Logiciel d'exploitation de la petite station solaire scientifique.

Par Nulentout : Vendredi 1ier Mars 2019.

Pour celles et ceux qui veulent confier "une bouteille à la mer" : ☞ michel.droui@laposte.net

Chère lectrice, cher lecteur, si vous ouvrez ce document, et surtout si vous en entreprenez sa lecture, c'est que pour une raison ou une autre vous vous sentez directement concernés par la programmation de la petite carte Arduino NANO. Le programme source BOLOMETRE.ino est considérablement documenté, ne serait-ce que pour me permettre le cas échéant, quand dans quelques temps j'aurais entièrement oublié tout ce qu'il contient, d'en reprendre l'édition en vue de corriger un "bug" ou d'en améliorer le comportement. Les innombrables lignes contenant des remarques du type // Texte d'explication seront les bienvenues. Le livret NOTICE technique du BOLOMETRE.pdf dans les chapitres regroupés coté Maintenance informatique me serait probablement suffisant pour reprendre le code et remémorer mes intentions initiales. *(Le livret plus la fiche qui commente la commutation des deux transistors T1 et T2.)*

Encore que ! Certaines séquences sont "inattendues", et la lecture des lignes de code source qui les concerne reste absconde. Certains choix ont été le fruit de longues errances dans les méandres de la mise au point. Enfin, le désir d'intégrer de nouvelles fonctionnalités ou options, dans un logiciel qui sature depuis longtemps la place disponible dans l'ATmega328, ont abouti à un historique "fastidieux" conduisant à des "patchs" et "repatch" qui s'entassent dans un fichier source qui frise les 950 lignes. Aussi, ce document qui n'a rien d'indispensable pour celles et ceux qui ne sont pas les victimes du démon de la programmation, je vous propose épars et pas forcément structurés, quelques développements que je crois ... au final bien utiles. Du reste, comme certaines séquences ne sont pas faciles à comprendre, tout au moins l'approche qui a été la mienne, je crois pouvoir vous avouer qu'un jour ou l'autre je serai certainement content d'avoir créé ce document dans lequel l'historique justifie pas mal d'éclaircissements. Comme le précise la petite parenthèse proposée ci-dessous, un logiciel qui ne contient strictement aucune erreur est possible, et à la portée d'un tout débutant. Ce n'est probablement pas le cas de BOLOMETRE.ino qui un jour où l'autre va nous titiller malicieusement. Que la force soit avec nous ...

Un logiciel qui ne comporte aucune erreur au sens mathématique existe. Il ne comporte strictement aucune instructions exécutables sauf celles qui délimitent le début et la fin des deux procédures void setup() et void loop(). Composé uniquement de remarques, quand il est compilé il est d'une compacité admirable puisque ne consommant que 450 octets de programme et 9 octets en mémoire dynamique. Il fonctionne à la perfection et présente une fiabilité totale. La seule faiblesse de cette merveille ... son rendement strictement nul, c'est Totoche qui ne va pas être content !

Ben Môa môa, j'ai pas pigé pourquoi le message Internet doit être mis dans une bouteille qui sera balancée dans l'océan !
La houle va plus rapidement que la toile informatique ? ? ?

1) RESET froid ou P.E.P.

Informatiquement, le **P**anic **E**ntry **P**oint est un point particulier dans un gros logiciel vers lequel sera redirigé le programme lorsqu'un aléas de fonctionnement est détecté et place le système dans une configuration critique. Les ingénieurs logiciel prévoient une telle vectorisation par raison de sécurité. Globalement, le P.E.P. est placé après le démarrage qui initialise toutes les variables et les périphériques.

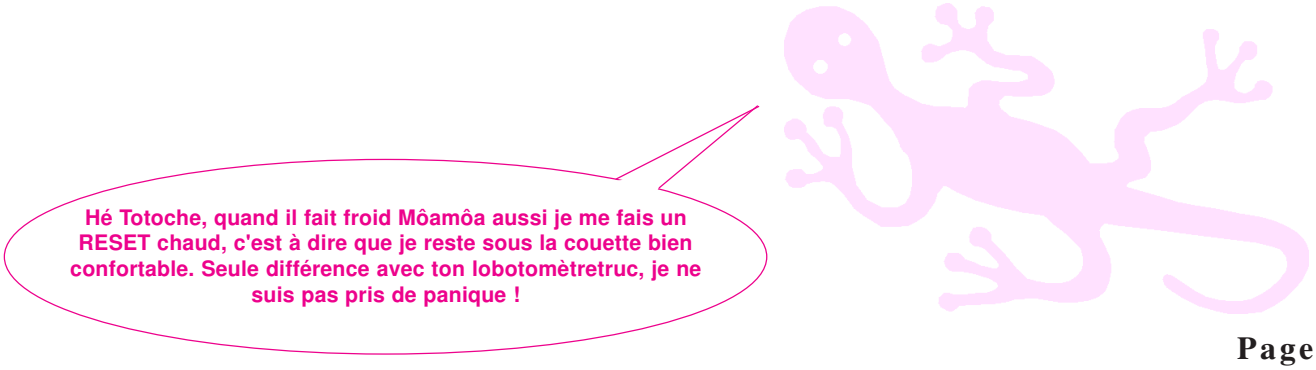
➢ Problème du RESET intempestif sur le bolomètre.

D'une façon générale, un système informatisé classique, c'est à dire alimenté par le secteur et sans source de secours déclenche un RESET intempestif lorsque se produit une coupure accidentelle d'énergie. Actuellement, de telles coupures secteur sont relativement épisodiques, et ce n'est pas souvent qu'un tel incident se produit, à tel point que sur les petites installations on observe de moins en moins de "blocs de sauvegarde". Pour notre petit appareil électronique, des RESETs "intempestifs" peuvent potentiellement se déclencher à la fin de chaque nuit en hiver, quand la journée qui précède n'a pas rechargé suffisamment les accumulateurs. S'il y avait redémarrage "standard", le logiciel afficherait la page d'accueil et attendrait un clic sur le clavier. De ce fait, quand potentiellement ce type de comportement serait probable, il y aurait obligation tous les matins, à ce que l'opérateur aille vérifier s'il faut éventuellement "relancer la machine". Cette contrainte serait à l'encontre de la qualité opérationnelle de notre Bolomètre. Aussi, il nous faut prévoir deux sortes de redémarrage en fonction des circonstances :

- Démarrage à froid : C'est en quelques sortes un RESET standard, avec la page d'accueil qui précise la version du logiciel, la possibilité de réinitialiser toutes les données avec BP+ long etc.
- Démarrage à chaud : C'est précisément le P.E.P. dans notre cas. Pour que les enregistrements des données puissent se faire dès que la carte Arduino est suffisamment alimentée, un indicateur sera positionné manuellement par l'opérateur sous forme d'un inverseur imposant un état logique sur l'entrée binaire **D8**. Le programme sur un redémarrage consultera ce "booléen" matériel, et, en fonction de son état réalisera toutes les opérations d'initialisation, ou sautera les séquences spécifiques à la mise en service sur site de l'appareil.

Ouvrez le petit livret NOTICE technique du BOLOMETRE.pdf en **p7**. L'organigramme de la Fig.14 détaille le déroulement des instructions lorsque se produit un RESET. C'est le test situé en (3) qui consulte l'état de l'entrée binaire D8 pour vectoriser sur les initialisations de base, ou pour sauter ces dernières dans le cas d'un RESET "chaud" de type P.E.P.

NOTE : Il importe de relativiser les conséquences d'un tel redémarrage intempestif. En effet, lors de la pire des journées hivernales, lorsque l'énergie collectée confine à une peau de chagrin, les accumulateurs contiendront probablement assez d'énergie pour le fonctionnement jusqu'à la transition. Ainsi, les données de la journée qui vient de s'écouler ne seront pas perdues, et sauvegardées en mémoire EEPROM de l'ATmega328. Si le lendemain il faut plusieurs heures de soleil pour arriver à redémarrer, ces dernières n'auront pas été enregistrées en tant qu'énergie fournie par les panneaux solaires. Peu importe, car cette dernière représente un pourcentage dérisoire au regard du cumul pour la semaine, et encore moins pour l'année. Alors considérer que cette énergie non mesurée est nulle ne fausse pas la pertinence des données collectées.



2) Mise en œuvre du mode SOMMEIL.

Figurez-vous que la version primaire du programme était testée début janvier. BRRRRRrrrrrr, non seulement il fait froid, mais la météorologie est ... hivernale, c'est à dire avec un ciel gris uniforme toute la journée. Dans ces conditions, l'énergie engrangée dans les accumulateurs est très insuffisante pour assurer les 25mA de courant consommé par l'électronique jusqu'au lendemain. Du coup ... trop de RESETs et de données non enregistrées. C'est alors qu'est née l'idée de ***mettre le processeur en Sommeil*** durant la nuit. *(Se reposer la nuit est un droit fondamental pour tous !)*
Le mode SOMMEIL n'est pas foncièrement facile à émuler. Il faut définir les horaires durant lesquels le processeur sera mis en faible consommation, et ceux qui seront pris en compte pour le réveiller. Les plages d'endormissement sont saisonnières, il faut donc définir une notion "locale" d'Été et d'Hiver autre que celle légale, sans compter que durant la nuit, il faut gérer des urgences.

➢ Ne dormir que d'un œil.

Concrètement, il faut bien un moyen de réveiller le processeur quand ce dernier est en sommeil. En interne, une foule de circuits sont débranchés d'où l'économie d'énergie. Il ne se passe plus rien, car dans ce mode le "compteur ordinal" ne s'occupe plus d'aller chercher les instructions, le programme est "oublié". Sans entrer dans les détails, un chronomètre interne est initialisé à une valeur décidée par le programmeur. La plus grande possible est d'environ huit secondes, c'est celle adoptée dans notre programme. Ce chronomètre particulier continue à fonctionner et décompte. Quand il arrive à zéro, il réveille le processeur qui reprend alors ses activités.

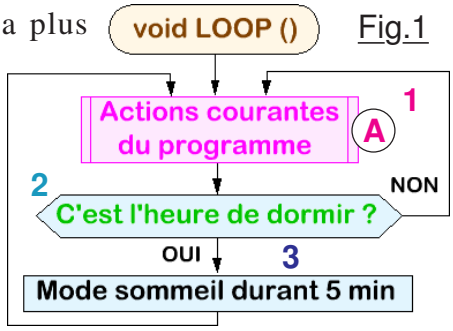


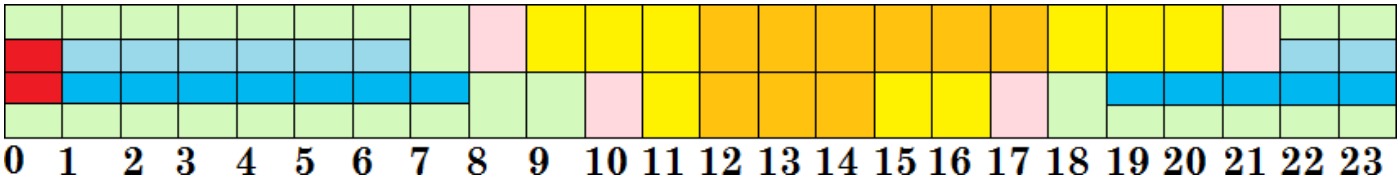
Fig.1

L'idée de base est symbolisée sur l'organigramme de la Fig.1 qui montre le déroulement global d'un tel processus. Tant que le test en **2** est négatif, on boucle dans la branche **1** et le programme se déroule de façon banale. Si le test détecte que nous sommes la nuit aux horaires de repos, alors c'est la branche **3** qui sera suivie. En entrée dans l'action particulière, le processeur va s'enfermer dans une séquence durant laquelle pendant cinq minutes il va sommeiller ne titillant que 14mA dans les réserves. Puis, le programme va reboucler. Tant que les "conditions nocturnes" seront satisfaites, la branche **3** sera parcourue et la carte Arduino consommera un minimum de courant pour préserver les accumulateurs. Il importe de savoir que l'action réalisée en **A** ne dure qu'une seconde, donc durant la nuit, l'ATmega328 va passer le plus clair de son temps à dormir.

➢ C'est quand la nuit pour dormir ?

Initialement nous pourrions imaginer qu'il suffit de laisser au repos le processeur durant toute la période nocturne pour laquelle le panneau solaire ne délivre pas d'énergie. Malheureusement les choses ne sont jamais simples. Première difficulté : Pour savoir si l'on se trouve "la nuit", le programme va utiliser, c'est une évidence, l'Horloge/Calendrier. Mais déjà il faut savoir que ce programme est réalisé en 2019, c'est à dire que nous continuons à vivre avec l'heure d'été et l'heure d'hiver. Le Soleil pour son compte ignore sereinement nos préoccupations horaires. Pour mieux comprendre la difficulté, considérons le petit dessin de la Fig.2 qui symbolise sur la moitié du haut une journée estivale, et sur celle du bas sa complémentaire hivernale. Les chiffres situés sur le dessous représentent la valeur du début de l'heure représentée, c'est à dire pour la

Fig.2



première colonne de 0H à 0H59, deuxième colonne de 1H00 à 1H59 etc. Ce sont les heures légales qui seront fournie par le petit module interne au Bolomètre. Les plages horaires sont coloriées en fonction du flux énergétique. Par exemple, en été, entre 8H00 et 9H, colonne coloriée en rose, le panneau solaire ne collecte que peu d'énergie. En jaune, la situation s'améliore. En orange, le flux est suffisamment intense pour avoir à dissiper du courant dans les résistances de charge. N'oublions pas que lorsque l'horloge indique 17H en été, en réalité au

Soleil il n'est que 15H et ce dernier ne s'est pas encore beaucoup décalé du méridien, raison pour laquelle le secteur est colorié en orange.
Moitié inférieure, c'est l'hiver et le Soleil n'arrive pas à se hausser dans le ciel. Aussi, à 10H sur notre horloge, il n'est que 9H solaire, et le flux est très médiocre. Il faut donc trouver un compromis dans lequel l'été on prendra en compte la plus grande période énergétique possible, tout en laissant dormir le processeur un maximum l'hiver. La période "nocturne" a été définie entre 19H00 et 8H00 et coloriée en bleu foncé sur le dessin. *(Il serait possible d'élargir "le jour" en été en testant le mois de l'année, mais pour des raisons de simplicité le compromis de la Fig.2 a été retenu, et ce d'autant plus que l'on n'a plus de place pour loger du programme, 100% de la zone réservée au code est consommée. Finasser imposerait de faire "des sacrifices".)*

➢ C'est quand l'ÉTÉ ?

Posant la question au passant dans la rue, on peut s'attendre à une réponse du genre : "***ben quand il fait chaud gros malin !***". Dans le cas particulier du bolomètre, la notion est liée à la quantité d'énergie que peut potentiellement fournir le Soleil quand il n'est pas "voilé par la météorologie". On cherche à préserver au maximum les accumulateurs lorsque la journée n'apporte que peu d'énergie pour les recharger. Il serait possible d'utiliser des statistiques pour choisir les mois considérés comme Ensoleillés et les mois déficitaires, sauf que nous n'avons pas les données nécessaires actuellement. Donc, tant que le bolomètre n'est pas pleinement opérationnel et surtout qu'il n'a pas encore fourni des données sur une période d'un an, il faut improviser les valeurs qui seront soumises au programme. La Fig.3 traduit avec des couleurs les critères de décision provisoires, avec en orange la période considérée comme estivale et en vert celle des frimas.

J	F	M	A	M	J	J	A	S	O	N	D

Fig.3

➢ Se relever la nuit pour les urgences.

Autre difficulté : Durant la nuit, précisément à 0H 00min 00S on change de jour. ***- Et alors ?*** Et bien c'est précisément à cet instant que le programme doit enregistrer les données journalières, décaler le bilan pour la semaine en cours, corriger l'avance de l'Horloge/Calendrier etc. Aussi, chaque journée suffit sa peine et notre fidèle processeur est de garde. Donc, durant la première heure de la journée qui commence, il sort de sa léthargie et assure les missions qui lui sont confiées.
- Ouf, nous y sommes arrivés !
Ben non, ce n'est pas tout à fait terminé. Encore une pierre d'achoppement : Vous venez d'achever vos circuits imprimés, vous avez logé les données en EEPROM et le programme d'exploitation. Bref, vous démarrez pour la toute première fois votre Bolomètre, ou suite à un changement de programme, l'horloge de façon aléatoire indique une heure nocturne.
- Et alors ?
Ben ya pas moyen de sortir du mode sommeil, car la ligne série n'est pas prise en compte. Pour tourner cette difficulté, ***le mode Sommeil ne sera activé que si l'on se trouve dans les périodes bleues*** de la Fig.2 ***et uniquement si le mode Veille est autorisé***. Ainsi, sur un RESET le mode Veille est désactivé et la ligne série permet alors de mettre à l'heure le module. Pour déterminer si l'opérateur autorise le mode sommeil, un "strap" à languette impose l'état logique de l'entrée binaire **D4**.

RÉSUMÉ : Le processeur sera mis en sommeil :
- Entre [1H et 8H] **Si** C'est l'Hiver **OU** Entre [1H et 7H] **Si** C'est l'Été, **ET**
- Entre [19H et 00H] **Si** C'est l'Hiver **OU** Entre [22H et 00H] **Si** C'est l'Été, **ET**
- **Si** le mode Mode_VEILLE est actif. *(État logique de **D4**.)*

Autant dire que formuler cette condition combinée en C++ sans se tromper sur les opérateurs booléens utilisés et le positionnement des parenthèses n'a rien d'évident. Comme il y a pas mal de tests à enchaîner et combiner pour déterminer les périodes de sommeil, les instructions sont regroupées dans la procédure de service nommée Gerer_le_mode_Sommeil() dont le déroulement est détaillé en haut de la page 5.

```
void Gerer_le_mode_sommeil() {
①  byte Debut_Jour, Debut_Nuit;
②  if (HIVER) Debut_Jour = 8; else Debut_Jour = 7;
③  if (HIVER) Debut_Nuit = 18; else Debut_Nuit = 21;
④  if (!digitalRead(Sommeil_interdit)) // Ne passer en Sommeil si le strap est enlevé.
⑤    if (((rtc.getHour() > Debut_Nuit) || ((rtc.getHour() > 0) && (rtc.getHour() < Debut_Jour))))
⑥      Mode_Sommeil();}
```

Pour faciliter la prise en compte des limites de la plage d'endormissement, on crée en ① les variables Debut_Jour et Debut_Nuit. Quand on invoque cette procédure, le booléen HIVER est déjà mis à jour. En ligne ② on définit l'heure pour Debut_Jour alors qu'en ligne ③ on initialise Debut_Nuit. Les lignes ④ et ⑤ ne constituent que la combinatoire du test dont ***les opérateurs logiques sont repérés en rouge***. Enfin, si le test est positif, en ⑥ on invoque la routine complémentaire Mode_Sommeil() qui fait passer le processeur en sommeil durant cinq minutes, puis l'en fait sortir proprement. Noter au passage que cette routine boucle trente cinq fois, où le processeur est endormi puis réveillé pour aboutir au délai désiré. Autant dire que son repos n'est pas "profond" !

NOTE : Si vous consultez NOTICE technique du BOLOMETRE.pdf coté Maintenance informatique, vous constaterez des informations de type "doublon", notamment en **p8** et **p9**. En réalité il s'agit de résumés condensés à disposer sur le bureau quand on reprend le logiciel. On y trouve des détails complémentaires tels que ceux en **p9** sur le "chien de garde" par exemple. Il s'agit d'une sorte de sentinelle qui surveille le chronomètre binaire de huit secondes alors que presque toutes les ressources matérielles internes du processeur sont mises hors tension.

3) *Pas bon le clignoteur !*

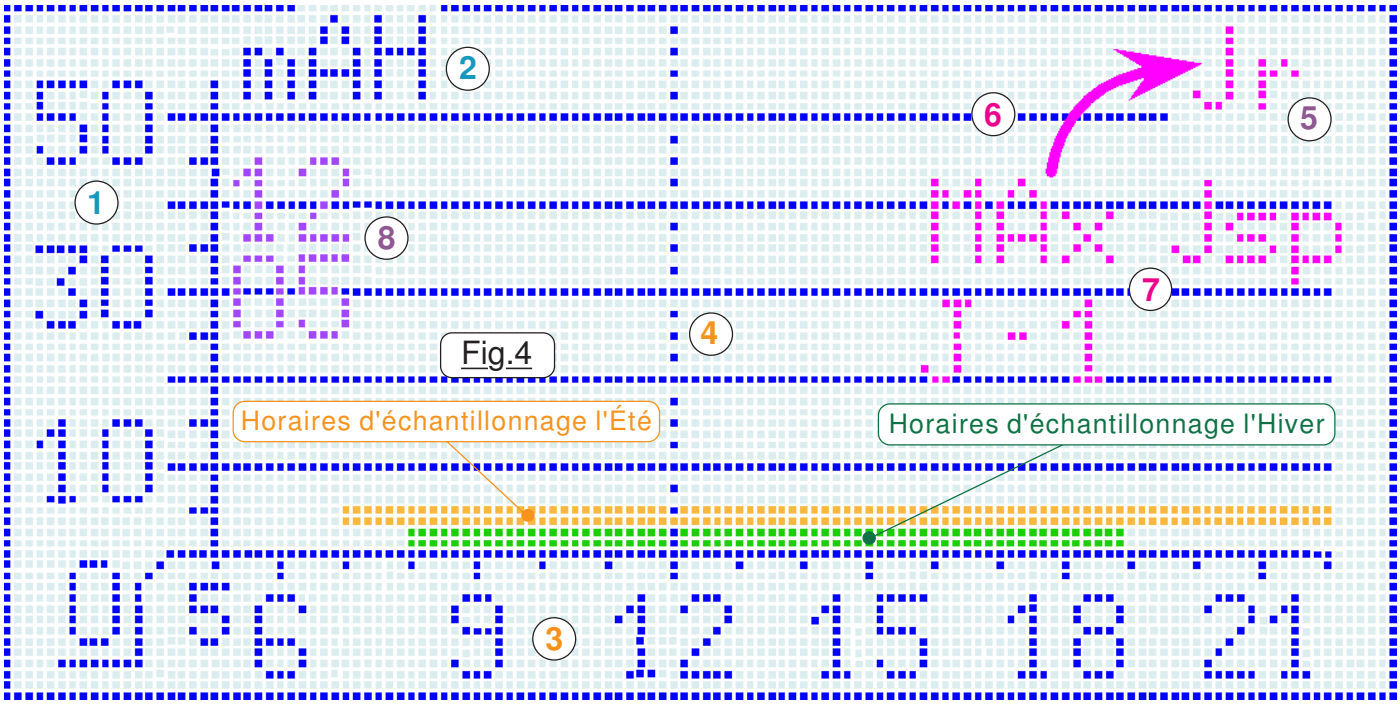
Rien à voir avec le code de la route ... circulez ! Nous allons dans ce chapitre aborder un aspect assez particulier du logiciel qui tente d'optimiser le rechargement des accumulateurs en hiver. L'été, pas de problème, soit l'ensoleillement tolère de charger avec une résistance, soit avec les deux. Comme l'éclairement des cellules photovoltaïques est suffisant, quand les résistances sont mises en circuit la tension ne s'effondre pas. Elle est suffisante pour persister dans la saturation de **T1** et (ou) **T2**. Consultez également la fiche technique ***Commutation de T1 et de T2*** qui regroupe beaucoup d'informations indispensables à la compréhension des séquences d'économie d'énergie. Mais l'hiver il en va tout autrement. Supposons que **T1** et **T2** isolent les résistances de charge. Mis à part le rechargement des accumulateurs et l'alimentation de la carte Arduino, les capteurs sont peu sollicités. La tension aux bornes monte allègrement. Le logiciel détecte "un bon ensoleillement". En fonction de la tension mesurée, il va saturer **T1** et (ou) **T2**. Du coup le débit exigé augmente au delà de ce que peuvent réellement fournir les cellules photosensibles. La tension s'effondre. Le programme s'en aperçoit et une seconde plus tard isole les résistances de charge. Si rien n'est fait pour empêcher ce phénomène, ce qui a été vérifié lors des premiers essais en temps réel, les LEDs vertes de témoin de saturation de **T1** et **T2** clignotent avec une régularité de métronome. Elles s'allument durant une seconde, puis s'éteignent pendant également une seconde. Ce cycle régulier présente un inconvénient majeur : Pendant les périodes où l'ensoleillement est tout juste capable d'alimenter la carte Arduino et recharger les accumulateurs, la moitié du temps on gaspille la faible énergie disponible pour bêtement la transformer en chaleur dans les résistances de puissance.

Comme il n'y aura jamais d'urgence à mettre les résistances **R** en service durant l'hiver, entre les mois d'Octobre à fin Avril on imposera un cycle qui ne commute **T1** et **T2** que 20S après avoir détecté une coupure. Hors période hivernale ainsi définie, on reprendra le cycle "non" protégé, car en été il ne faut pas surcharger les accumulateurs en leur imposant un courant exagéré. Nous avons vu dans le chapitre précédent que le choix de la période allant d'Octobre à fin Avril reste dans un premier temps relativement arbitraire. Il faudra attendre de collecter des données sur au moins une année complète pour pouvoir affiner les paramètres concernés dans le programme d'exploitation. Affaire à suivre ...

HIVER = (Mois < 5) ET (Mois > 9)											
J	F	M	A	M	J	J	A	S	O	N	D
1	2	3	4	5	6	7	8	9	10	11	12

4) *Présentation graphique des données journalières.*

Des informations que l'on désire faire afficher sur l'écran des données et surtout de la façon dont elles seront présentées à l'écran découlent plusieurs conséquences sur les choix à effectuer pour gérer ces informations et surtout l'approche pour les stocker en mémoire non volatile, l'EEPROM présentant une limite de 1023 octets disponibles au maximum. Comme l'écran est limité à 128 x 64 Pixels, cette définition impacte directement les graphiques. Aussi, il importe de bien étudier les écrans graphiques et leurs contenus. La nature des données ainsi que leur format de stockage en découlera directement. Suite à des études préliminaires, la présentation de l'écran journalier ressemble à ce que montre la Fig.4 :



En "hauteur" on visualise l'énergie collectée à chaque échantillonnage de la journée. Horizontalement, on opte assez rapidement pour un étalement sur une plage 17H avec 6 pixels par heures. Chaque "colonne pixel" représente 10 minutes. On "étalera" 6 x 17 = 105 "représentations" sur la largeur, ce qui laissera 22 Pixels pour le cadre extérieur et les graduations **1** à gauche. Par raison d'optimisation pour la représentation des horaires ensoleillés estivaux, on visualisera entre 5H du matin et 22H le soir, ce qui aboutit à l'échelle temporelle **3**. La mie journée à 12H est repérée verticalement par la ligne pointillée **4**.

On désire afficher toutes les 10 minutes. Avec 300mA au maximum constant durant une heures, on totalisera une énergie de 300 / 6 = 50 mAH au maximum par échantillon. La "hauteur" de l'afficheur étant limitée à 64 lignes, on voit en **1** que chaque 10mAH sera divisé en huit graduations, avec une résolution "verticale" de 1,25mAH. En **2** les unités d'énergie sont indiquées dans une zone graphique inutilisée aux horaires concernés, et de ce fait il n'y a pas de risque d'interférence avec la courbe de l'historique.

Outre le graphe de l'évolution diurne de l'énergie mesurée, on désire aussi pouvoir visualiser les valeurs enregistrée lors de la journée la plus énergétique, et celle d'une journée spécifique au cours de l'année. Les trois pages d'écran se ressembleront, aussi il faut pouvoir les différencier. On profite que la zone **5** ne sera jamais occupée par du tracé de données pour préciser la nature des données. Si **Jr** est affiché, c'est qu'il s'agit de l''évolution pour ***la journée en cours***. Comme montré en **6** et **7** les deux journées particulières seront précisées par **MAX** et par **Jsp**. *(Et **J-1** pour la veille.)* Pour la **J**ournée **sp**éciale utilisateur le graphe est surchargé en **8** par le jour et le mois de l'enregistrement réalisé en manuel.

➢ *La mémorisation des données.*

Maximum par échantillon = 50mAH. On désire afficher avec le maximum de précision possible, donc on doit mémoriser la plus grande valeur possible sur un octet. On peut multiplier par 5 la valeur mesurée, qui aboutira à un maximum de 250 qui reste compatible avec le codage sur un Octet. Ainsi on enregistre avec une précision largement suffisante pour afficher avec pertinence à 1.25mAH.

Enregistrer l'historique de la journée précédente, de la journée la plus énergétique et celle de la journée spéciale utilisateur aboutit à consommer 3 x 90 = 270 Octets. *(Et 8 Octets dans les données de base pour préserver les deux réels correspondant à MAX et à min.)*

5) *Présentation graphique des données annuelles.*

Comme pour les données journalières, la façon dont seront présentées à l'écran les enregistrements effectuées sur une année conditionne directement les méthodes de traitement et surtout les formats de sauvegarde dans l'EEPROM. Il importe avant d'effectuer des choix stratégiques, de définir avec soin la mosaïque graphique, car ses limites en taille imposent la finesse avec laquelle seront visualisées les valeurs, et par voie de conséquence l'optimisation qui en résultera pour le choix du type de donnée sauvegardée. Après divers essais, on aboutit au projet de la Fig.5 pour lequel on représentera les niveaux enregistrés pour les 52 semaines couvrant la période d'une année complète avec en **6** l'ordre du jour courant. En réservant sur la largeur de l'écran deux colonnes par semaine, on étalera sur 104 pixels l'année visualisée, ce qui couvre une bonne majorité de l'écran tout en laissant assez de marge pour afficher le cadre extérieur et surtout à gauche en **1** les valeurs que représentent les graduations.
Taille pour sauvegarder sur l'année : On sauvegarde une fois par semaine soit 52 échantillons. Si on sauvegarde directement les réels, ce qui simplifie le logiciel, la sauvegarde impose donc 4 * 52 = 208 octets.
Chaque heure l'énergie maximale *(Si ensoleillement constant.)* est de 300mAH.
0.3AH * 24 = 6AH maximum par jour soit 42AH maximal par semaine. La "hauteur" disponible étant de 64 Pixels, comme montré en **1** on va graduer entre 0 et 45, chaque pas étant représenté sur une ligne. La

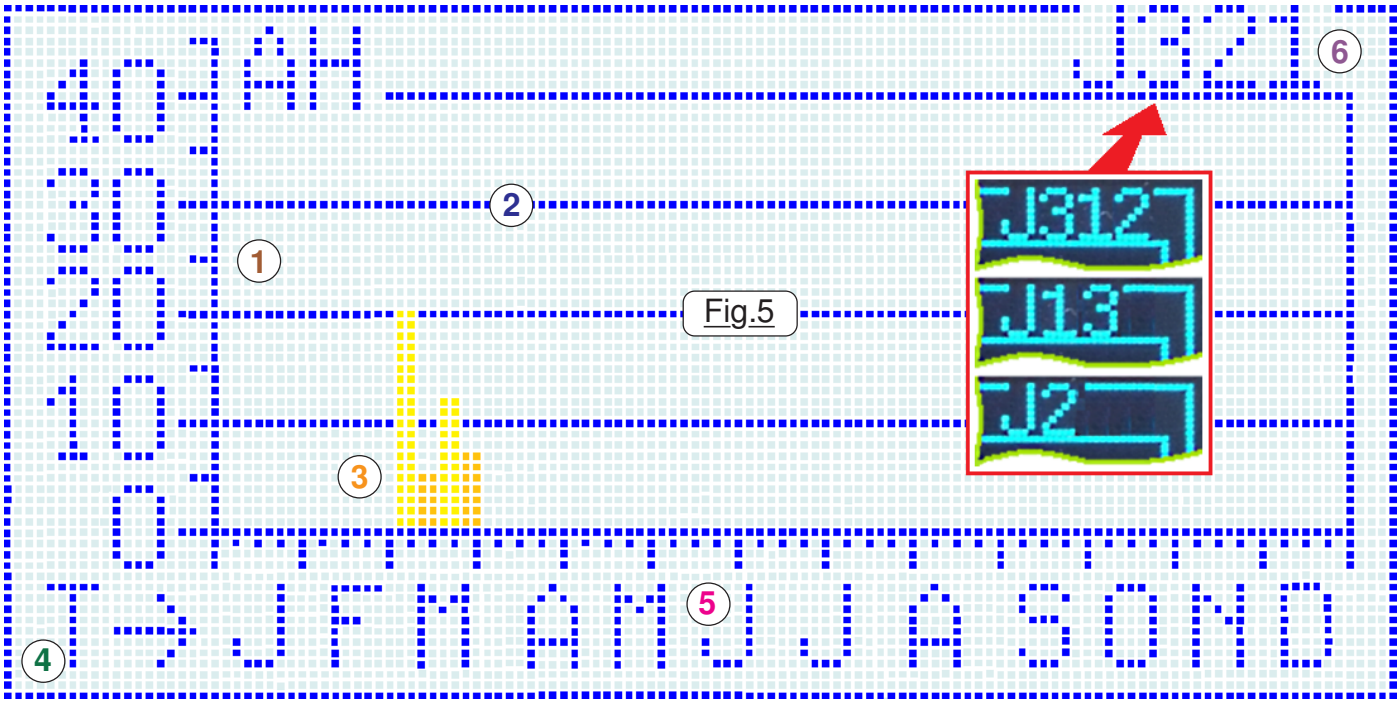


Fig.5

surface de la mosaïque est ainsi bien utilisée. Pour faciliter la lecture, les niveaux des dizaines sont tracés en **2**. En **3** chaque "bâton constitué de deux colonnes" représente une semaine. En **4** la flèche précise dans quel sens s'écoule le **T**emps. Les graduations les plus grandes symbolisent quatre semaines.

➢ *Le treizième mois !*

Chaque année "banale" qui se respecte contient en principe 52 semaines. Donc, sur la Fig.5 sont étalés en largeur 52 "colonnes doubles" du genre **3**. La largeur de l'écran pour visualiser sur une année consomme donc 104 colonnes.

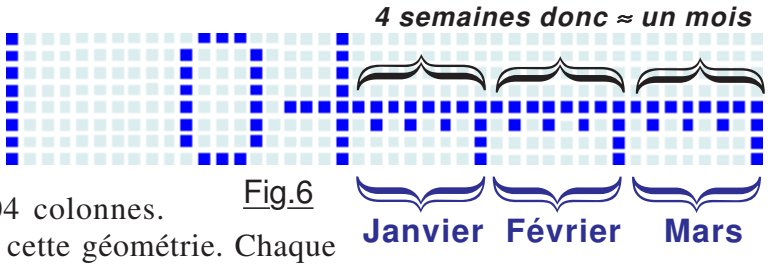
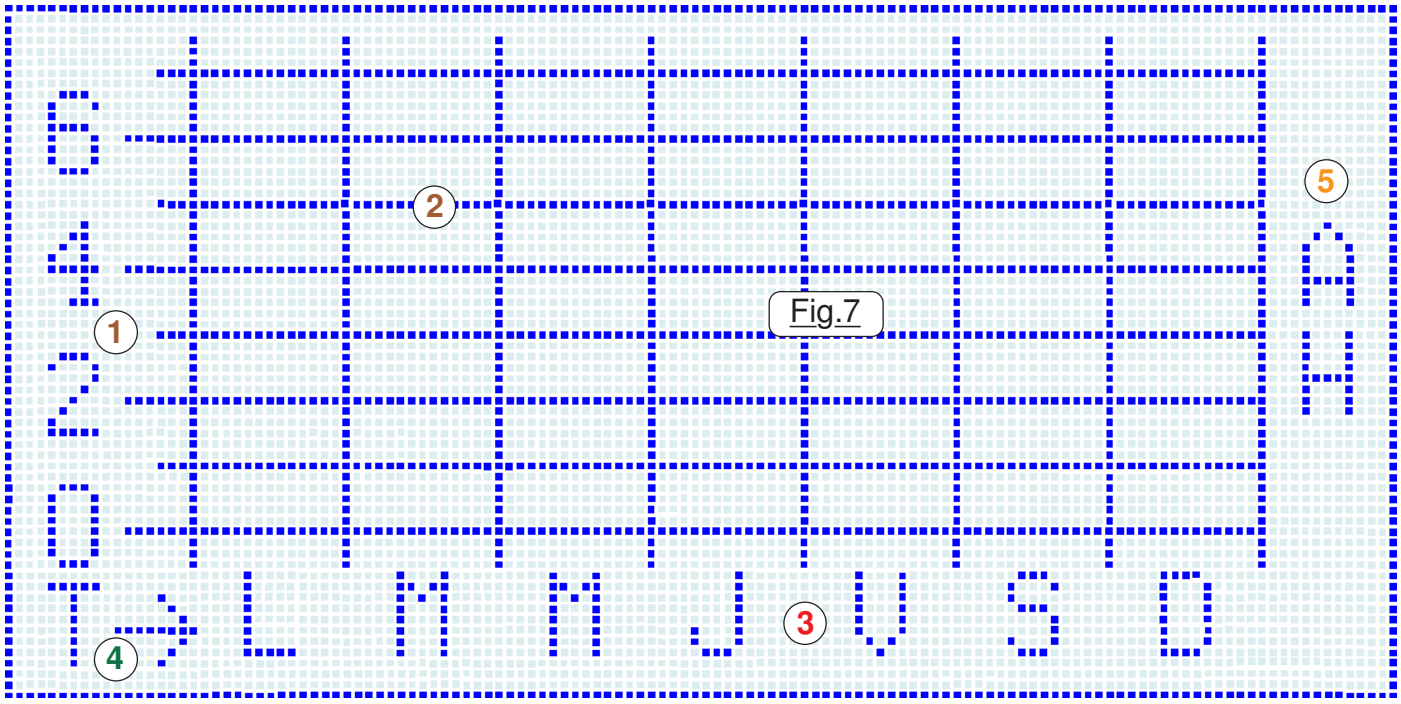


Fig.6

La Fig.6 précise l'interprétation qui résulte de cette géométrie. Chaque quartet de quatre semaines ressemble globalement un mois. Comme l'écran manque de définition pour tenir compte des durées différentes pour les divers mois de l'année, on va simplifier en supposant qu'ils sont tous égaux et font exactement quatre semaines. Du coup, l'année conserve bien sa structure de 52 semaines, mais elle grandit et affiche fièrement 13 mois ! Pour tourner cette difficulté purement géométrique, on étale **J**anvier à **D**écembre sur 13 mois ce qui donne le compromis représenté en **5**. Avec ces arbitraires, on exploite au mieux la matrice de Pixels, et l'on peut situer assez rapidement la période de l'année visualisée sur ce graphe. Si l'on désire savoir avec précision l'énergie collectée au cours d'une semaine particulière, il suffira de noter son ordre sur un calendrier et de rechercher sa position sur le graphe.

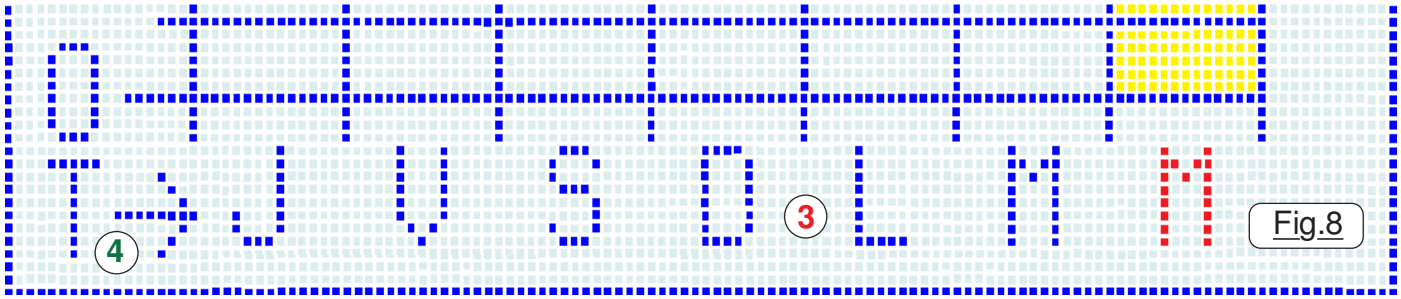
6) *Présentation graphique des données d'une semaine.*

Présenter l'écran avec des invariants identiques à ceux de la page des données annuelles est très favorable pour minimiser le code. C'est la raison pour laquelle la mosaïque envisagée en Fig.7 reprend certains des tracés de la page précédente. Dans cette optique on conserve le cadre extérieur ainsi que la flèche **4** qui précise dans quel sens s'écoule le **T**emps et donne lieu à une procédure de construction d'image. Les graduations en **1** et les lignes des **AH** en **2** sont spécifiques puisque l'échelle va de 0 à 7,5 environ. En **3** sont indiqués les jours de la semaine, et comme il reste une large plage non occupée à droite, on ajoute en **5** les unités **AH** qui expriment l'énergie mesurée donnant lieu au graphe.

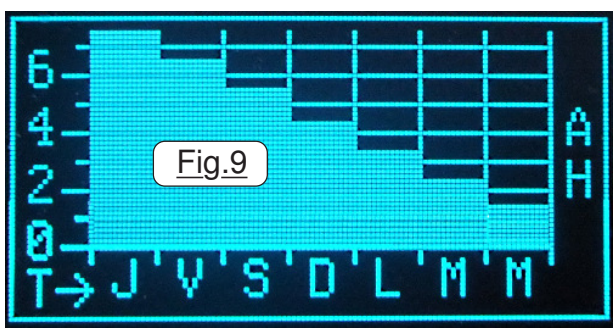


➢ ***Le recyclage des inscriptions "journalières" dans la semaine.***

Étant donné que la page écran de la Fig.7 présente les valeurs des sept derniers jours enregistrés, le plus ancien est à gauche, et "la veille" est à droite. Le jour courant étant quelconque, la colonne J-7 n'est pas forcément un Lundi. Aussi, pour que le graphe soit cohérent et surtout d'une interprétation conviviale, il importe de positionner correctement les lettres représentatives des jours de la semaine.



Considérons la Fig.8 qui visualise ce qu'il faut afficher si la journée courante **Jr** est un Jeudi. Sur ce dessin, "Hier" est colorié en jaune pour le mettre en évidence. Pas besoin de pousser loin le raisonnement pour déduire que **J-1** est un **M**ercredi. Sur les graphes, le temps "s'écoule" de la gauche vers la droite, sens de variation précisé par la flèche **4**. De ce fait, on se déplace dans le passé de la droite vers la gauche, ce qui impose l'ordre des lettres **3** comme présenté sur la Fig.8 imposant un traitement spécifique de cet affichage lorsque l'on change de journée. L'exemple pris en compte engendre l'affichage de la Fig.9 les données en EEPROM ayant été inscrites avec Ecrire_les_Valeurs_en_EEPROM.ino. La date correcte est directement issue du petit circuit Horloge / Calendrier qui équipe le BOLOMÈTRE.



7) L'ordre des semaines au cours de l'année.

Pour gérer l'écran annuel de la Fig.5 proposée en page 7, il importe de savoir quel est le numéro de la semaine dont on sauvegarde les données. Calculer l'ordre dans les 52 positions n'est pas aussi élémentaire que l'on pourrait le penser. En effet, le premier Janvier ne tombe pas forcément un Lundi. Du coup, la première semaine, la n°1 peut fort bien débuter fin Décembre, et certaines années peuvent comporter jusqu'à 53 semaines. Sans entrer dans les détails, l'encadré de la Fig.10 résume les points importants :

L'ISO propose les recommandations suivantes : Fig.10

- Le lundi est considéré comme le premier jour de la semaine. *(Norme ISO 8601)*
- Les semaines d'une même année sont numérotées de 01 à 52. *(Parfois on peut avoir 53)*
- La semaine qui porte le numéro 01 est celle qui contient au moins quatre jours. En général c'est celle qui contient le premier jeudi de janvier.

NOTE : Il peut exister une semaine n° 53 en particulier les années finissant un jeudi, et les années bissextiles finissant un jeudi ou un vendredi.

Compte tenu de l'application envisagée, on peut largement simplifier en ne représentant systématiquement que 52 semaines, aussi un positionnement approximatif est largement suffisant pour tracer le graphe de la Fig.5 raison pour laquelle nous avons "homogénéisé" la durée des treize mois. Cependant, pour tracer les "bâtons" représentatifs de l'énergie collectée durant chaque semaine, il importe de déterminer assez précisément le n° de cette dernière dans l'année. Comme un logiciel qui tiendrait compte automatiquement des informations de l'encadré de la Fig.10 serait trop complexe vu le contexte, c'est manuellement que nous lui fourniront l'information de base dont il a besoin.

Nous devrons lui indiquer le numéro de ***la journée dans l'année*** qui correspond au **L*undi de la deuxième semaine*** de l'année, c'est à dire la journée n°**7** qui sur la Fig.11 est coloriée en jaune. Cet exemple correspond à l'année 2019 durant laquelle est achevé le développement du programme. Comme nous le savons, 2019 commence un Mardi. Donc sa première semaine contient plus de 3 jours et de ce fait est la n°1. Le **L**undi de la Semaine n°2 étant le 7 Janvier, c'est la valeur **7** qu'il faut préciser au logiciel. Voici comment procéder :

Fig.11

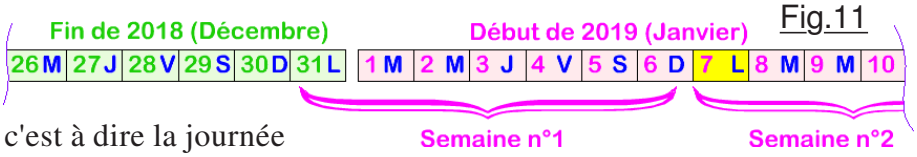


Fig.12

1) Enlever de "strap" qui ponte **D7** nommée Presence_Strap à GND.
2) Provoquer un RESET. La LED verte clignote rapidement invitant à cliquer sur l'une quelconque des deux touches du clavier.
3) Le premier clic fait afficher l'invite de la Fig.12 sur l'écran OLED. La LED verte clignote encore rapidement. Cliquer une deuxième fois.

*Normalement la première page du menu d'exploitation s'afficherait, mais comme **D7** est à l'état "1", c'est le sous menu d'initialisation de l'ordre des semaines qui s'affiche. Montré en Fig.13 il précise en **1** la valeur actuellement mémorisée en EEPROM.*

4) Chaque action sur la touche rouge augmente la valeur avec recyclage à 1 arrivé à la valeur 8. Donc en quelques clics, qu'ils soient longs ou courts on fait afficher la bonne référence. La LED verte clignote rapidement précisant que l'on n'a pas fini.
5) La seule façon de ***sortir de cette fonction***, et qui remplacera la valeur actuelle stockée en EEPROM par celle affichée sur l'écran, consiste à effectuer un ***clic long sur la touche noire***.
6) Replacer le "strap" languette sur **D7** pour informer le logiciel que la pile de sauvegarde est isolée.

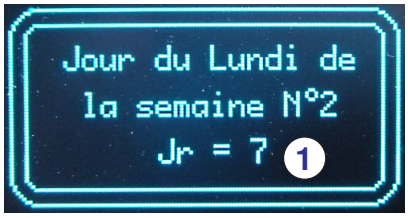


Fig.13

A partir de cette information, le logiciel est maintenant capable de calculer quel est l'ordre de toute semaine qui commencera en fonction de l'ordre de son Lundi dans l'année. Notez au passage que le cinquième écran d'exploitation affiche dans le petit cadre en haut à droite le n° de la semaine en cours ainsi qu'à gauche le jour concerné. Aussi il sera facile à tout moment en comparant avec un calendrier de l'année en cours, de vérifier la cohérence de ces information. S'il y a lieu, on peut alors aisément procéder à la réinitialisation souhaitable.

8) Procédure pour initialiser la date et l'heure sur l'orloge temps réel.

Toute horloge électronique, qu'elle soit organisée avec des composants discrets, des circuit intégrés, qu'elle utilise comme référence la vibration d'un quartz ou celle d'un atome "truc-machin", ne peut en aucun cas découvrir toute seule la date, l'heure ainsi que le jour de la semaine puisque ces notions sont totalement arbitraires. C'est l'historique de la vie sur Terre et notamment les préoccupations matérielles des humains, qui avec beaucoup d'hésitations et de difficultés ont au cours des millénaires ont bâti les notions de la fuite du temps, et élaborés des jalons temporels. Que "la machine" qui sert à indiquer ces jalons soit mécanique, clepsydre, électronique ou atomique, il importe forcément qu'au moment de la mettre en service on puisse initialiser ses données et les synchroniser "à l'instant" précis où ces machines vont égrener les secondes. Naturellement le petit circuit horloge utilisé dans le Bolomètre n'échappe pas à cette contrainte. Comme c'est une procédure qui par principe ne sera qu'une étape très occasionnelle dans la vie du petit appareil électronique, avoir à faire intervenir un ordinateur et l'**IDE** pour procéder à cette étape incontournable n'est pas du tout pénalisant. C'est la raison pour laquelle une procédure qui utiliserait uniquement les ressources du bolomètre en autonomie n'a pas été envisagée. Avec seulement deux boutons pour son clavier de servitude, le protocole aurait singulièrement manqué de convivialité. Comme de toute façon le contexte de téléchargement impose l'usage d'un ordinateur associé à l'**IDE**, autant se simplifier la vie. De toute façon, avant de téléverser le programme d'exploitation du bolomètre, on doit passer par l'étape qui gave l'EEPROM de textes et de données, alors faire intervenir un petit logiciel de servitude de plus n'est pas la mer à boire. Voici la procédure d'initialisation de l'Horloge qui du reste est également intégrée en **p4** de Manuel d'UTILISATION.pdf :

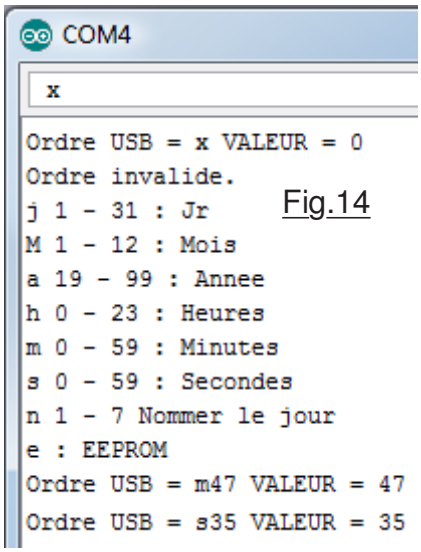


Fig.14

1) Mettre en service l'ordinateur, l'**IDE** et téléverser un croquis quelconque,
2) Retirer le strap à languette qui autorise le mode Sommeil pour permettre les manipulations quelle que soit l'heure ainsi que la saison actuellement contenue dans le module Horloge/Calendrier,
3) Cliquez sur l'icone qui ouvre le Moniteur série, action provoque un RESET du Bolomètre,
4) Vérifier que la vitesse de dialogue sur la ligne USB est bien de 115200 bauds,
5) Activer l''écran OLED et choisir une page écran dans laquelle l'heure défile en haut à droite.
6) Sur le **M**oniteur, envoyer un caractère '**x**' qui ne fait pas partie des commandes valides. Comme montré en Fig.14 le texte du compte rendu liste les commandes possibles de cette fonction,
7) Par exemple commencer par initialiser le jour courant. Par exemple frappez "j19" ou "j 19" ou "J19", dans les trois cas l'écran affiche `Consigne USB = j 19  ARGUMENT = 19` précisant que la consigne est valide et prise en compte. La valeur du jour est alors inscrite dans la mémoire du circuit intégré spécialisé.

> **NOTE :** Cette procédure peut être réalisée en aveugle pour s'entrainer, c'est à dire avec uniquement la petite carte Arduino branchée sur la ligne série. Toutefois, si le module horloge n'est pas branché, on se doute que rien de concret ne se passera. Lorsque l'électronique du Bolomètre est complète, l'écran OLED est utilisé pour afficher les données pertinentes. Si une donnée passée en paramètre est hors limites, *(Limites précisées par des informations du genre 1 - 31.)* un texte d'erreur du type `(Hors limites.)` est affiché.

8) Quand la date et le jour de la semaine sont entièrement initialisés, inscrire la valeur de l'Heure et celle des minutes. Puis, sans tarder imposer les secondes et valider sur le clavier en synchronisant ces dernières sur une horloge précise que vous utilisez en référence. *(Ce peut être une horloge électronique synchronisée par radio sur DCF, le TOP horaire à votre poste de réception radiophonique etc.)*

Maintenant l'horloge interne est à l'heure. Vous vous doutez qu'elle va forcément dériver légèrement au cours des semaines qui s'écoulent, car toute horloge autre que les "monstres atomiques" subit l'influence de son entourage comme les variations de température pour les quartz par exemple. Une procédure simple et autonome qui recale l'heure directement sur site est intégrée au MENU du bolomètre et décrite en page **p4** du livret réalisé à partir de Manuel d'UTILISATION.pdf .

AVIS à la population : Les chapitres qui suivent ne concernent que les programmeurs sur Arduino qui veulent vraiment en savoir plus car on aborde dans ces pages des aspects assez "particuliers" peu développés dans la littérature Arduino.

9) *Fin de mois 30 jours ... tout s'arrête.*

C'est un peu comme le réputé "bug de l'an 2000", si on ne fait rien tout se bloque ! C'est un peu comme cette bonne vieille pendule suisse qui bat la seconde, quand les poids du mécanisme sont arrivés en bas, si on ne les remonte pas, le balancier s'immobilise et la pièce n'est plus bercée par le tic tac régulier qui diffuse sa sérénité. Pour notre petite station scientifique, les actions du programme sont déclenchées une fois par seconde. Examinons sur la Fig.15 le fonctionnement de l'horloge informatique :

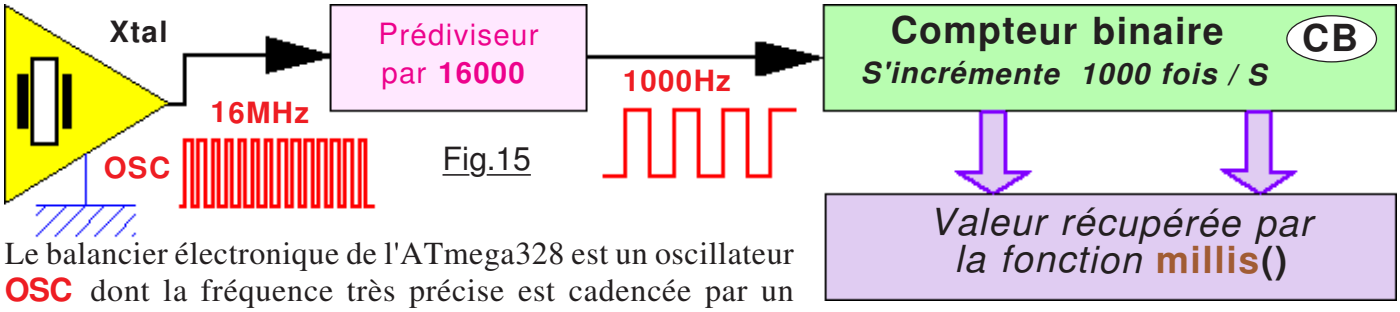


Fig.15

Le balancier électronique de l'ATmega328 est un oscillateur **OSC** dont la fréquence très précise est cadencée par un quartz qui vibre à "exactement" 16.000.000 fois par seconde. Divisé par 16000 cette électronique fournit un signal d'exactement 1kHz dont chaque alternance fait exactement 1mS.
En langage C++ adapté à Arduino, donc au microcontrôleur ATmega328, la fonction millis() sans paramètre se contente de lire le contenu actuel du compteur binaire **CB**. Dans le programme, on utilise une variable **A**ncienne_**v**aleur_**T**emps_**E**coule_**d**epuis_**R**ESET qui comparée à (millis() + 1000) sert à cadencer exactement la seconde pour gérer temporellement les actions de la carte Arduino.

PROBLÈME : Le compteur **CB** qui "tourne" sur 32 BITs présente forcément une capacité maximale de 4294967295. Arrivé à cette valeur, l'impulsion suivante le refait circuler à zéro ce qui arrive tous les 49,71 jours environ. *(Si le processeur n'est jamais mis en sommeil.)* Pour comprendre le phénomène examinons le tableau de la Fig.16 donnant les valeurs critiques toutes les 1000 comparaisons :

Fig.16

Cas	millis()	AvTEdR	Action
1	4294962295	4294962294	millis() > AvTEdR : Actions temporelles.
2	4294963295	4294963294	millis() > AvTEdR : Actions temporelles.
3	4294964295	4294964294	millis() > AvTEdR : Actions temporelles.
4	4294965295	4294965294	millis() > AvTEdR : Actions temporelles.
5	4294966295	4294966294	millis() > AvTEdR : Actions temporelles.
6	4294967295	4294967294	millis() > AvTEdR : Actions temporelles.
7	0	4294967294	millis() < AvTEdR : Aucune action.
8	1000	4294967294	millis() < AvTEdR : Aucune action.
9	2000	4294967294	millis() < AvTEdR : Aucune action.

Dans ce tableau sont mises en évidence par la couleur rose les valeurs qui changent. Les cas **1** à **6** sont relatifs aux six dernières secondes avant que le problème ne se produise. Toutes les 1000 impulsions du prédiviseur **AvTEdR** s'incrémente d'une unité. Sa valeur dépasse celle de la variable de comparaison **AvTEdR** et le test est positif. Les actions à conduire toutes les secondes sont déclenchées. Puis, en **7** le compteur binaire **CB** "bascule" à zéro. À partir d'ici **millis**() devient définitivement inférieur à la valeur d'**AvTEdR** et le test systématiquement négatif. Tout se bloque car les actions temporelles sont figées.

RÉSOLUTION du PROBLÈME : Comme dirait le légendaire La Palisse, il suffit de ne plus arriver à ce cas critique, ou sous une autre forme "ne plus avancer" quand on est au bord du ravin !
L'idée de base consisterait à parer ce cas critique quand il se produit, c'est à dire en **7**. Dans ce but on ajoute au programme l'instruction :

```
if ((AvTEdR - millis()) > 10000) AvTEdR = millis() + 1000;
```

Certains remarqueraient que la valeur 10000 pour la comparaison est forte, que l'on pourrait logiquement se contenter de 1000. Possible. Reste qu'il faudrait examiner avec soin le comportement du programme, et comme je n'ai pas envie de tester en temps réel sachant que chaque vérification ne

pourra se faire que tous les 45 jours, *(Et bien plus puisque souvent le processeur fait dodo.)* et encore, si entre temps il n'y a pas eu de RESET, je préfère de loin assurer la sécurité du test. Le tableau de la Fig.17 reprend les données en tenant compte de la nouvelle instruction introduite dans le programme.

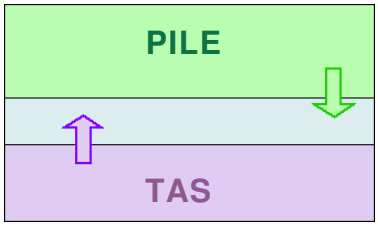
Cas	millis()	AvTEdR	AvTEdR - millis
1	4294962295	4294962294	< 10000 : Test sans effet.
2	4294963295	4294963294	< 10000 : Test sans effet.
3	4294964295	4294964294	< 10000 : Test sans effet.
4	4294965295	4294965294	< 10000 : Test sans effet.
5	4294966295	4294966294	< 10000 : Test sans effet.
6	4294967295	4294967294	< 10000 : Test sans effet.
7	0	4294967294	> 10000 : Valeur de millis() forcée
8	1000	1000	dans la variable AvTEdR.
9	2000	2000	Le cycle recommence.

Fig.17

AVIS à la population : Le chapitre qui suit ne concerne que les programmeurs purs et durs qui veulent en savoir plus.

10) Vérifier que l'impensable ne va pas se produire.

Fig.18



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 *(En bleu clair sur la Fig.18)* 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 PAFFFFFFffffff, c'est du n'importe quoi.

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 100% de zone programme utilisée, on se doute que le TAS et la PILE sont plus que sollicités. Du reste pour pouvoir conduire la manipulation il faut libérer de la place en "cachant" dix lignes de la matrice du PAPILLON. 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()); ATTENDRE_un_BP();
//@@@@@@@@@@@@@@**********************************************************@@@@@@@@@@@@
```

Les deux lignes de commentaire qui encadrent le code contiennent des chaînes de caractère du genre @@@@@@@@@@ pour pouvoir facilement les repérer dans le listage. 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 et de passer en remarque les dix lignes délimitées dans la matrice du PAPILLON. Le programme démarre normalement. Puis, quand on clique dans la page d'accueil pour passer en exploitation, l'écran devient noir avec au centre une valeur numérique. C'est précisément la valeur de la zone disponible. Le programme attend un clic au clavier pour passer en exploitation, nous laissant ainsi le temps de lire la valeur. Actuellement, dans le cas de BOLOMÈTRE la validation de cette séquence annonce un espace confortable de 542 octets, auquel il faut retrancher les 141 octets qui reconstitueront intégralement le PAPILLON. Il n'y a donc pas de "plantage vicieux" trop à craindre, même avec la dernière version "abusive" qui consomme 100% de l'espace réservé au programme.

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 consomme inutilement des ressources du microcontrôleur. Le laisser serait impertinent si la marge de place disponible est limite, car ces séquences viennent encombrer de la place mémoire. 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 dédiée et montrée en Fig.19 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 seule ne présente de la pertinence le nombre **B**. Aussi, vous avez bien compris que ce luxe stupide est présent dans le coude source uniquement parce-que le programme a montré qu'il restait de la place à revendre. Dans un contexte de "vaches maignes", il ne faudrait surtout pas faire afficher ce texte ou ... risque de COLLISION !

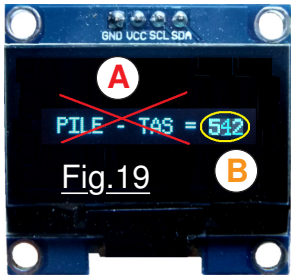


Fig.19

11) Abuser n'est pas jouer !

J'avoue que lorsque je développe un petit projet qui sera autre chose qu'une modeste expérience, c'est à dire qu'il doit se concrétiser par un quelconque appareil, généralement à vocation ludique, j'adore pousser le bouchon un peu loin. En d'autres termes, mon plus grand plaisir consiste à gaver le processeur en frisant l'exagération. Je savoure intensément lorsque ***Dudule*** ne peut plus faire remarquer comme en page 23 que l'on gaspille de la place.

➢ Un peu d'histoire.

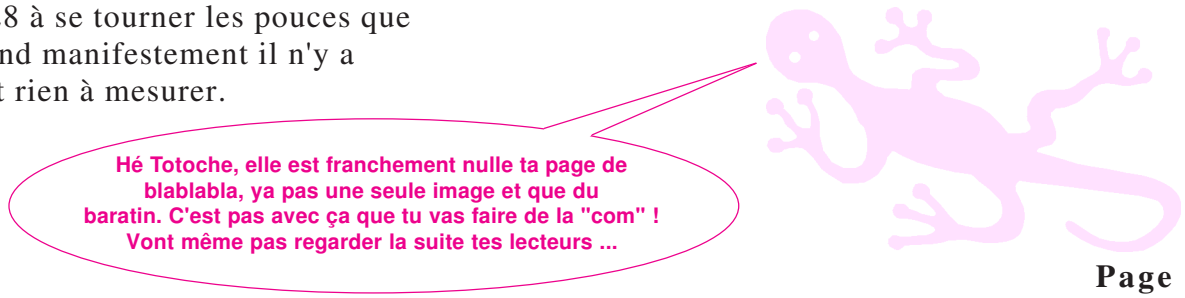
Tous mes programmes un tant soit peu sérieux comportent ***en tête du listage source un historique du développement qui situe chaque étape, et surtout son incidence sur les zones mémoires consommées par le croquis. Ces données peuvent s'avérer indispensable si à un moment donné on manque de place et qu'il faut sacrifier des options. Avoir une idée précise de ce que dégage chaque séquence de code permet alors de faire un choix sur les parties à sacrifier.*** Ce sont ces données qui ici permettent de reconstituer l'historique de BOLOMETRE.ino et annoncer des valeurs précises. Chronologiquement, pour faire plaisir à ***Dudule***, on a ajouté le PAPILLON pour "griller" des octets. Il restait encore beaucoup de place à gaspiller. Alors, pour améliorer la convivialité du Bolomètre et ne pas avoir à utiliser un programme spécifique tel que MAJ_Horloge_Par_IDE_sur_USB.ino pour initialiser l'Horloge/Calendrier, un protocole utilisant le moniteur de l'**IDE** a été intégrée. Alors qu'il restait encore 892 octets à dilapider, a germée l'idée de pouvoir récupérer à tout moment les données sauvegardées en EEPROM avec le moniteur de l'**IDE** pour pouvoir par exemple les intégrer dans un tableur. Avec cette possibilité, le petit module NANO confine à du PRO ! Et surtout, lorsque la procédure de transfert des données a été intégrée au programme, il ne reste plus que 68 octets disponibles en mémoire de programme.

Franchement, à ce stade on doit considérer que l'on est à la limite du raisonnable, et qu'il est grand temps de se calmer. C'est à la fois peu et beaucoup. Peu car si l'on doit apporter une modification notable au programme, ces 68 emplacements risquent de ne pas suffire. Beaucoup, car avec si peu il reste possible de coder encore pas mal d'instructions. Figurez-vous que, comme mentionné en page 3, dans la version testée en début janvier est née l'idée de mettre le processeur en Sommeil durant la nuit. Et bien 68 octets ont été suffisants pour ajouter le code nécessaire au programme ... ***mais il ne restait plus que 6 octets, de quoi passer en correctionnelle !***

➢ Ne pas confondre Veiller et Sommeiller.

Lorsque le programme fonctionne et que l'afficheur OLED présente une page de données, la carte Arduino avec ses périphériques consomme environ 33mA. Aussi, dès le début le ***mode Veille*** qui engendre un écran tout noir a été émulé. Dans ce cas, la consommation de la carte NANO diminue aux environs de 25mA. Seul l'écran de l'afficheur est neutralisé, l'ATmega328 continue à dérouler les instructions, à piloter les transistors de commutation et à surveiller le clavier. Donc, dans le mode veille le processeur est actif et continue sans relâche à gérer ses périphériques. Hors, tout microcontrôleur digne de ce nom doit pouvoir être mis en sommeil, c'est à dire "débrancher" des ressources internes pour que sa consommation soit réduite à loisir. L'ATmega328 possède plusieurs ***modes SOMMEIL***. On a utilisé celui qui réduit le plus la consommation, c'est à dire le mode SLEEP_MODE_PWR_DOWN.

Quand ce mode est activé, la consommation passe de 25mA à 14mA, c'est à dire presque deux fois moins. C'est génial, sauf que ... le processeur fait dodo, donc il ne s'occupe plus de rien ! La petite station scientifique devient un parasite qui chauffes des résistances sans plus savoir pourquoi. C'est la raison pour laquelle il ne faudra autoriser ce cher ATmega328 à se tourner les pouces que la nuit quand manifestement il n'y a strictement rien à mesurer.



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

Compilons la dernière version du programme d'exploitation. ***GLUPS ... il ne reste plus que ZÉRO octet de disponible pour le programme***. Du reste le compilateur ne se sens pas à l'aise, il nous avertit par le message de la Fig.20 de couleur orange. Pour faire court : Dans l'intégralité des zones mémoire disponibles de l'ATmega328 il ne reste non utilisé que 10 octets en EEPROM, plus rien pour le programme et comme on l'a vu 401 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 401 octets entre la PILE et le

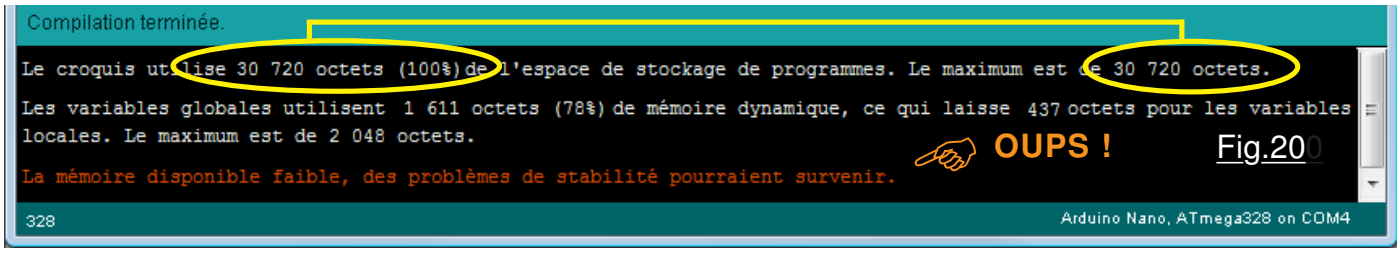


Fig.20

TAS le programme devrait se montrer stable. Donc, si un jour on doit faire du vide car une modification du programme l'impose, il ne sera pas très dramatique d'enlever une partie du PAPILLON ou retirer l'initialisation intégrée de l'Horloge/Calendrier et téléverser provisoirement le croquis dédié MAJ_Horloge_Par_IDE_sur_USB.ino disponible dans <Pour les programmeurs>.

Relativisons et surtout, un petit historique s'impose. Il faut savoir que c'est mon tout premier programme où j'ose consommer l'intégralité de la zone mémoire réservée au programme. Et pourtant, curieusement la place disponible en mémoire dynamique reste globalement supérieure à celle qui généralement subsiste pour mes autres logiciels. En général, le résidu disponible ne dépasse presque jamais les 200 octets pour mes croquis "cossus". Cette particularité est notamment liée au fait que la grande majorité des textes est logée en EEPROM et que le programme n'utilise pas beaucoup de tableaux. Toutefois, il faut savoir que durant le développement il m'est arrivé souvent de dépasser les fatidiques 100% imposant de trouver de la place. Malgré toutes les optimisations dont je suis capable, y compris les plus subtiles, il s'est avéré indispensable de faire à plusieurs reprises "du vide".

12) Rien ne va plus !

À trop exagérer, arrive un moment où l'avenir du programme se joue à la roulette. Traduisez : Boum scrach prouichhhhh, le couperet tombe. Le programme tournait comme une montre suisse, maissss sa mise en action sur le réel a fait surgir des faiblesses, des failles qu'il faut impérativement corriger. On modifie le code source, et c'est reparti pour un tour. Sauf que brusquement le compilateur se fâche et signale à sa façon que "ça ne rentre pas". C'est laconique, mais parfaitement clair : Le code objet dépasse les fatidiques 30720 octets. C'est la "cata" car nous n'avons plus de variable d'ajustement. Il est possible d'enlever le PAPILLON, c'est l'action de loin la plus rentable pour dégager beaucoup de la place.

- NON, pas le PAPILLON ! *(Il est durdur Dudule !)*

On peut également enlever la fonction de mise à l'heure par la ligne USB. Ce n'est pas pénalisant mis à part qu'il faut téléverser deux croquis chaque fois que l'on devra réinitialiser le module. Comme de toute façon nous sommes obligés de brancher le Bolomètre sur le Moniteur de l'**IDE** et de charger un croquis quelconque pour saisir la date et l'heure, c'est peu pénalisant et reste facile à faire.

- NON, pas la mise à l'heure avec la Uessebé ! *(Vraiment intraitable, veut rien lâcher le diable !)*

- OK Dudule, mais si l'on enlève rien, où trouver de la place ?

- Ben dans le programme, c'est pas sorcier !

Bon, pour ne pas contrarier notre exigeant copain, nous allons aller à la recherche d'octets pour faire des économies. Dans ce type de situation c'est par divers "gagnepetits" que l'on arrive ici et là grappiller quelques babioles. ***D'une façon générale c'est sur le LOGO s'il y en a un, ou dans les textes que l'on trouvera le plus facilement des octets sans empiéter sur les fonctionnalités du programme.*** Pour imager ce propos, et renonçant à enlever quoi que ce soit, nous allons faire des "économies bout de chandelle". *(Vous avez certainement compris que ce brave Dudule n'y est pour rien, c'est Môamôa qui ayant mis au point de belles séquences n'arrive pas à envisager de les effacer. Le luxe est pire que la cigarette, rien à faire pour y renoncer quand on y a goûté ...)*

➢ Compacter des chaînes de caractères.

Commençons nos économies en cherchant à grappiller dans les "bla bla bla", ceux qui sont directement dans le programme car il n'y avait plus de place en EEPROM pour les y loger. Nous savons que chaque caractère enlevé dans une chaîne de caractère, donc dans du texte, se traduit par un gain d'un octet. C'est bien peu de chose, toutefois il ne restait à ce stade du développement que peu d'options. Alors compactons. Les seules chaînes de caractères bien dodues encore présentes dans le code source sont celles relatives à la transmission des données sur la ligne série USB. On enlève des points '.', on modifie quelques mots et c'est déjà un petit tas d'octets récupérés. Le texte perd un peu en élégance, encore que si vous comparez l'affichage actuel de la Fig.21 avec celui de la Fig.5 en page 4 du manuel, la différence n'est pas frappante.

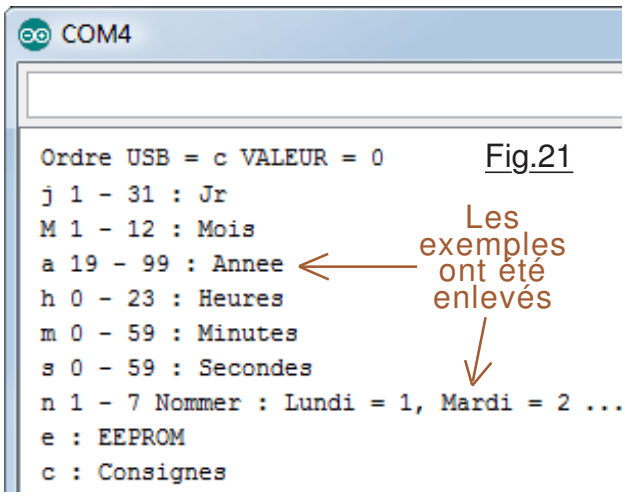


Fig.21

Continuons par le LOGO, c'est à dire dans notre cas le lépidoptère.

- Chuuuuutttt, Dudule ne doit pas le savoir !

➢ Le jeu des sept différences.

Observez attentivement les deux dessins de la Fig.22 et trouvez les différences. Si vous avez du mal, alors c'est parfait car Dudule n'y verra que du feu ! La technique a consisté à réduire la définition verticale de l'image. En supprimant deux lignes en haut, une au milieu et trois en bas, on fait une économie de 48 0ctets, un trésor considérable en période de disette. Seule la suite du développement peut confirmer que les économies de place ainsi effectuées sont suffisantes. *(Dans l'état du moment plusieurs modifications se sont avérées indispensables et leurs coûts étaient inconnus. Il a fallu procéder par étapes.)* Couper le haut et le bas impose une chirurgie esthétique décrite dans ce qui suit, opération très indigeste car il faut redessiner certains contours et coder à nouveau la matrice de pixels.

Fig.22

➢ Trouver des Octets : Tout un art !

Reprenons les pinceaux, replaçons le tableau sur le chevalet, et c'est reparti pour peinturlurer notre belle toile binaire. La Fig.23 résume les modifications à apporter au dessin virtuel. La première modification consiste à supprimer les 16 premiers Octets de la matrice descriptive, on enlève ainsi la zone rouge du haut. Puis on efface également les 24 octets situés en fin du tableau qui correspondent aux zones rouges du bas. C'est fait, nous avons gagné 40 cellules pour le programme avec un papillon dont l'abdomen est pointu et l'aile droite plate au sommet. BERRRrrrkkkk !

Pour redonner des contours agréables à l'œil, on redessine la pointe de l'aile en passant à noir les pixels verts en **2** et en comblant le trou jaune en **1**. Entre les deux ailes en **4** on ajoute un point éclairé pour lisser la forme. N'oublions pas que le texte **RST** est dans l'image. Il faut le descendre en **3** au niveau du haut de la nouvelle image comme montré par le tracé violet. Pour redonner une forme plus ronde à l'abdomen on fait passer à "noir" les quatre pixels en **7**. L'aile dans la zone **9** n'est plus très belle, aussi on "noirci" les deux pixels roses et on comble le trou par les deux pixels jaunes. En **8** se trouvait déjà un "trou" qui n'avait pas été détecté. On en profite pour le combler. En **5** se trouvait une "grande" zone sombre. On allume les cinq pixels jaunes et l'aile devient plus belle. Enfin en **6** dominait un rectangle un peu trop géométrique. Par l'extinction des deux pixels roses on apporte une dernière touche à notre chef d'œuvre. Comme ce n'était pas suffisant on enlève la ligne en **10**. *(On gagne encore 8 octets.)* Toutes ces modifications sont faciles à "tartiner"

sur un dessin d'ordinateur, en revanche, apporter les modifications au tableau informatique est autrement plus laborieux. On commence par modifier le dessin pixel. Puis, comme montré sur la Fig.24 on "retourne les tranches verticales". À partir de ce nouveau modèle, Octet par Octet dans le programme on va repérer les définitions qui changent pour les corriger. Travail particulièrement indigeste qui pour le maigre salaire de 40 Octets a exigé environ deux heures. C'est le prix à payer quand on a dilapidé les ressources lorsque nous pensions que 30720 est un nombre obèse. Force est de constater qu'à un moment où à un autre la facture nous est présentée, elle est salée. Et encore, rien ne prouvait à ce stade que ce serait suffisant. Pour terminer ce chapitre, réjouissons-nous que l'image pour sa construction était paramétrée.

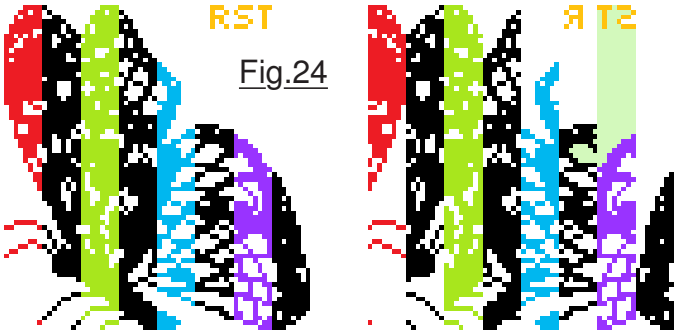


Fig.24

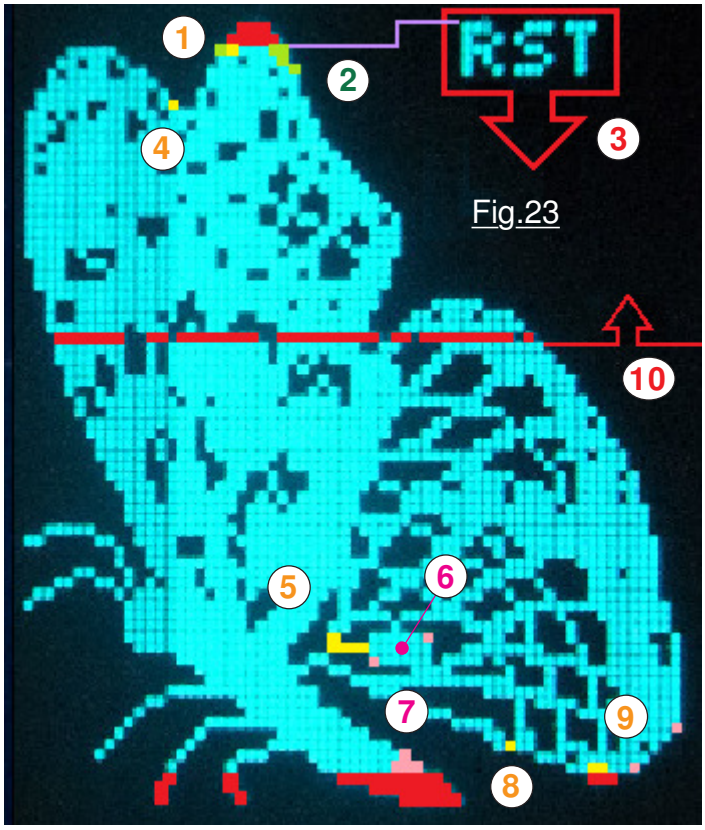


Fig.23

En effet, il ne sert à rien de diminuer la taille de la matrice, si pour construire le dessin on va lire autant d'Octets dans l'ATmega328. D'une façon générale, paramétrer les constantes est à la base de l'efficacité d'un programme. Toutes les constantes devraient se voir décrites en tête de listage sous forme de #define. Pour résumer nos économies sur le PAPI ... lépidoptère :

```
#define Largeur_du_dessin 64 // Largeur de la matrice.
#define Hauteur_du_dessin 67 // Hauteur de la matrice.
```

Pour les programmeurs qui le désirent, PAPILLON.ino ***contient la version initiale du dessin et permet diverses expérientations.***

Version	image (Pixels)	Lignes	Colonnes	Octets
Initiale	4672	73	64	584
Réduite	4288	67	64	536

➢ *Épilogue, conclusion.*

Bien qu'il soit totalement exclus de vous livrer un programme qui comporterait des "vermines" connues, attendre des mois pour s'assurer qu'il tient parfaitement la route n'est pas envisageable non plus. Après trois semaines de fonctionnement sur site et dans des conditions réelles, la petite station scientifique fait preuve de bonne volonté. Aussi, on peut raisonnablement penser que globalement le programme est sain. Bien qu'un "couf" ultérieur soit potentiellement possible, en l'état le programme d'exploitation me semble assez fiable pour vous le confier. Avec 100% d'espace programme occupé, j'ai par ce croquis battu mon record d'altitude. Il nous reste à espérer que comme Icare, le logiciel ne perdra pas ses plumes en s'approchant ainsi exagérément de l'astre diurne. Si un jour d'aventure "la cire fond" et que le programme s'égare, alors on rebranchera le ligne USB et sur l'**IDE** on replacera l'ouvrage. Si vous constatiez une vermine, n'hésitez pas à me la signaler, ce sera avec plaisir que si je dispose du temps nécessaire, je reprendrais avec conviction l'analyse de cette poésie si particulière que l'on nomme "programme".

Chaleureusement : Nulentout.