calculé	Ouverture réelle
1mm	30°	2,0mm
2 ou 3mm	31°	3,2mm
4mm	32°	4,4mm
6mm	34°	6,7mm
10mm	38°	11,3mm
20mm	47°	20,5mm
30mm	60°	31,3mm
35mm	68°	36,2mm
38mm	75°	39,3mm
40mm	82°	41,2mm
41 ou 42mm	90°	42,0mm

À partir de 75° la variation d'ouverture est de plus en plus faible

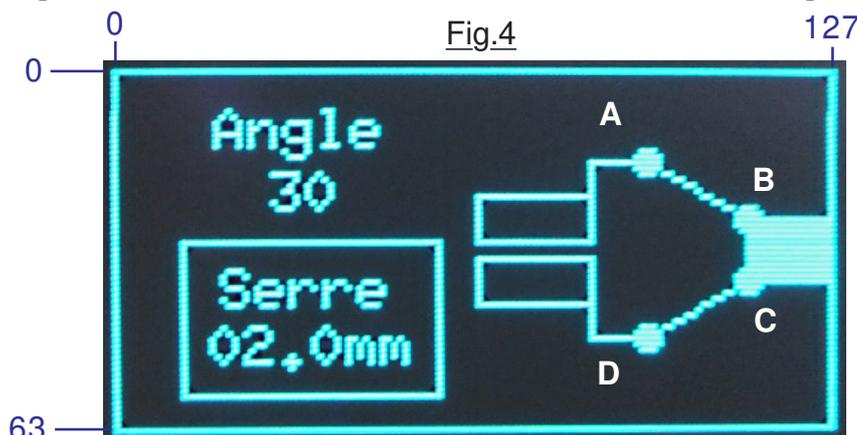
Fig.3

Pleine ouverture.

Force est de constater que la valeur réelle obtenue est légèrement supérieure à celle demandée. Comme l'angle qui la génère est connu, on peut avec la commande 's' tenter la valeur immédiatement inférieure. Par exemple pour une ouverture désirée de 30mm on teste 59°. La distance devient 30,6 ce qui est meilleur. On peut tenter encore un degré de moins. Pour 58° on obtient un serrage de 29,8mm. C'est la valeur que l'on retiendra pour ce cas de figure. Notez que pour affiner au mieux l'écartement, nous sommes amenés à effectuer plusieurs ajustements. C'est donc pour simplifier les manipulations que majuscule est employé pour définir l'écart, et minuscule pour le ou les ajustements, car ne pas avoir à passer en majuscule au clavier simplifie le travail à l'opérateur.

L'affichage graphique de la configuration de la pince.

I nformatiquement nous allons voir que l'on reprend les mêmes et on recommence, façon comme une autre de préciser que l'on va encore titiller de la trigonométrie. Les deux commandes conditionnant le serrage, c'est à dire 's' et 'S', imposent un écran symbolisant la configuration qui en résulte pour la pince. Bien que le symbole ne soit pas à l'échelle des dimensions pour des raisons de manque de définition sur la matrice de points, on respecte avec rigueur l'orientation des deux bielles. Du coup, pour les tracer, nous sommes confrontés au problème qui consiste à déterminer la position de l'articulation extérieure en fonction de l'angle α . La Fig.4 définit le dessin que l'on désire tracer à l'écran. Pour cette analyse on ne compte plus les dimensions en mm, mais en PIXELS. Sachant que la mosaïque de l'écran fait 128 point de largeur sur 64 de hauteur, on doit choisir une représentation qui tiendra compte de ces limites. Pour des raisons esthétiques, on diminue "notre espace vital" en décidant de tracer un cadre tout le tour pour mettre en valeur notre beau tableau.



L'image est symétrique par rapport à l'axe horizontal.

En bleu sont précisées les coordonnées des PIXELS sur la mosaïque graphique.

B : x = 112 ,y = 26

C : x = 112 ,y = 37

La longueur des biellettes sur l'écran est choisie à 20 PIXELS.

➤ **Tracer le symbole des biellettes et des mors.**

Géométriquement les deux articulations **B** et **C** ont des centres de positions invariantes sur l'écran. Pour tracer les deux autres extrémités **A** et **D** il faut en déterminer la position dans la mosaïque des luminophores. (Les doigts seront également représentés à partir de ces points représentatifs **A** et **D**.) Pour calculer les coordonnées de **A** et **D** il faut encore faire appel aux notions de base de la trigonométrie. La réponse est totalement analogue à celle qui était présentée en Fig.2 sauf que certaines dimensions changent. On copie sans vergogne la Fig.2 et sur son clone en Fig.5

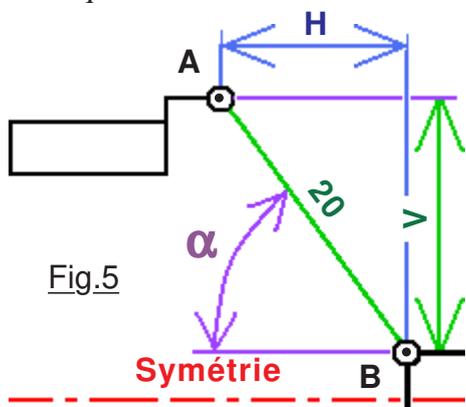


Fig.5

on se contente de modifier les valeurs par celles du "dessin électronique". On retrouve des formules issues de la trigonométrie à savoir :

$$V = AB * \sin(\alpha) \text{ et } H = AB * \cos(\alpha)$$

Dans cette formule, **AB** représente la longueur graphique du symbole des biellettes exprimée en PIXELs, on doit donc remplacer par la valeur **20**. Enfin, pour exprimer les coordonnées à l'écran des points **A** et **D** il faut calculer à partir du point **B** et du point **C**. Pour le point **A** par exemple on obtient :

$$\begin{aligned} xA &= xB - H = xB - AB * \cos(\alpha) = 112 - 20 * \cos(\alpha) \\ yA &= yB - V = yB - AB * \sin(\alpha) = 26 - 20 * \sin(\alpha) \end{aligned}$$

➤ **Codage des formules en C++.**

Exactement comme c'était le cas pour calculer l'ouverture en fonction de l'angle des biellettes, nous allons forcément utiliser des fonctions similaires accompagnées des problèmes de représentation binaire des données manipulées dans le microcontrôleur. Le codage des instructions en langage C++ sera par voie de conséquences très similaire :

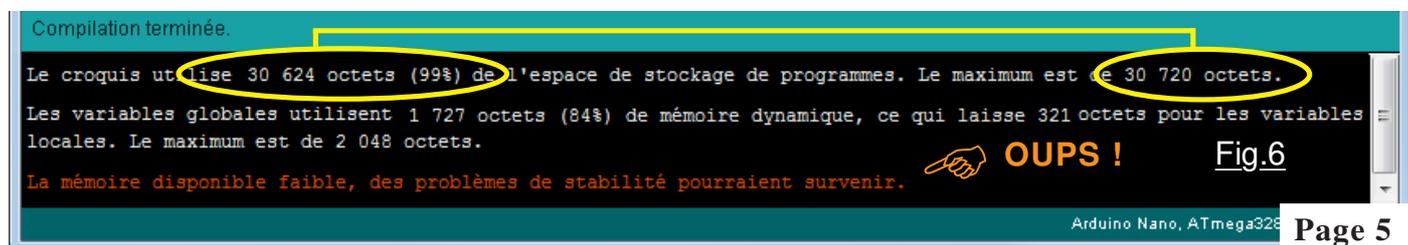
- ① `Position = 112 - abs(20 * (cos(radians(Angle_Serrage_Pince))));`
- ② `Hauteur = 26 - abs(20 * (sin(radians(Angle_Serrage_Pince))));`

Dans l'instruction ① l'identificateur **Position** représente **xA** de la formule, alors que dans la ligne ② l'identificateur **Hauteur** est relatif à **yA** de l'expression mathématique. Les deux variables locales à la procédure d'affichage **PAGE7()** nommées **Position** et **Hauteur** sont de type **byte**. Ce sont donc des entiers au sens des mathématiques, donc directement utilisables pour tracer des éléments avec les méthodes de la bibliothèque "U8glib.h" qui gère l'écran graphique OLED. Le calcul d'un **sinus** ou d'un **cosinus** retourne une donnée de type **float**. Dans les instructions ① et ② c'est le compilateur qui sans le dire effectue le codage pour avoir une compatibilité de représentation binaire dans l'ATmega328. Enfin, quand on fournit à la bibliothèque les coordonnées des centres des cercles et des segments de droite, c'est cette dernière qui se charge d'allumer ou d'éteindre les divers points en gérant les "arrondis" de position. En résumé, c'est le compilateur C++ assisté des méthodes de la bibliothèque "U8glib.h" qui fait tout le travail.

Des préoccupations purement informaticiennes.

Avouons que peu de programmeurs sur Arduino savent qu'en "roulant avec un taux d'Octet trop important dans le programme", on risque une **collision entre la PILE et le TAS. ScratchhhhBOUM et le microcontrôleur diverge**. Dans cet ultime chapitre nous allons aborder ce thème dont les programmeurs ne devraient jamais faire l'économie, surtout quand ils ont agencé un logiciel "volumineux".

Compilons la dernière version du programme d'exploitation. **GLUPS ... il ne reste plus que 96 octets de disponibles pour le programme**. Du reste le compilateur ne se sent pas à l'aise, il nous avertit par le message de la Fig.6 de couleur orange. Pour faire court : Dans l'intégralité des zones mémoire disponibles



de l'ATmega328 il ne reste non utilisé que 16 octets en EEPROM, 96 pour le programme et comme indiqué 321 octets entre la PILE et le TAS. Autant dire que *Dudule* ne pourra pas nous dire que l'on gaspille des ressources. On peut certifier que le taux d'occupation est presque déraisonnable. Pour circonstances atténuantes j'invoquerais le fait qu'avec 321 octets entre la PILE et le TAS le programme devrait se montrer stable, cette affirmation péremptoire est issue d'une longue expérience avec les microprocesseurs. Donc, si un jour on doit faire du vide car une modification du programme impose plus d'octets que de place disponible, il faudra songer à "pousser les murs", c'est à dire supprimer quelques fonctions estimées secondaires au regard de ce que l'on désire ajouter.

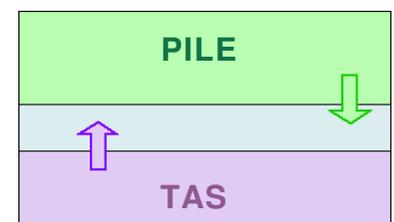
➤ **Abuser n'est pas gratuit ... ça peut faire mal.**

Quand on sacrifie presque toute la place disponible dans les ressources dynamiques du microcontrôleur aux exigences de rentabilité, arrive forcément un moment où le fil se casse. Dans le domaine de la programmation, cette "exagération" peut conduire à ce que les informaticiens nomment "une collision de PILE. Ce n'est pas spectaculaire, c'est même d'une discrétion presque totale. On change une petite virgule dans le source, un deux fois rien, et, brusquement le programme commence à avoir un comportement complètement incompréhensible. Examinons ce qui se passe, et surtout comment vérifier que ça n'arrivera pas.

Vérifier que l'impensable ne va pas se produire.

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 (En bleu clair sur la Fig.7) qui sépare les deux antagonistes devient exigüe. Si vraiment la dynamique du programme exige trop de place simultanément dans ces deux zones, elles se superposent et il y a écrasement de données vitales. On dit alors qu'il y a **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. 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.

Fig.7



Particulièrement agassif, vous allez y passer le réveillon. (*Façon de parler, car franchement au changement d'année je peux vous assurer que l'ordinateur est au chômage !*) Vous aurez un mal fou à comprendre, car en toute logique n'ajouter qu'un modeste point final dans un texte affiché n'a aucune raison de perturber le déroulement d'un programme. Donc si un jour un tel incident se produit, pensez à la **COLLISION DE PILE**.

➤ **Surveiller la COLLISION de PILE.**

Généralement, lorsque je finalise un programme "cossu", je sais par expérience que la PILE et le TAS sont très sollicités. On peut parfaitement imaginer que la combinatoire des cheminements dans les séquences de code n'ont jamais engendré de collision de PILE même si parfois on a sans le savoir "frisé la correctionnelle". Il n'est jamais prouvé qu'en utilisation normale, avec une marge parfois limite, que dans une configuration particulière TAS ou PILE débordent d'un ou deux Octets. C'est la sanction imparable. Aussi, pour considérer qu'un programme est fiable à ce point de vue, avant de le valider, une bonne pratique consiste à effectuer une mesure de la "marge" qui existe entre le TAS et la PILE. On ne considèrera que le risque de collision est écarté que si la marge calculée par une petite séquence particulière dépasse les 150 Octets. Ce n'est pas une preuve à proprement parler, mais une sorte de principe qui depuis que je programme n'a jamais été remis en cause. Voici comment procéder :

➤ **Surveiller la COLLISION de PILE.**

Techniquement, l'approche consiste à activer une séquence qui mesure la place restante en RAM dédiée quand le programme a effectué toutes ses initialisations. Il a ainsi "entassé" toutes ses variables et le **TAS** atteint probablement la hauteur maximale. On fait afficher la taille en Octets qui reste encore disponible pour la **PILE**. Si cette zone est inférieure à 150 octets je considère que ce n'est pas suffisant. Une longue recherche d'optimisation du code source est alors engagée pour faire "maigrir" les exigences de place imposée sur le **TAS** et sur la **PILE**. (*Cette optimisation exige une bonne expérience en programmation.*) Lorsque les résultats obtenus sont satisfaisants, on neutralise les séquences qui servent à cette vérification pour alléger le programme et libérer le maximum de place mémoire. Je dois avouer que pour cette application, la manipulation est loin d'être redondante, car avec 99% de zone programme utilisée, on se doute que le TAS et la PILE sont plus que sollicités. De plus, pour que la séquence ajoutée puisse contenir dans "ce qui reste", il faut passer **case 'L'** en remarque dans **void Traite_Consigne()** pour avoir assez de place. Action :

En tête de programme on trouve à la fin de la procédure **void setup()** la séquence suivante :

```
//@@@@@@@@@@@@@@@@ Ci-dessous code ajouté pour afficher la place disponible. @@@@@@@@@@@@
// u8g.firstPage(); do {u8g.setPrintPos(22, 35); u8g.print("PILE - TAS = ");
// u8g.print(SRAM_LIBRE());} while(u8g.nextPage()); while(!Serial.available());
//@@@@@@@@@@@@@@@@*****@@@@@@@@@@@@@@@@
```

Les deux lignes de commentaire qui encadrent le code contiennent des chaînes de caractère du genre @@@@@@@@@@@@ pour pouvoir facilement les repérer dans le listing. Vous noterez que le code source colorié en orange est ignoré par le compilateur car transformé en remarque par le // placés en début de ligne. Suivant directement la séquence d'initialisation et juste avant **void loop()** on trouve :

```
//@@@@@@@@@@@@@@@@ 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; // Dernière variable allouée, donc occupe le "haut" du TAS.
// if (__brkval == 0) {return (int) &BIDON -(int) &__heap_start;}
// else {return (int) &BIDON -(int) __brkval;} }
//@@@@@@@@@@@@@@@@*****@@@@@@@@@@@@@@@@
```

Cette séquence est comme pour la précédente entièrement neutralisée par des // placés en début de ligne. Lorsque l'on désire faire une mesure de la place qui reste actuellement disponible entre la **PILE** et le **TAS** il suffit de valider ces lignes de code. Le programme démarre normalement et l'écran tout noir affiche au centre la valeur de l'information attendue. C'est précisément la valeur de la zone disponible. Le programme attend que l'on valide une touche avec le Moniteur USB pour passer en exploitation, nous laissant ainsi le temps de lire la valeur. Actuellement, la validation de cette séquence annonce un espace confortable de 252 octets pour mon programme spécifique. Sur le vôtre il y aura plus que ça. Il n'y a donc pas de "plantage vicieux" trop à craindre, même avec la dernière version "abusive".

L' introduction de ce code de servitude qui ne concerne que le programmeur consomme 140 octets de programme et 24 octets de mémoire dynamique. C'est du "code parasite" qui glotonne inutilement des ressources du microcontrôleur. Le laisser serait impertinent si la marge de place disponible est limite, car ces séquences viennent encombrer la mémoire dédiée. Par ailleurs, avoir à sauter cette phase à chaque RESET manque de convivialité. Aussi, chaque fois que ces séquences viennent encombrer un croquis quel qu'il soit, il est plus que logique de les neutraliser en les transformant en remarque en rétablissant les //. Notez que la page écran temporaire montrée en Fig.8 est scandaleusement incongrue. En effet, on est supposé craindre un manque de place entre la **PILE** et le **TAS**. Et pour calculer cette dernière on affiche bêtement en **A** le texte "PILE - TAS =" alors que seul présente de la pertinence le nombre **B**. Aussi, vous avez bien compris que ce luxe stupide est présent dans le code source uniquement parce-que le programme a montré qu'il restait presque de l'espace à revendre. Dans un contexte de "vaches maignes", il ne faudrait surtout pas faire afficher ce texte ou ... risque de COLLISION !

