Chapitre 26 - Pourquoi le programme ne fonctionne-t-il pas ?

Avec ce que vous avez déjà appris en électronique programmable, vous êtes capables de concevoir vos propres programmes. Il est assez rare que ceux-ci fonctionnent du premier coup comme on le voudrait. Les erreurs de syntaxe sont détectées lors de la vérification du programme par le compilateur et sont les plus simples à corriger. Par contre, lorsque le programme est correctement compilé mais ne donne pas les résultats escomptés, c'est plus difficile de trouver la cause. Ce chapitre est là pour vous expliquer tout un tas de raisons qui peuvent entrainer des dysfonctionnements.

Codage de l'information

Bit et byte (ou comment sont stockées les informations)

Un circuit électronique est capable de réagir en fonction de l'absence ou de la présence d'une tension. Absence de tension = 0 V (on marquera cet état par 0) et présence de tension = 5 V (par exemple). On marquera ce dernier état par 1.

Un bit est donc la plus petite information élémentaire qu'on puisse donner à un circuit électronique. Un bit ne peut donc avoir que deux valeurs 0 ou 1 ; c'est un peu juste pour faire des choses compliquées. On a donc inventé **le byte (octet en français) qui est la juxtaposition de huit bits (c'est pour cela qu'on l'appelle octet)**.

Imaginez huit petites cases que vous pouvez remplir de 0 ou de 1 à votre convenance : combien pourriez-vous faire de possibilités ? Et bien 256 (soit 2 à la puissance 8 ou encore 2 x 2 x 2 x ...x2, 8 fois de suite). Ainsi, si vous mettez des 0 dans toutes les cases, vous aurez un octet égal à 0, et si vous mettez 1 dans toutes les cases votre octet est égal à 255, ce qui fait au total 256 possibilités. Et vous pouvez avoir toutes les possibilités entre 0 et 255.

256 possibilités, c'est déjà mieux que 2. Peut-on faire mieux ? Oui car je peux dire à mon programme qu'il doit examiner deux octets pour déduire un nombre. Dans ce cas, j'ai donc 16 bits et 2 à la puissance 16 possibilités (soit un nombre compris entre 0 et 65535). Et en prenant encore plus d'octets, on peut traiter des nombres encore plus grands, positifs ou négatifs et même des nombres à virgule (qu'on **remplace par un point en informatique** (notation anglo-saxonne)).

256 possibilités pour un octet, c'est amplement suffisant pour coder les lettres de l'alphabet (qui ne sont que 26) ; oui, on peut même mettre majuscule ou minuscule, et même d'autres caractères comme la virgule, le point, etc. Un même octet peut être un nombre (par exemple 65) ou bien un caractère (par exemple A) ; c'est le programme qui fait la différence car il sait (c'est vous qui lui avez appris) que dans telle partie, il traite un nombre, dans telle autre partie, il traite un caractère. Et il ne les traitera pas de la même façon !

Ces octets sont écrits en mémoire du microcontrôleur et sur Arduino, il y a trois types de mémoires : la mémoire programme (là où votre programme est écrit, car un programme, c'est une suite d'octets), la mémoire RAM où on stocke les données qui peuvent évoluer et qu'on appelle variables (mais cette mémoire s'efface dès qu'Arduino n'est plus alimenté) et la mémoire EEPROM (l'information persiste même après coupure de l'alimentation) qui permet de stocker des informations qu'on veut retrouver même après une coupure d'alimentation.

Le programme doit toujours savoir si un octet doit être considéré comme un simple nombre entier, un nombre réel ou bien un caractère. C'est pourquoi il est nécessaire de **déclarer les variables au début d'un programme**, ce qui permet au compilateur de réserver un espace mémoire suffisant pour stocker la valeur de la variable.

Les types de variables

On appelle **variable** une donnée dont la valeur peut **varier au cours du programme** et on appelle **constante** une donnée dont la valeur est **fixe pendant tout le programme**. Néanmoins, le terme variable est utilisé pour désigner toutes données utilisées dans un programme, que sa valeur soit constante ou variable. Cette variable peut être un nombre entier ou un nombre réel ou encore un caractère alphanumérique ; cette différence est ce qu'on appelle le **type de variable**.

La mémoire vive pour stocker des données est limitée ; dans la plupart des programmes simples, cette limite ne sera jamais atteinte. Mais dans des programmes plus complexes, on peut arriver à ne plus avoir assez d'espace mémoire, ce qui peut faire planter votre programme ! **Il est donc important de bien choisir le bon type de variables** pour ne pas gaspiller de l'espace mémoire et pour ne pas avoir de choses bizarres lors des calculs que nous ferons.

Pour suivre ce que je vais vous dire maintenant, consultez la page **Reference du site officiel Arduino** car pour ma part, je vais simplifier. Voici **différents types de variables qu'il faut bien connaître** :

byte : le programme réserve **un octet en mémoire** pour stocker la variable ; celle-ci doit donc être comprise entre 0 et 255 (voir plus haut). Le type byte est parfait pour des compteurs de boucles s'ils ne dépassent pas 255.

int : le programme réserve **2 octets** cette fois, soit 16 bits (4 octets soit 32 bits pour une carte Due, mais ne compliquons pas et supposons qu'on travaille avec Uno ou bien Mega2560). Le bit le plus fort (le plus à gauche si vous visualisez) est le bit de signe, le reste est la valeur. Je passe sur la technique du complément à deux, elle ne vous apprendra rien dans un premier temps sauf de vous embrouiller l'esprit. La valeur est donc comprise entre -32768 et +32767.

unsigned int : le programme réserve toujours **2 octets** (4 sur Due) mais cette fois, la variable **n'est pas signée, elle est donc positive** et comprise entre 0 et 65535.

Comme ce n'est parfois pas suffisant pour les grands nombres, on a également :

long : le programme réserve **4 octets en mémoire** pour stocker un **nombre signé** (positif ou négatif) compris entre -2 147 483 648 et +2 147 483 647.

unsigned long : vous l'avez deviné, même chose que long (**4 octets**) mais **la valeur est positive** et comprise entre 0 et 4 294 967 295.

On a également **word** identique à unsigned int et **short** identique à int (ne me demandez pas pourquoi il y a les deux, je suis incapable de répondre...).

Tout cela n'est encore pas suffisant car on n'a que des nombres entiers naturels, or dans la vraie vie, **les nombres sont réels**, c'est à dire avec une virgule et des décimales (le fameux PI = 3,141592.....).

Pour cela, on a inventé :

float : le programme réserve **4 octets** pour stocker un nombre réel avec une **précision de 6 à 7 décimales** après la virgule qui est remplacée **par un point**. La valeur peut donc être comprise entre - 3.40... 10^38 et +3.40... 10^38. Le symbole ^ veut dire puissance.

Attention : les calculs avec float ne sont pas forcément exacts (6.0 / 3.0 peut ne pas donner 2.0, mais peut donner 1.9999999) et de plus, calculer avec float cela **prend du temps** !

double : identique à float, donc codé sur **4 octets**, ce qui n'amène aucune précision supplémentaire sur les cartes Uno, mais codé sur 8 octets sur les cartes Due ce qui permet une précision plus grande. Voilà, on a fait le tour des données numériques, mais il y a aussi d'autres types de variables :

char : le programme **réserve 1 octet** en mémoire pour stocker un caractère alphanumérique. Voir la table ASCII sur le site Arduino ; par exemple, le caractère A est un octet égal à 65.

boolean : uniquement deux valeurs TRUE ou bien FALSE, ce qui **prend un simple petit octet**. On verra par la suite à quoi cela peut servir (les fameux tests).

array : ce sont des tableaux, c'est-à-dire une suite de données ayant la même propriété (le même type, le même profil). On accède à ces données par **un index qui commence toujours par zéro**. On verra par la suite l'utilité des tableaux.

Voyons un exemple pour terminer ; si vous voulez déclarer la broche sur laquelle vous branchez une diode LED, vous pouvez utiliser int ou même unsigned int (vous le verrez dans de nombreux programmes), mais vous utilisez deux octets alors qu'un seul suffirait en utilisant byte ! Voilà, c'est comme cela qu'on peut économiser la place en mémoire !

Il est extrêmement important de bien comprendre comment la place mémoire est occupée par les variables : en effet, on peut très vite déborder de cette place allouée en mémoire et écraser d'autres données importantes dans le programme qui plantera inexorablement.

À retenir sur le codage de l'information :

- L'information (quelle que soit sa forme) peut être représentée avec un ou plusieurs octets.
- Pour que le programme fasse la différence entre un nombre et un caractère, il faut lui indiquer le type des variables et déclarer celles-ci dans un programme.
- En fonction du type de variable, la place en mémoire pour stocker la valeur varie de 1 à 4 octets.
- Parmi les types de variables, on a les entiers signé ou non, les nombres réels forcément signés, les caractères alphanumériques, les tableaux pour chaque type.
- Il faut prendre l'habitude d'économiser la place en mémoire et pour cela, choisir le bon type de variable.

Systèmes de numération

Après avoir vu que les octets constituent des nombres, voici comment passer d'un système de numération à un autre. Nous nous limiterons pour ce chapitre à des octets non signés, c'est-à-dire compris entre 0 et 255. Ces notions vous seront utiles pour mieux comprendre les programmes trouvés sur le net ou dans des livres.

Système décimal

On compte tous en décimal, ce qui signifie qu'il y a dix chiffres (0 à 9) pour constituer un nombre.
Prenons le nombre 137 par exemple. Si on le décompose, on a 1 centaine (10^2), 3 dizaines (10^1) plus 7 unités (10^0). On rappelle ici que $10^1 = 10$ et $10^0 = 1$.
Donc, quand on regarde les chiffres qui composent un nombre décimal, **de la droite vers la gauche**, on a :
le nombre d'unités (nombre de fois 10 puissance 0)
le nombre de dizaines (nombre de fois 10 puissance 1)
le nombre de centaines (nombre de fois 10 puissance 2)
etc.

L'avantage du décimal est qu'il nous parle car nous y sommes habitués.

Système binaire
Là, il n'y a que deux chiffres possibles, 0 et 1.
Un octet est une combinaison des différentes possibilités avec 8 cases comprenant chacune 0 ou 1.
Par exemple l'octet 11100101 ; c'est un nombre composé de chiffres égaux à 0 ou 1.
Si on le regarde **de la droite vers la gauche**, le nombre se décompose en :
1 fois 2 à la puissance 0 (tout nombre à la puissance 0 vaut 1)
0 fois 2 à la puissance 1 (soit 0 x 2 = 0)
1 fois 2 à la puissance 2 (soit 4)
0 fois 2 à la puissance 3 (soit 0 x 8 = 0)
0 fois 2 à la puissance 4 (soit 0 x 16 = 0)
1 fois 2 à la puissance 5 (soit 1 x 32 = 32)
1 fois 2 à la puissance 6 (soit 1 x 64 = 64)
1 fois 2 à la puissance 7 (soit 1 x 128 = 128)
En faisant la somme, notre octet vaut :
1 + 0 + 4 + 0 + 0 + 32 + 64 + 128 = 229 (en décimal)
11100101 en binaire vaut 229 en décimal.

L'avantage du binaire est de matérialiser les 8 bits d'un octet ; ceux-ci peuvent représenter une certaine information comme la direction d'une ligne d'entrée-sortie (0 pour entrée et 1 pour sortie dans le registre de direction DDR d'un PORT, ce qui sera évoqué au chapitre 18).

Système hexadécimal
Ce système compte avec 16 chiffres, de 0 à 9, puis A, B, C, D, E et F, ce qui fait bien 16 symboles.
Par exemple A en hexadécimal vaut 10 en décimal ; B vaut 11, C vaut 12, etc.
Prenons le nombre 3FF en hexadécimal. Si on le décompose, on a 3 fois 16^2, F (c'est-à-dire 15) fois 16 et F (c'est-à-dire 15) unités, soit une somme de 1023.

Le passage du binaire en hexadécimal est très simple. Reprenons notre octet 11100101 et décomposons le en deux fois quatre bits ; nous obtenons 1110 et 0101.
1110 vaut 14 soit E en hexadécimal (c'est expliqué plus haut)
0101 vaut 5
En hexadécimal, cet octet vaut E5
Ou encore (E étant égal à 14) : 14 X 16 + 5 = 224 + 5 = 229.

Pour un humain, avec un peu d'habitude, on passe très vite de quatre bits au chiffre hexadécimal de 0 à F ; en effet pour un groupe de 4 bits, **en regardant de gauche à droite cette fois**, on voit combien de fois 8 + combien de fois 4 + combien de fois 2 + combien de fois 1. Finalement, E5 est plus digeste que 11100101 !!!

Les adresses des espaces mémoire sont toujours données en hexadécimal ; la mémoire flash de notre microcontrôleur vont de 0000 à 03FFF (soit 16383). Vu que la mémoire flash occupe 32 kilo-octets, on peut se demander pourquoi l'adresse la plus haute n'est pas 7FFF (soit 32767) : la solution dans un prochain chapitre !

La mémoire SRAM est de 2048 octets ; elle devrait donc s'étendre de 000 à 7FF, mais en fait elle commence à l'adresse 100 et finit donc à l'adresse 8FF. La mémoire EEPROM est de 1024 octets et s'étend de 000 à 3FF.

L'avantage de l'hexadécimal est qu'il est plus digeste que le binaire, surtout pour des mots constitués de plusieurs octets.

Beaucoup de calculatrices font la transformation hexadécimal en décimal ou binaire et réciproquement (notamment la calculatrice de Windows (ou Mac ou Linux, pour ne vexer personne) en affichage programmeur !). Vous pouvez essayer par vous-même et vous verrez notre nombre (229) affiché en binaire et en hexadécimal.

Le « langage » d'Arduino connait les différents systèmes de numération et ne se trompe jamais, contrairement à un programmeur qui peut, par inadvertance, entrer la valeur hexadécimale d'un nombre en pensant que cette valeur est en décimale. Bien entendu, le résultat n'est pas le même et peut aussi faire planter votre programme.

À retenir sur les systèmes de numération :

- En informatique, les systèmes de numération les plus utilisés sont décimal, binaire et hexadécimal.
- Le décimal est le système que tout le monde connait et cela nous parle car nous y sommes habitués.
- Le binaire permet de mieux matérialiser les 8 bits d'un octet, sachant que chaque bit peut représenter de l'information.
- L'hexadécimal est plus digeste que le binaire surtout pour des mots constitués de plusieurs octets.
- On passe très facilement du binaire à l'hexadécimal.
- L'hexadécimal est utilisé pour exprimer des adresses d'espaces mémoire.
- La calculette du système d'exploitation d'un ordinateur permet de passer d'un système de numération à un autre.

Arduino et les calculs

Nous allons voir comment Arduino effectue des calculs. Comme vous le verrez, il peut y avoir quelques pièges et les connaître vous évitera de tomber dedans.

Addition, soustraction, multiplication et division (+, - , * , /).
Vous connaissez tous ces quatre opérations de base. Arduino sait additionner, soustraire, multiplier et diviser. Le piège vient du type des variables (voir plus haut « Codage de l'information »). Voici quelques exemples :

La division de 9 par 4 donne 2 au lieu de 2,25. En effet, 9 et 4 sont deux entiers et pour Arduino, le résultat doit donc être un entier, puisqu'on ne lui a donné aucune autre directive.

Pour éviter cela, il faut qu'au moins un des nombres de notre opération soit du type float ou double, car dans ce cas, Arduino considérera que le résultat doit aussi être de ce type.
Ainsi, 60 * 1000 donne un résultat négatif, mais 60.0 * 1000 = 60000.0.

On peut aussi arriver à ce qu'on appelle un **débordement** (overflow en anglais) si le résultat de l'opération est plus large que ce que peut contenir la donnée. Si on ajoute 1 à un entier signé (int) égal à 32767, on n'obtient pas 32768 mais -32767, tout simplement parce que 32768 ne peut pas tenir dans un entier signé. Par contre, cela aurait donné le bon résultat avec un entier non signé (qui peut être compris entre 0 et 65535).

Quand on fait un calcul, il faut choisir un type de variable dont la taille soit assez grande pour contenir le résultat.

Signe égal (=)
En algèbre, on ne peut pas écrire x = x + 1 ; cela n'aurait aucun sens.

En informatique, c'est possible et c'est parfois ce qui trouble le programmeur débutant (vraiment débutant car ceux qui ont fait un peu de basic ont déjà compris).

En fait, quand on écrit x = x + 1, cela signifie qu'on prend la valeur de x, qu'on lui rajoute 1 et que le résultat devient la nouvelle valeur de x.

Le signe = permet de stocker la valeur qui se trouve à droite du signe, dans la variable qui se trouve à gauche du signe.

X = 16 ; // signifie que X prend la valeur 16
X = X + 1 ; // X est maintenant égal à 17
(J'ai même respecté la syntaxe avec le point-virgule, et le double slash pour les commentaires !)

L'opérateur modulo (%)
Cet opérateur donne le reste de la division euclidienne qu'on a appris à l'école primaire, et qui concerne des nombres entiers.

Par exemple, si je divise 9 par 2, j'obtiens 4 et il reste 1.
X = 9 % 2 ; // X est égal à 1

Comme on l'a dit, la division euclidienne concerne les nombres entiers, donc **modulo ne fonctionne pas avec des nombres de type float**.

Autres opérations mathématiques
Carré (en anglais square) et racine carrée (en anglais square root) (sqrt ( )) :

sq (x) calcule le carré du nombre x (soit x^2 ou encore x fois x)
sqrt (y) calcule la racine carrée de y (c'est l'opération inverse du carré) (y peut être de n'importe quel type, sqrt (y) sera de type double)

Arduino est aussi capable d'élever à la puissance, de traiter des fonctions trigonométriques (sinus, cosinus et tangente) et de générer des nombres aléatoires (enfin pas complètement aléatoires, mais presque ; on les appelle nombres pseudo-aléatoires). Pour plus de détails, on se référera à la page Reference du site officiel Arduino.

Avant de terminer ce paragraphe, voyons deux autres fonctions qui peuvent avoir de l'intérêt dans notre hobby, la fonction valeur absolue et la fonction map :

Valeur absolue :
abs (x) donne la valeur absolue de x, c'est à dire x si x est plus grand que zéro (positif) et -x si x est plus petit que zéro (négatif).
abs (2) = 2
abs (-2) = 2
La valeur absolue d'un nombre est ce nombre sans considérer le signe (oh, que les mathématiciens doivent me haïr !). Prenez la définition que vous voulez, ce qui compte, c'est que ça vous parle.

Fonction map :
Cette fonction sert à ramener une valeur évoluant dans un intervalle, dans un autre intervalle. Par exemple, la mesure d'une tension sur une entrée analogique donnera un résultat compris entre 0 et

1023. Si la tension est comprise entre 0 et 5 volts, la fonction map fera la conversion.

Tension = map (analogRead(pin), 0, 1023, 0, 5) ;

En général, la fonction map sera intéressante lorsque vous utilisez des entrées analogiques.

Maintenant, vous êtes capables de comprendre pourquoi certains de vos calculs réalisés avec Arduino ne donnent pas le bon résultat.

A retenir sur les calculs :

- Les types de nombres servant aux calculs.
- Les possibilités de débordement.
- Le signe égal = qui attribue une valeur à une variable et qui est différent du double égal == utilisé dans des tests if.

Arduino et le temps

Dans un programme informatique, il est souvent nécessaire de faire une pause, c'est-à-dire de **ralentir le cours du temps pour pouvoir observer ce qui se passe**. Si on regarde la page Reference du site Arduino, nous pouvons voir que quatre fonctions existent dans le langage Arduino pour traiter le temps.

delay() : cette fonction met en pause le programme pour un certain délai exprimé en millisecondes. De même qu'il y a 1000 millimètres dans un mètre, il y a mille millisecondes dans une seconde. Donc si on veut faire une pause de 1 seconde, la fonction est delay(1000). C'est tout simple, et si vous avez déjà utilisé le programme blink (donné en exemple avec le logiciel Arduino), vous êtes habitués à cette fonction.

L'argument passé entre parenthèse (le délai en ms), est une variable de type **unsigned long** (voir plus haut ce qui a été dit sur les erreurs dues au type de variable).

delayMicroseconds() : cette fonction est identique à la précédente, sauf que le délai est exprimé en microsecondes. Il y a 1 000 000 de microsecondes dans une seconde et il y a mille microsecondes dans une milliseconde.

L'argument passé entre parenthèse (le délai en µs), est une variable de type **unsigned int** (voir plus haut ce qui a été dit sur les erreurs dues au type de variable). Pour avoir une bonne précision, cet argument doit être compris entre 3 et 16383 ; au-delà de 16383, il vaut mieux utiliser delay() en millisecondes.

L'inconvénient de ces deux fonctions est qu'on ne peut rien faire d'autre en attendant. Si vous entrez un délai trop grand dans un programme, vous aurez l'impression que votre programme ne fait plus rien et qu'il est planté. Les deux fonctions suivantes vont nous permettre de résoudre ce problème.

millis() : cette fonction retourne le nombre de millisecondes **écoulées depuis le début du programme**.

La valeur retournée est du type **unsigned long** ; **pour traiter ce résultat, il faut donc utiliser le même type de variable**. Cette valeur recommence à zéro au bout de 50 jours approximativement.

La fonction millis() peut donc servir à traiter des attentes sans pour autant bloquer le déroulé du programme. On peut en avoir un exemple avec le programme « blink without delay » donné en exemple dans le logiciel Arduino.

micros() : comme la fonction précédente, cette fonction retourne le nombre de microsecondes écoulées depuis le début du programme. La variable de type **unsigned long**, retourne à zéro après approximativement 70 minutes. La résolution de cette fonction est de 4 µs sur une carte cadencée à 16 MHz (Uno, Mega2560, Nano) et 8 µs sur une carte cadencée à 8 MHz.

Vous obtiendrez d'autres précisions à partir de la page Reference du site Arduino.

Je vous invite également à regarder les programmes donnés en exemples (notamment Blink et BlinkWithoutDelay) dans votre environnement de développement intégré Arduino.

Programme Blink :
Nous l'avons déjà commenté et c'est la façon la plus simple de faire clignoter une LED. Cependant, pendant que le microcontrôleur passe son temps à attendre, il ne fait rien d'autre.

Programme BlinkWithoutDelay :
Ouvrez ce programme
Fichier > Exemples > 02.Digital > BlinkWithoutDelay

Ce programme utilise la LED branchée sur la broche 13 déclarée en sortie dans le setup. La variable entière ledState désigne l'état de la LED, ici initialisée à LOW.

Dans le programme principal (fonction loop), on interroge la fonction millis() pour connaitre l'heure en quelque sorte, décompté depuis la mise sur on du module. Le résultat est placé dans une variable (de type unsigned long) qui s'appelle currentMillis. On lui soustrait la valeur précédente (previousMillis) et on regarde à quel moment cela devient plus grand que l'intervalle de clignotement, et dans ce cas, on change l'état de la LED.

Pour cela, on teste si l'état est LOW et dans ce cas, on le met à HIGH. Sinon, c'est qu'il est HIGH et on le met à LOW. Cette façon de programmer permet de bien comprendre ce que l'on fait, mais pour les programmeurs un peu habitué, les tests if … else peuvent être remplacés par :

```
ledState = !ledState ;
```

Rappelez-vous que **le point d'exclamation est la fonction NOT** ou INVERSE.

Voici ce que donne le programme (j'ai retiré les commentaires pour faire la copie d'écran) :

BlinkWithoutDelay_Chris

```
const int ledPin =  13;
int ledState = LOW;
unsigned long previousMillis = 0;
const long interval = 1000;

void setup() {
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // Mettre ici le code pour faire autre chose en même temps.

  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= interval) {
    // save the last time you blinked the LED
    previousMillis = currentMillis;

    ledState = !ledState ;
    digitalWrite(ledPin, ledState);
  }
}
```

En vous inspirant de ce programme, concevez un programme capable de faire clignoter deux LED à des fréquences différentes.

Pour ceux qui veulent en savoir plus, consultez cet article « Comment gérer le temps dans un programme » du site Locoduino, à cette adresse :
http://locoduino.org/spip.php?article6

Enfin, ceux qui veulent devenir expert du voyage dans le temps, peuvent consulter l'article « La bibliothèque ScheduleTable » du site Locoduino, à cette adresse (pour des gens avancés car nous n'avons pas encore parlé des bibliothèques) :
http://locoduino.org/spip.php?article116

À retenir sur la gestion du temps :

- Les quatre fonctions du langage Arduino permettant de gérer le temps.
- Que les fonctions delay et delayMicroseconds sont bloquantes, c'est-à-dire que le module ne fait rien d'autre que d'attendre.
- La façon d'utiliser millis et micros pour résoudre le problème précédent.

Autres erreurs de programmation à éviter

Afin de ne pas confondre les variables entre elles, donnez-leur des noms explicites même si cela augmente le nombre de caractères à taper pour écrire le programme ; en cas de problème, vous verrez mieux ce qui se passe, et au cas où vous abandonnez votre programme un certain temps, ce sera plus facile de le reprendre.

Méfiez-vous des boucles qui tournent indéfiniment parce que la valeur du compteur a été trafiquée au cours du programme ou bien parce que la condition pour en sortir ne se présente jamais. L'utilisation du moniteur permet d'afficher les valeurs de certaines variables et ainsi de comprendre ce qui se passe.

Apprenez à structurer votre programme comme s'il était composé de petits modules ; chacun d'eux sera ainsi plus facile à déboguer. Enfin, rappelez-vous qu'écrire un programme ne peut se faire que si l'analyse des tâches à accomplir a été faite sérieusement, en prenant la peine de comparer chaque variable entrée avec un intervalle de crédibilité ; par exemple, si votre programme a besoin du nombre de locomotives sur votre réseau, vérifier que ce nombre est inférieur à 255 parce qu'il n'y a pas à ma connaissance de réseau capables de faire circuler plus de 255 trains en même temps. Vous serez ainsi assuré que l'octet réservé à la valeur de cette variable n'a pas débordé sur une autre variable en mémoire.