PICAXE BASIC
pour débutants

(en français s'il vous plait...)

Sommaire

1 Préambule

1.1 Audience

Ce guide s'adresse avant tout à celles et à ceux qui souhaitent pour la première fois écrire un programme en BASIC pour PICAXE.

Ceux qui connaissent déjà un autre langage y trouveront sans doute les quelques particularités qui leur permettront d'être rapidement opérationnels sur ces petits processeurs.

Ceux qui programment déjà en mode graphique trouveront ici des notions de base importantes pour structurer un programme et profiter de fonctions plus puissantes.

Au début, vous allez sans doute vous dire que ce texte insiste sur des détails pour des programmes de quelques lignes. (alors que d'autres manuels de BASIC vont rapidement à des choses plus complexes) C'est voulu : le but d'un PICAXE, c'est de manipuler des tensions sur des "pattes" de circuits intégrés. Pas de retoucher des photos (ce que l'on peut faire en BASIC sur un ordinateur ...)

Donc il est important de bien comprendre ce qui se passe à chaque étape sans quoi votre programme risque au mieux de "tomber en marche", mais plus probablement de ne pas du tout fonctionner.

1.2 BASIC

Comme son nom l'indique, le BASIC a été conçu pour être « facile à utiliser » par opposition à d'autres langages jugés plus complexes à mettre en œuvre.

La conséquence de cette approche est que BASIC fait beaucoup de choses « par défaut ». Ceci peut être très troublant pour les esprits les plus rigoureux. C'est pourquoi, même pour les exemples les plus simples, il est utile d'indiquer ce qui a été pris « par défaut » ainsi que la syntaxe complète quand c'est utile.

Ce manuel est très loin d'être exhaustif ; en particulier, les explications données pour chaque commande se limitent au cas les plus courants. De même, le BASIC du PICAXE regorge de commandes spécifiques adaptées au pilotage de tel ou tel périphérique. Seules les commandes qui structurent l'essentiel des programmes sont expliquées.

Pour toutes ces raisons, le recours au manuel de référence demeure indispensable. Un effort est en cours pour le traduire.

1.3 PICAXE

Les PICAXES sont des microcontrôleurs PIC capables d'interpréter le langage BASIC. Un chapitre entier est nécessaire pour expliquer tout ce que cela implique. Mais pour commencer retenez simplement :

- qu'ils sont faciles à programmer,
- que ce sont des outils pédagogiques formidables,
- que certaines fonctions sont très performantes,
- que pour d'autres, ils sont au moins 1000 fois plus lents qu'un microcontrôleur compilé.

2 Guide pas à pas

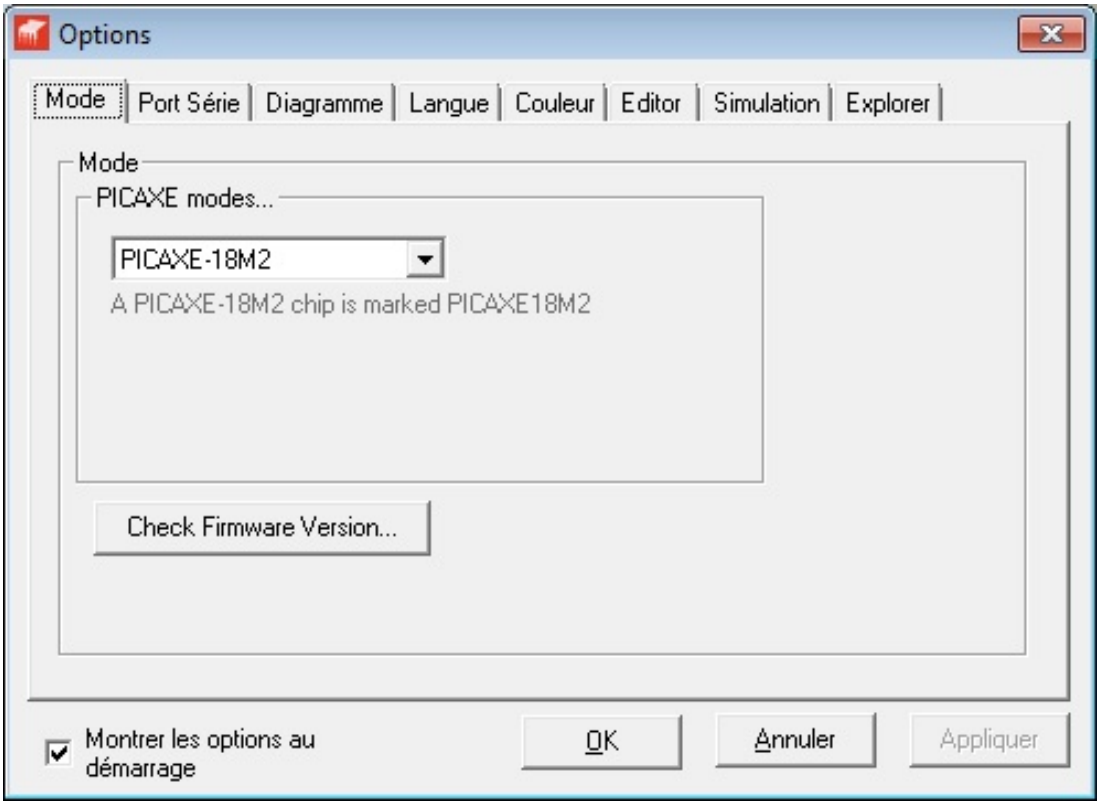
Tous les manuels d'initiation à un langage commencent par ce test simplissime : écrire quelques mots.

C'est important parce que c'est une bonne façon de « voir » ce qui se passe. Si votre programme n'affiche jamais rien, peut-être fonctionne-t-il parfaitement, peut-être pas, comment savoir ?

Supposons donc que vous disposez d'un ordinateur sur lequel est installé « Programming Editor » (PE pour les intimes). (Si ce n'est pas encore le cas, voir…)

Vous êtes fortement invités à exécuter les manipulations indiquées dans PE et pas seulement à lire le manuel. La mémoire ne fonctionne pas seulement avec les yeux, les doigts aussi participent...

Quand vous lancez PE, la première fenêtre qui s'affiche est la suivante :



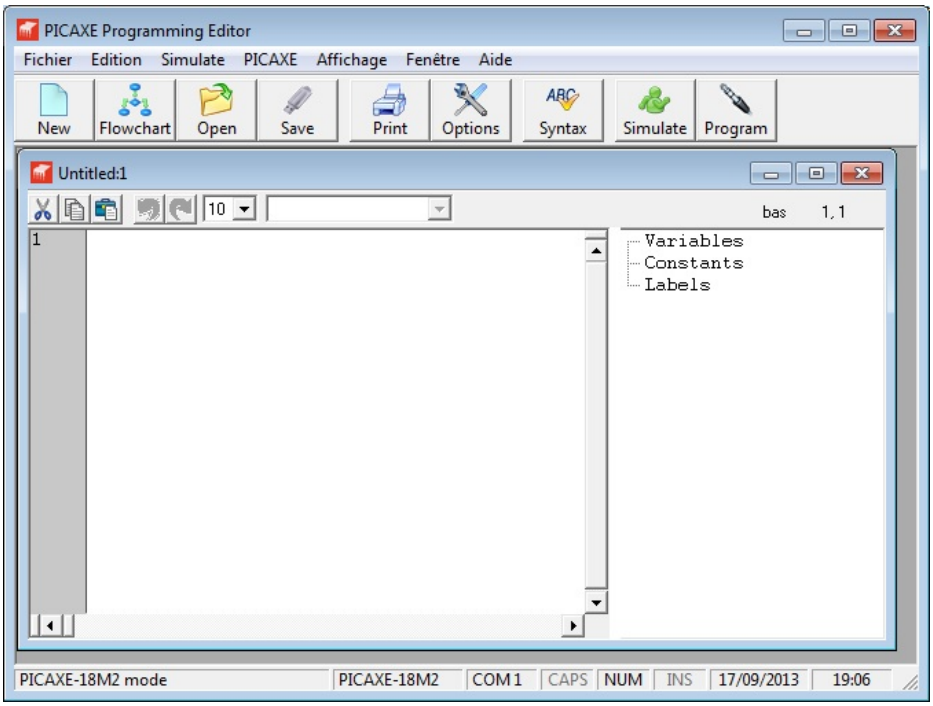
C'est assez logique ; vous êtes censés programmer un PICAXE. Donc PE vous demande lequel ?

Si vous disposez d'un modèle précis, vous pouvez le choisir.

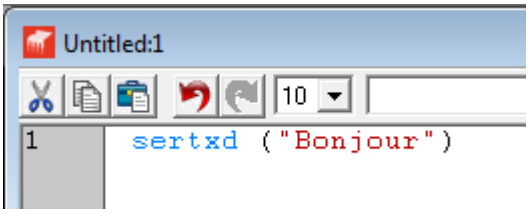
Mais le but c'est d'apprendre à écrire des programmes et on peut le faire même sans PICAXE réel.

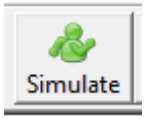
Cliquez "OK"

Vous devriez maintenant voir ceci :

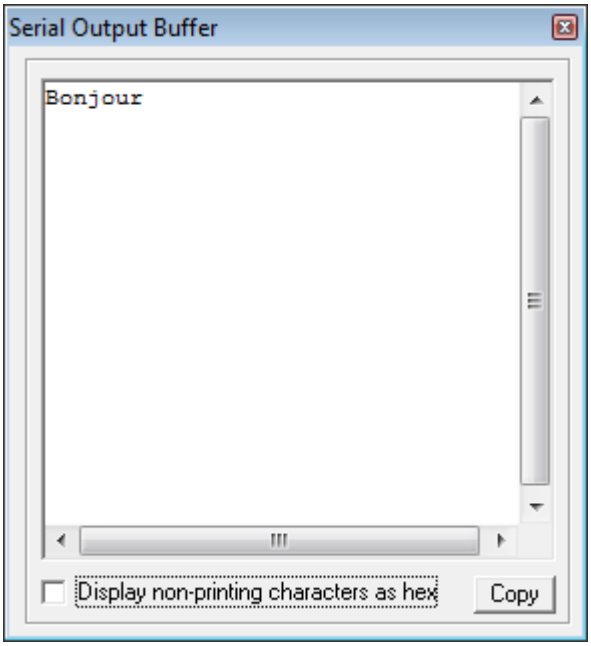


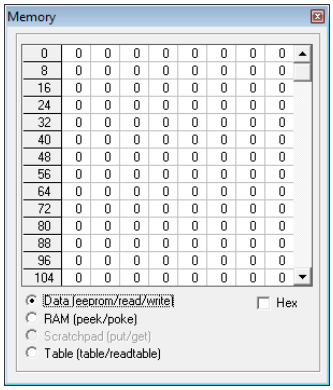
Ecrivez dans la page blanche :

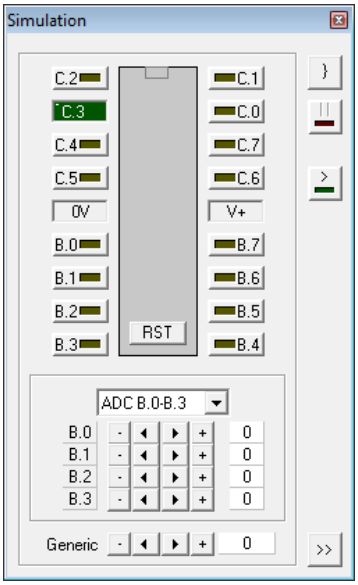


puis appuyez sur le bouton 

Vous devriez voir apparaitre 3 fenêtres :





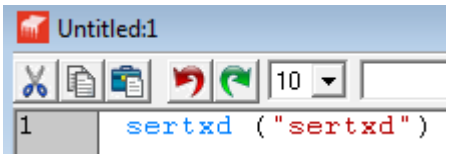


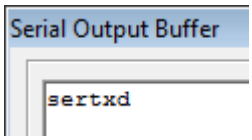
Seule la première nous intéresse pour le moment.

2.1.2 Que s'est-il passé

Vous constatez que le texte "Bonjour" a été écrit dans la fenêtre "Serial Output Buffer" qui se traduit par "Fichier de sortie du port série".

Première remarque : c'est le texte Bonjour sans les guillemets qui a été écrit. Les guillemets sont des séparateurs qui indiquent que **PE NE DOIT PAS** chercher à interpréter ce qui est écrit entre les guillemets. Il l'écrit, point final, sans chercher à comprendre. On pourrait très bien écrire :



et on obtiendrait : 

Vous constatez que PE a bien interprété le premier sertxd, mais il a simplement écrit le second sans chercher à comprendre ce qui est écrit entre les guillemets.

Deuxième remarque :

La commande **sertxd** signifie quelque chose comme "transmettre ce qui suit sur le port série".

Un "port série" est un fil électrique sur lequel on peut envoyer des mots. Evidement, on suppose que le PICAXE va envoyer ces mots sur le fil et que "quelque chose" va les recevoir.

Et bien dans notre cas, ce "quelque chose", c'est le "serial output buffer"

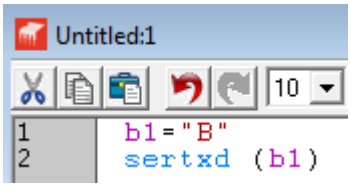
Le bouton 

sert à simuler le fonctionnement d'un véritable PICAXE. Ceci fait gagner beaucoup de temps quand on est au stade de l'apprentissage parce que la procédure qui permet de faire la même chose avec un PICAXE réel est un peu plus longue.

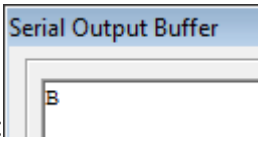
2.2 Variables

2.2.1 Le programme

La maison ne reculant devant aucun sacrifice, voici un programme deux fois plus complexe :



Maintenant vous savez faire : bouton 

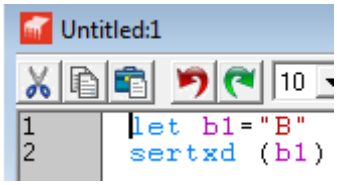
et résultat dans : 

2.2.2 Que s'est-il passé

La première ligne sert à mettre le texte "B" dans la "variable" b1.

Une variable est un emplacement de la mémoire du PICAXE qui peut contenir un nombre entier compris (inclusivement) entre 0 et 255.

Première remarque pour les puristes : il manque l'instruction LET. On aurait pu écrire (mais plus personne ne le fait) :



et le résultat serait rigoureusement le même !

LET signifie "fait en sorte que la variable b1 prenne pour valeur ce qui suit le signe ="

Il est implicite (ou par défaut si vous préférez). Voir chapitre 1.2

Deuxième remarque :

Mais comment une variable censée contenir un nombre peut-elle en fait accepter une lettre ?

Bonne question : je vous remercie de l'avoir posée.

Et bien en fait, une lettre ou un nombre entre 0 et 255, pour un PICAXE, c'est exactement la même chose !

Et même pour être plus précis, il a en fait pour la lettre B, le nombre 66 (diable !) soit `1000010`.

C'est exactement ce qui va passer sur le fil dont on parlait tout à l'heure (en partant de la droite) : 0 volts, puis 5 volts, puis 4 fois 0 volts, puis 1 fois 5 volts et le tour est joué.

Evidement, pour que nos petits yeux y comprennent quelque chose, c'est le récepteur (à savoir le "serial output buffer") qui va habiller tout ça :

1) grouper toutes ces tensions par paquets de 8,

2) convertir le nombre binaire en décimal : 66

3) regarder dans la table ASCII à quelle lettre correspond ce nombre : "B"

Troisième remarque :

L'instruction **sertxd** accepte aussi bien une constante (le "Bonjour" de tout à l'heure)

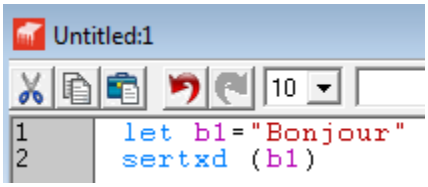
que notre variable b1.

Que ce serait-il passé si nous avions mis des guillemets autour de "b1" ?

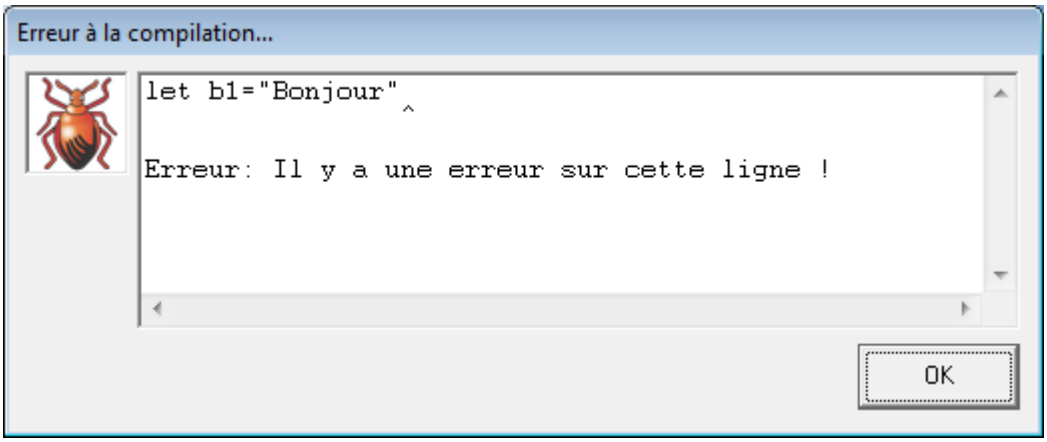
Je vous laisse expérimenter... et relire le § 2.1.2 si nécessaire.

2.2.3 Oui mais...

Ce qui serait bien, ce serait d'écrire :



Ce qui donne :



Nous abordons là un point important :

> On apprends autant (sinon plus) en commettant des erreurs qu'en écrivant juste du premier coup.

L'important est de comprendre pourquoi ça ne marche pas...

Dans le cas présent, l'explication figure quelques paragraphes plus haut :

> Une variable est un emplacement de la mémoire du PICAXE qui peut contenir UN nombre entier compris (inclusivement) entre 0 et 255.

Le mot "Bonjour" est en fait une succession de lettres, donc de nombres entiers. La variable b1 ne peut en contenir qu'un seul (ou une seule lettre, c'est pareil).

Considérez pour le moment qu'un PICAXE est plus à l'aise avec les nombres entiers qu'avec les mots. Seules les constantes (comme dans le tout premier exemple) permettent d'agrémenter les "sorties", mais c'est bien suffisant la plupart du temps.

2.2.4 Pour aller plus loin

Un PICAXE peut manipuler trois types de variables :

- les nombres entiers (entre 0 et 2^8-1=255)

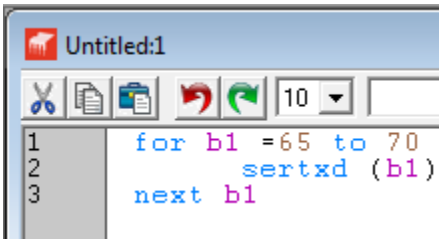
- les grands nombres entiers (entre 0 et 2^16-1=65535)

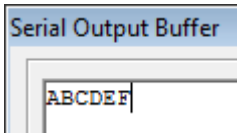
- les nombres binaires (soit 0, soit 1, mais ça peut signifier vrai/faux ou blanc/noir, ce que vous voulez...)

Sur un PICAXE, il y a des particularités d'utilisation. Donc nous verrons ça plus tard.

2.3 Boucles

2.3.1 Le programme



ce qui donne 

2.3.2 Que s'est-il passé

Certains ont essayé de faire un BASIC en français. C'est une fausse bonne idée. Certes les débutants y voient plus clair. Mais tôt ou tard ils devront passer à l'anglais. Quand aux "initiés" censés les aider, ils ne retrouvent plus leurs petits et se retrouvent en situation de "débutants"...

Donc, on n'y coupe pas, il faut se faire à l'anglais.

for b1=65 to 70 signifie quelque chose comme

Pour b1 prenant les valeurs successives de 65 jusqu'à 70

La deuxième ligne, vous la connaissez déjà.

Quand à la troisième ligne, **next b1**, comprenez

b1 suivant

Première remarque :

Les instructions **for** et **next** constituent un ensemble, un bloc.

Le PICAXE va répéter tout ce qui se trouve entre le **for** et le **next** en incrémentant **b1** depuis la valeur de départ jusqu'à la valeur finale (y compris).

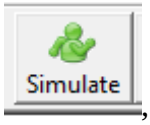
C'est pour cela qu'il est recommandé de décaler les instructions qui seront répétées pour bien mettre en évidence ce bloc.

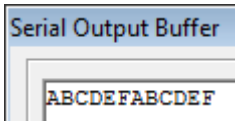
Deuxième remarque :

Le "B", c'est le 66 donc le "A" c'est le 65 !

Si vous ne comprenez toujours pas pourquoi ce programme écrit le début de l'alphabet, relisez ce qui précède...

2.3.3 Oui mais...

Si on appuie une deuxième fois sur le bouton 

, on obtient : 

Si on souhaite que le "serial output buffer" ale à la ligne, et bien il faut le lui dire !

Si vous ne le savez pas, vous ne pouvez pas le deviner :

Le codage des lettres en nombres obéit à une norme internationale largement respectée : le code ASCII

C'est purement arbitraire, le A pourrait valoir 78 ou 33, ça marcherait pareil, mais ça serait un sacré bazar. Donc tout le monde s'est mis d'accord et le "A" c'est 65 épicétou.

Pour les autres lettres, regarder **ASCII table** dans le menu "Aide".

Vous remarquerez que les lettres avant le 32 qui est l'espace (autrement dit un caractère tout blanc qui sert à séparer les mots) portent des noms bizarres : CR, LF, ESC,...

En fait il s'agit d'acronymes anglais qui datent des machines à écrire...

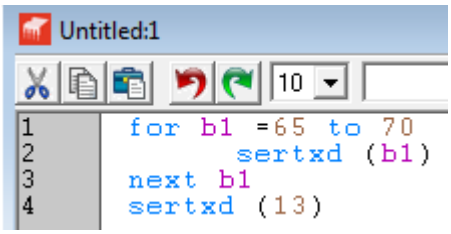
CR signifie **Carriage Return**, autrement dit : "retour au début de la ligne"

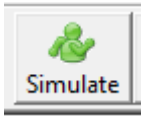
LF signifie **Line Feed** , soit "avance d'une ligne"

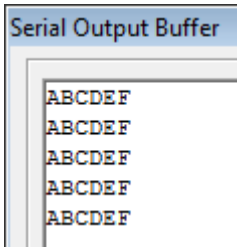
Donc en toute logique, pour revenir à la ligne, il faut faire CR LF.

Comme les informaticiens sont de grands paresseux, il font maintenant en sorte que le CR avance aussi d'une ligne et remplace donc la séquence CR LF.

Le programme devient :



et produit avec quelques 



2.3.4 Toujours plus fort !

Vous savez maintenant écrire les lettres de l'alphabet ASCII. Mais comment écrire les nombres correspondant ?

C'est là qu'intervient le fameux ouvrage :

picaxe_manual2.pdf

Page 210 vous pourrez lire :

Syntax:

SERTXD ({#}data,{#}data...)

- Data are variables/constants (0-255) which provide the data to be output.

Ce qui signifie que vous pouvez mettre entre les parenthèses autant de variables et de constantes que vous le souhaitez (à la place du mot **data**) en les séparant par des virgules.

Remarquez également le {#}

Les accolades encadrent un terme optionnel : vous pouvez le mettre ou pas.

Mais que se passe-t-il si vous ajouter un # devant une variable ?

En fait, les explications sont dans le paragraphe consacré à la fonction précédente :

serout

Data are variables/constants (0-255) which provide the data to be output.
Optional #'s are for outputting ASCII decimal numbers, rather than raw characters. Text can be enclosed in speech marks ("Hello")

Traduction :

Data représente des variables/constantes (entre 0 et 255) qui fournissent les données à envoyer.

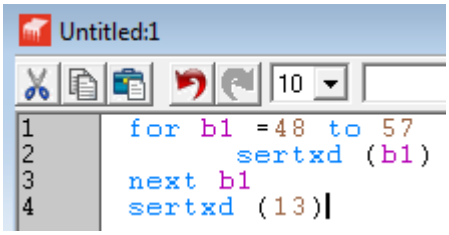
Le signe optionnel # sert à envoyer la représentation des nombres sous forme de caractères ASCII décimaux, à la place des caractères "bruts" (qui seraient envoyés sans cela).

Les textes peuvent être encadrés par des guillemets (exemple : "Hello")

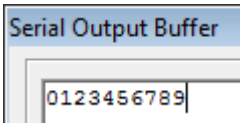
Explications :

Si vous consultez la table ASCII, vous constatez que les lettres 48 à 57 dessinent en fait les chiffres de 0 à 9.

Vous pouvez également modifier votre petit programme :

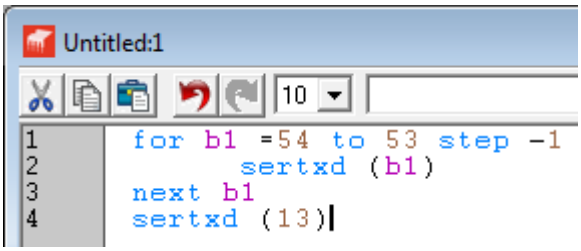


ce qui donne

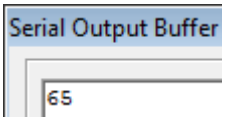


Et bien, les fameux "caractères ASCII décimaux" c'est ça.

Donc si on veut "écrire" le nombre 65, il faut envoyer les nombres 54 et 53.



donne



J'en profite pour vous présenter une subtilité de l'instruction **for next** :

en ajoutant **step** qui signifie "**par pas de**", on peut avancer de 2 en 2 ou de 3 en 3 ou de 10 en 10

ou à l'envers, c'est à dire reculer de 54 vers 53.

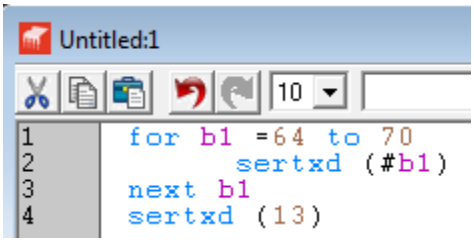
Au passage : la valeur "implicite par défaut" du mot-clé step est : 1

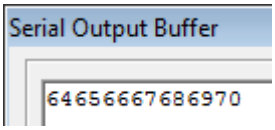
Notez que si vous ne mettez pas le **step**, la boucle s'arrête immédiatement, le critère final étant dépassé.

Mais revenons à notre #

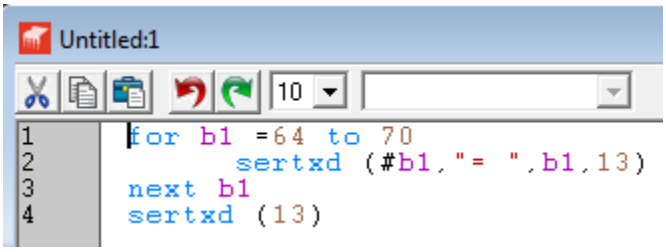
Si nous en croyons le manuel, le PICAXE sait faire tout ça par la simple présence d'un bête # avant la variable ?

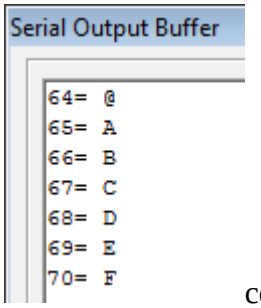
Vous êtes comme saint Thomas ? qu'à cela ne tienne :



donne


Il y a de l'idée, mais c'est encore un peu brouillon. je vous propose une chtite amélioration :



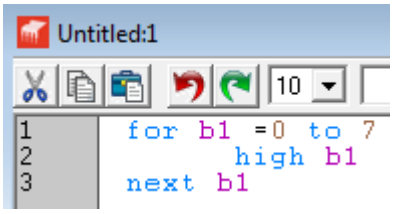
devrait donner


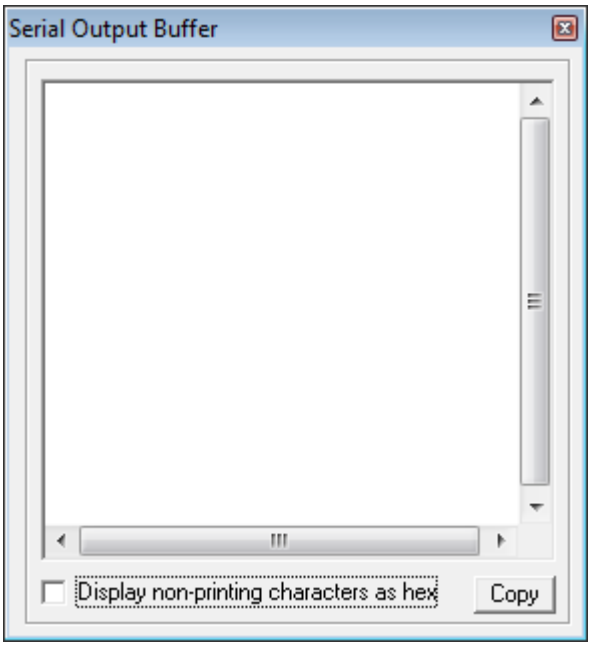
cétipabo ?

2.4 Entrées/Sorties

2.4.1 Sorties

2.4.1.1 Le programme

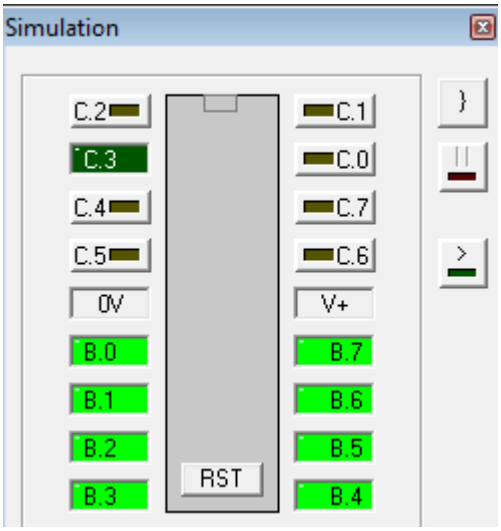


qui donne


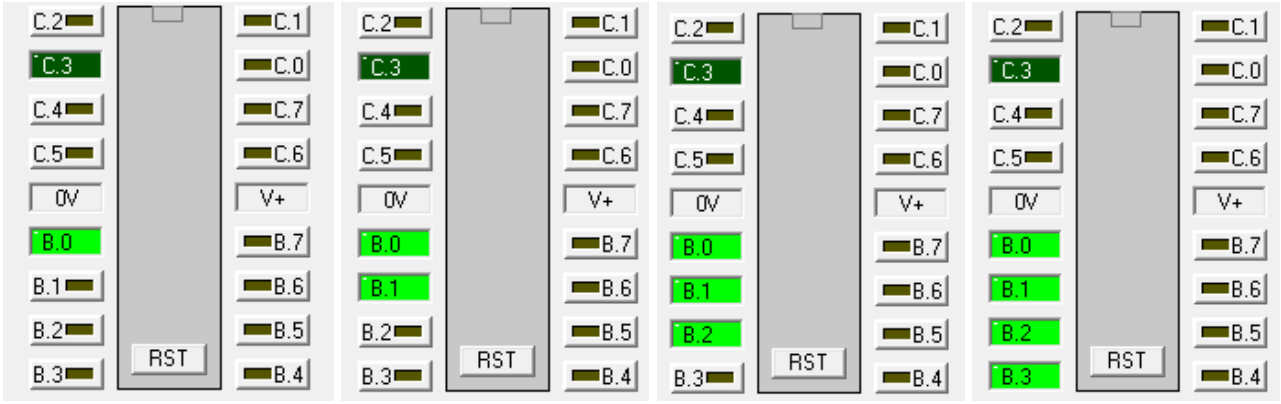
2.4.1.2 Que s'est-il passé

Rien du tout ?

Chef ! On ne regarde pas la bonne fenêtre :



et si vous êtes attentif, vous devriez voir successivement ça :



Le rectangle gris représente un PICAXE avec ses 18 pattes.

Le 0V et le V+ (=5V), c'est pour l'alimenter. (Il faut bien les nourrir ces petites bêtes...)

Les autres pattes peuvent être :

- soit à 0V (en sombre)
- soit à 5V (en vert clair)

Si vous branchez une LED sur chaque patte (via une résistance) elles vont s'allumer successivement.

Mais le simulateur est bien suffisant pour comprendre.

Une seule instruction est nouvelle : **high** qui signifie haut, c'est à dire **tension haute**.

Son opposé est **low**.

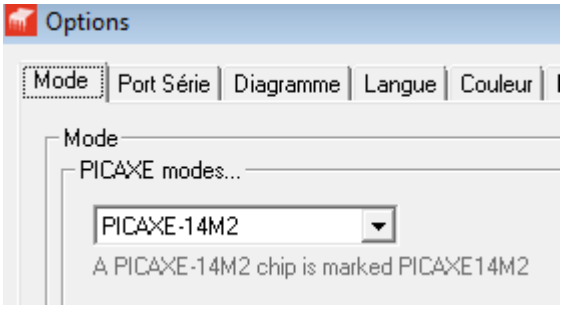
Ce qui est plus subtil, c'est la numérotation des pattes.

Premièrement, il faut savoir que "par défaut" les pattes de sortie du PICAXE sont constituées par le port B.

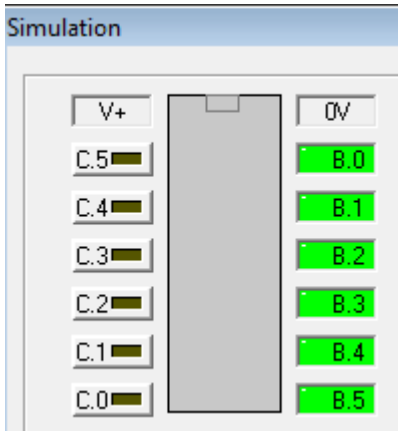
Un port comporte 8 pattes numérotées de 0 à 7.

Le manuel est indispensable pour situer ce port B suivant les différents modèles de Picaxe.

Vous pouvez aussi utiliser le menu Affichage/options et choisir un autre modèle de Picaxe :



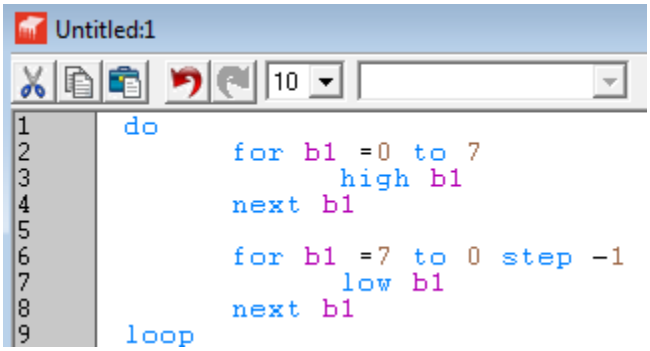
ce qui donne



Attention :

Le port B.1 n'a rien à voir avec la variable b1 : c'est une coïncidence.

2.4.1.3 Allons plus loin



Ca, c'est typiquement le genre de chose formellement prohibé sur un ordinateur...

mais très courant sur un microcontrôleur (comme un PICAXE par exemple).

Ca s'appelle une "boucle infinie".

Seule solution pour arrêter le simulateur : 

Ce bouton n'existant pas sur un PICAXE réel, il va exécuter infiniment le même programme (en tout cas tant qu'il est alimenté...) C'est normalement sans risque parce que le fabriquant a prévu une petite subtilité qui permet de lui dire par le port de programmation de s'arrêter pour recevoir un nouveau programme.

Mais si justement votre programme utilise le port de programmation, vous pouvez avoir l'impression que le PICAXE est bloqué, comme un PC qui ne répond plus.

Vous croyez vous en sortir en lui coupant le jus : erreur !

Dès que le 5V revient (il faut bien si vous souhaitez le reprogrammer) il repart de plus belle.

Seule solution : le reprogrammer juste au moment où on le remet sous tension. En effet, le fabriquant a prévu une deuxième roue de secours : le PICAXE attends un tout petit avant de démarrer le programme au cas où PE aurait besoin d'envoyer un nouveau programme.

Explications

do signifie **faire**

et **loop** signifie **boucler**

La traduction donne quelque chose comme :

Faire

ceci

celà

pi aussi ça

...

Boucler (sous entendu vers l'instruction faire)

Cette nouvelle façon de faire des boucles comporte en fait des arguments pour arrêter la boucle :

du genre :

Boucler tant que b1 est inférieur à 25

Boucler jusqu'à ce que b1 soit supérieur à 25

etc...

Mais si on n'en met pas, et bien c'est simple, le PICAXE ne s'arrête pas de boucler...

2.4.1.4 A quoi ça sert

Comme dit plus haut c'est très fréquent sur un microcontrôleur qui est par nature destiné à réaliser patiemment un tâche précise.

Par exemple un système d'alarme a toutes ses pattes reliées (via des résistances) au 5V.

Sauf que ces fils peuvent être interrompus par l'ouverture d'une fenêtre par exemple, ce qui laisse tomber la patte au 0V.

Le PICAXE va passer son temps à surveiller les pattes une à une :

Tant qu'il voit du 5V partout, tout va bien. Mais si une patte est au 0V, il faut activer la sirène. Mais ne croyez pas qu'il sorte de la boucle : il faut bien continuer à assurer la surveillance, même après une alarme.

Et même quand vous désactivez l'alarme, vous êtes coutent que le PICAXE puisse vous signaler, pas par la sirène mais par une petite LED verte, qu'une porte est restée ouverte.

Si le PICAXE doit faire plusieurs choses en même temps, par exemple surveiller les portes et en même temps faire clignoter une LED, il faut tout mettre dans la boucle principale et faire successivement les différentes tâches. Comme la boucle tourne plutôt vite, l'utilisateur à l'impression que les différentes choses sont gérées simultanément.

Sur les PICAXES récents, il est aussi possible de faire tourner plusieurs programmes en même temps : ça s'appelle du "pseudo" multitâche.

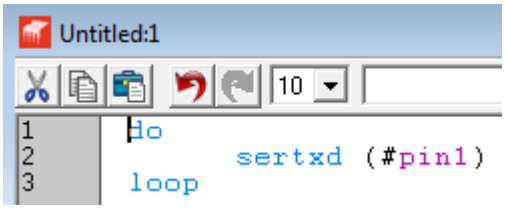
Le "pseudo" est là pour rappeler que le PICAXE a des limites dans ce domaine et que tout ne peut pas être fait en parallèle.

2.4.2 Entrées tout ou rien

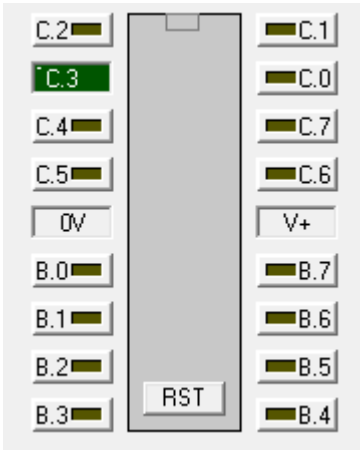
Le PICAXE est capable de générer une tension sur chacune de ses broches pour éclairer une LED, démarrer un moteur, etc...

Mais il doit aussi être capable de détecter une tension.

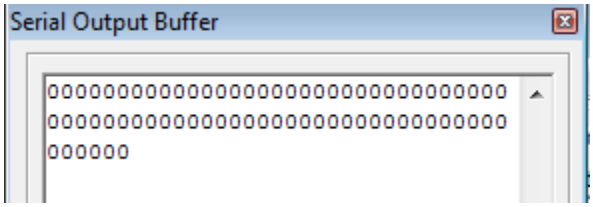
2.4.2.1 Le programme



donne



et



2.4.2.2 Que se passe-t-il ?

La variable **pin1** est définie par défaut et contient l'état de la patte numéro 1 en entrée. (Pour faire plus pro, on dit la **broche** N°1 en français...)

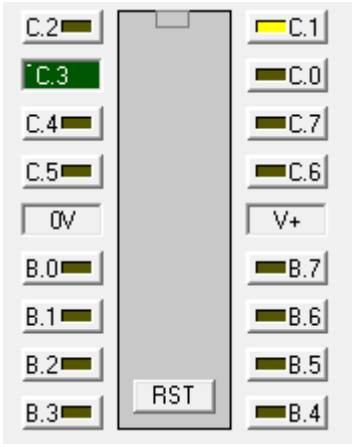
Comme pour les sorties, il y a un port par défaut qui le **port C.**

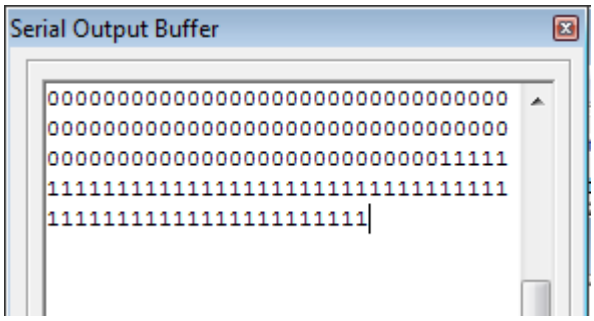
Donc pin1 représente l'état de la broche C.1

C'est pareil de pin0 à pin7 pour les broches C.0 à C.7 quand elles existent.

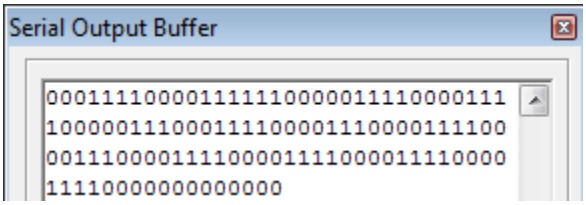
Mais comment changer la tension de ce PICAXE virtuel ?

Et bien c'est tout simple : il suffit de cliquer sur la patte en question (qui du coup nous fait une jaunisse pour indiquer qu'elle est passée à l'état haut) :



et


En cliquant plusieurs fois sur C.1, on voit bien le changement de la valeur de la variable pin1 :



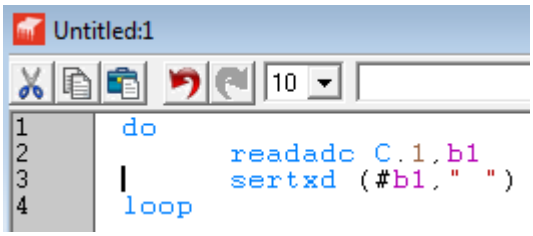
En cliquant environ une fois par seconde, ce petit test permet également de mesurer la vitesse du simulateur : on voit que la boucle tourne 4 ou 5 fois chaque seconde.

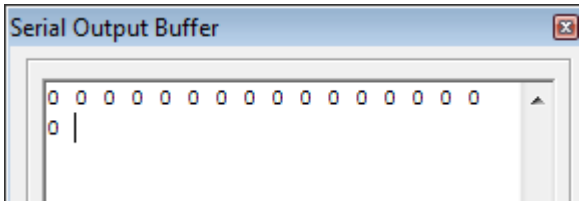
C'est spécialement lent, mais c'est bien pratique pour voir ce qui se passe.

Gardez à l'esprit que sur le PICAXE réel, la boucle tourne des milliers de fois par seconde.

2.4.3 Entrées analogique

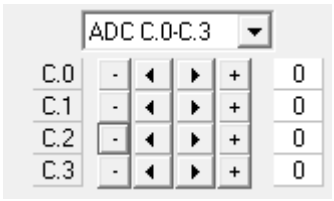
2.4.3.1 Le programme



donne


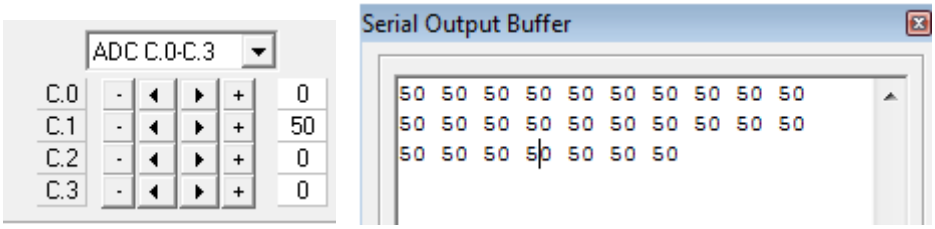
Là encore, il faut un moyen de simuler la variation de tension sur une broche d'entrée.

Il faut aller dans la partie inférieure de la fenêtre de simulation :



Puis que nous voulons lire la broche C.1, il faut choisir ADC C.0-C.3 dans la liste déroulante.

Cliquez sur le + en face de C.1 Vous constatez que la valeur associée est passée à 50.



2.4.3.2 Que s'est-il passé ?

ADC est l'acronyme de Analogic => Digital Converter

C'est à dire "Convertisseur analogique vers digital"

Digital vient du mot doigt : vous savez compter sur vos doigts n'est-ce pas ?

Un ADC permet de convertir une tension en un nombre. Un DAC fait exactement l'inverse.

Dans le cas du PICAXE la tension d'alimentation (5V) est divisée en 256 valeurs entre 0 et 255.

Donc pour 0V l'ADC donnera 0

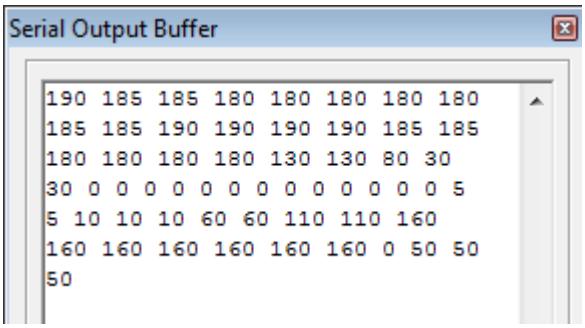
A partir de 20 mV, l'ADC donnera un 1

A partir de 39 mV, l'ADC donnera un 2

etc ... jusqu'à un peu moins de 5V qui donnera 255.

La commande **readADC C.1 b1** lit le résultat du convertisseur ADC connecté à la proche C.1 et place le résultat dans la variable b1.

Vous pouvez jouer avec les boutons - < > + et observer les résultats du convertisseur.



2.5 Gestion des ports

Sur les PICAXES modernes, et même si par défaut le comportement est le même que sur les modèles plus anciens (port B pour les sorties, port C pour les entrées), il est en fait possible de faire ce que l'on veut sur chaque patte.

Il y a donc des notations spéciales pour dire comment on souhaite utiliser la bête :

Chantier en cours...

Manuel : section 2 page 15 :

pinsB	- the portB input pins
outpinsB	- the portB output pins
dirsB	- the portB data direction register
pinsC	- the portC input pins
outpinsC	- the portC output pins
dirsC	- the portC data direction register

When used on the left of an assignment 'pins' applies to the 'output' pins e.g.

```
let outpinsB = %11000000
```

will switch outputs 7,6 high and the others low.

2.6 Tests

2.6.1 Test simple

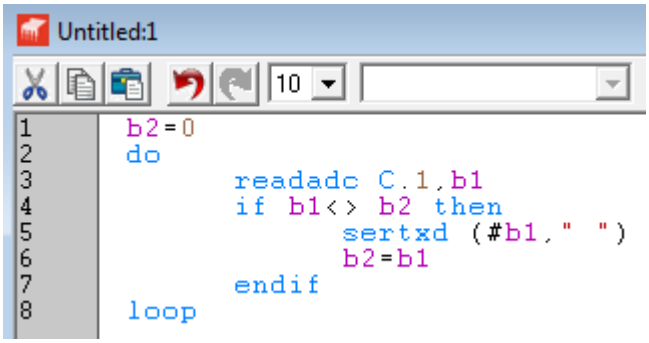
Vous pouvez être agacé de voir de nouvelles valeurs apparaitre sans arrêt dans le "Serial Output Buffer".

Bien sur, il y a le bouton  dans le simulateur qui permet de le stopper temporairement. Mais on peut faire plus riche...

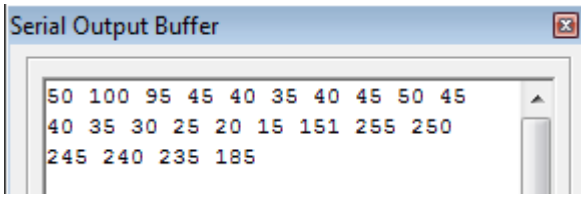
Supposons donc que nous ne souhaitions afficher une valeur que dans le cas où la tension change.

Nous avons besoin d'une nouvelle variable pour conserver la dernière valeur lue. Et nous avons besoin de comparer cette valeur à la valeur courante produite par l'ADC.

2.6.1.1 Le programme



donne (en jouant avec + -) :



2.6.1.2 Comment fonctionne ce programme ?

L'objectif est ici de présenter le bloc **If THEN ENDIF** que l'on peut traduire par

SI ceci **ALORS** cela **FIN DU SI**

ceci est la condition qui doit être **VRAIE** pour que le code **cela** soit exécuté.

Si **ceci** est **FAUX**, alors le programme passe directement à ce qui se trouve après le **ENDIF.**

Mais comment doit-on écrire cette condition ?

Nous verrons plus loin qu'il y a des tas de possibilités, mais retenez déjà les plus courantes :

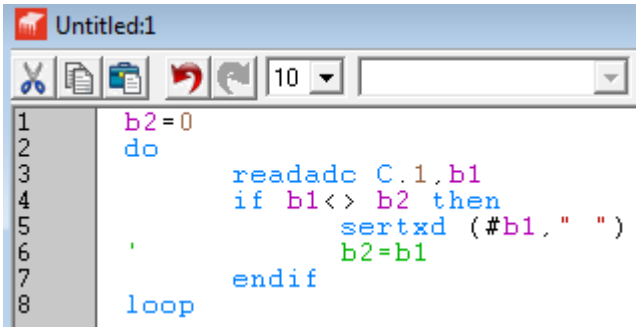
if b1<5	signifie	Si la variable b1 contient une valeur inférieure strictement à 5
if b1<=5	signifie	Si la variable b1 contient une valeur inférieure ou égale à 5
if b1>5	signifie	Si la variable b1 contient une valeur supérieure strictement à 5
if b1>=5	signifie	Si la variable b1 contient une valeur supérieure ou égale à 5
if b1<>5	signifie	Si la variable b1 contient une valeur différente de 5
if b1=5	signifie	Si la variable b1 contient une valeur égale à 5

Bien sur, vous pouvez remplacer la constante 5 par une variable de votre choix comme dans l'exemple ci-dessus.

2.6.1.3 Traquons la petite bête

Que se serait-il passé si on avait oublié le **LET** b2=b1 ?

Pour le savoir, je vous propose de rajouter une petite apostrophe en début de ligne :



Vous constatez que le texte passe en vert, ce qui signifie qu'il est ignoré par le PICAXE.

C'est exactement comme si il n'y avait rien d'écrit.

C'est bien pratique comme ici pour "mettre de coté" une ligne sans avoir à la retaper (il suffit d'enlever le ' et la ligne redevient bleue)

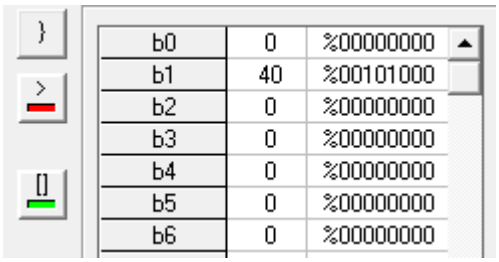
C'est aussi de cette façon que vous commenterez votre code pour vous souvenir plus tard de ce à quoi sert chaque partie...

Mais revenons à notre programme : il ne marche plus très bien !

Nous avons de nouveau un affichage de valeurs successives, sauf si nous ramenons C.1 à 0V.

Il ya un problème dans ce test entre b1 et b2 il me semble. Mais comment voir à chaque étape ce qui se passe dans ces variables ?

Il y a un outil très commode pour ça caché dans le bouton >> en bas du simulateur :

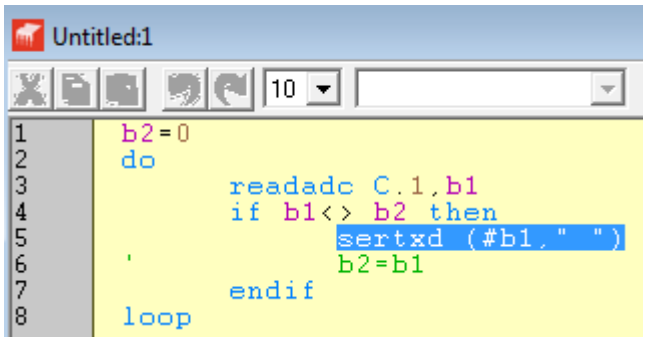


Vous retrouvez les variables b1 et b2 et leurs valeurs courantes.

Vous pouvez maintenant bloquer le programme avec le bouton "pause" : 

puis avancer pas à pas avec le bouton : 

Observez en même temps quelle instruction le PICAXE est en train d'exécuter dans le texte de votre programme (surligné en bleu) :



Après avoir observé l'évolution de b1 en fonction des sollicitations de C.1, stoppez la simulation et retirez le ' pour que la ligne b2=b1 soit de nouveau active (bleue donc).

Observez que cette fois b2 conserve bien la dernière valeur lue et que le programme ne passe dans le bloc que si la tension a changé !

CQFD !

Ce que vous venez de faire, ça s'appelle du "debugging" en anglais, "débogage" ou même "déverminage" en français. Ca consiste à traquer la petite bête, le **bug** en anglais...

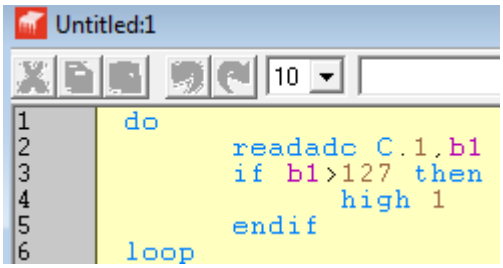
C'est important de bien maîtriser ce type de techniques : même en ayant beaucoup réfléchi avant d'écrire un programme, il arrive souvent que le résultat ne soit pas exactement celui souhaité...

2.6.2 Le YIN et le YANG

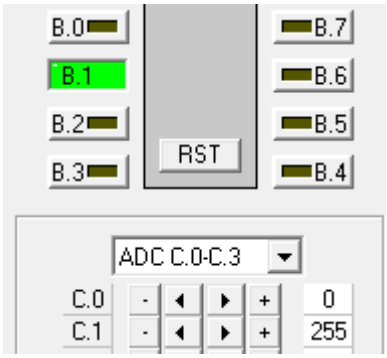
Supposons maintenant que nous souhaitions allumer une LED quand la tension présente sur la broche C.1 est supérieure à 2,5 V soit 127 en sortie de l'ADC.

2.6.2.1.1 Le programme

Fastoche :



donne



Ca marche : B1.1 s'est éclairé en vert dès que C.1 est passé au dessus de 127 !

2.6.2.2 Testez à fond !

En êtes vous bien sur ?

Que se passe-t-il si la tension redescend en dessous de 2,5 V ?

RIEN DU TOUT et c'est bien là le problème ! (la LED reste éclairée)

Pourtant le PICAXE a fait exactement ce qu'on lui a demandé :

éclairer la LED B.1 dès que la tension sur C.1 passe au dessus de 2,5V

Sauf que (implicitement) nous souhaitions (évidement : qu'il est bête alors...) que la LED s'éteigne quand la tension repasse en dessous de 2,5V (ça va sans dire...)

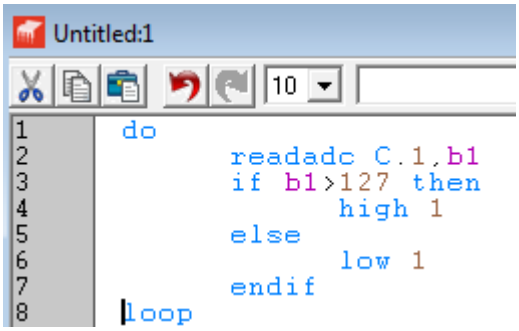
Sauf que pour une fois le PICAXE est totalement incapable de deviner ce cahier des charges "implicite".

Il faut tout lui dire :

éclairer la LED B.1 dès que la tension sur C.1 passe au dessus de 2,5V

mais penser à l'éteindre quand la tension sur C.1 repasse en dessous de 2,5V

Voici le code correspondant :



Je vous engage à vérifier que ça fonctionne bien mieux comme cela, toujours en jouant avec le + et le - sur la broche C.1

Comme ça clignote très fort du coté du code, vous pouvez aussi passer en pause et en mode pas à pas pour observer l'évolution de la variable b1 et ses conséquences dans les blocs de code qui sont exécutés.

Au passage, voici donc une variante du bloc **SI** ceci **FAIRE** cela **FIN DU SI**

C'est le double-bloc **SI** ceci **FAIRE** cela **SINON** ... **FINDUS**

qui se traduit en anglais par **If** ceci **THEN** cela **ELSE** ... **ENDIF**

Remarque N°1 :

Ca me change de traduire du français vers l'anglais pour une fois !

Remarque N°2 :

Si un vendeur de poissons surgelés souhaite sponsoriser le club des Picaxiens, il est le bienvenu.

Remarque N°3 :

Tous allusion à mon film culte "Demolition Man" est voulue...

Toutes plaisanteries mises à part, retenez une chose :

Pour tester un programme, il faut lui faire subir toutes les conditions possibles, que ça soit réaliste ou pas, et dans tous les sens de variation possibles.

Certains ont perdu une fusée Ariane à cause d'un test incomplet dans un programme sur des nombres entiers...

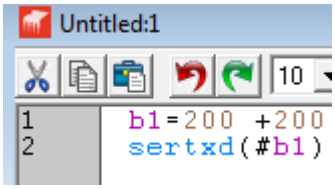
2.7 Arithmétique sur les nombres entiers

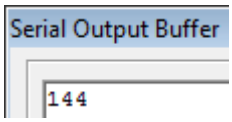
Et bien puisque nous parlons des nombres entiers, allons-y à fond.

Et je vous prévient : sur un PICAXE, c'est pas triste...

2.7.1 Addition

2.7.1.1 Le programme



qui donne 

DIANTRE !

2.7.1.2 Que s'est-il passé ?

Souvenez-vous (sisi, je l'ai déjà dit...) :

Une variable est un emplacement de la mémoire du PICAXE qui peut contenir un nombre entier compris (inclusivement) **entre 0 et 255**.

Que fait le PICAXE quand le résultat d'un calcul dépasse 255 ?

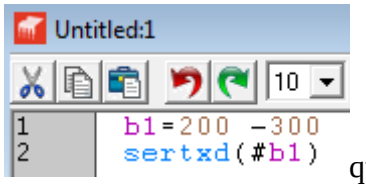
Et bien il soustrait 256 !

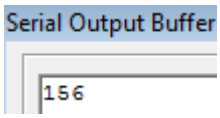
200 + 200 - 256 = 144

Vous avez d'autres questions ?

2.7.2 Soustraction

2.7.2.1 Le programme



qui donne 

Logique ! (vu ce qui précède...)

2.7.2.2 Que s'est-il passé ?

Vous avez trouvé ? bravo : passez au paragraphe suivant (vous ne passez pas par la case "départ", vous ne recevez rien...)

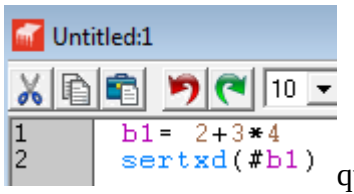
Il faut réexpliquer ? No problem :

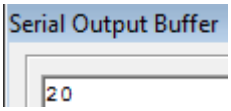
Le résultat de l'opération est un nombre inférieur à zéro : donc le PICAXE a rajouté 256 !

200 - 300 +256 = 156

2.7.3 Multiplication

2.7.3.1 Le programme



qui donne 

Caramba ! Encore raté !

2.7.3.2 Que s'est-il passé ?

Tout bon BASIC standard aurait donné 14 et c'est sans doute ce à quoi vous vous attendiez.

Sauf que sur ce point le PICAXE n'est pas standard :

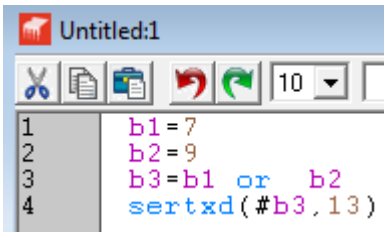
- en général on applique la "préséance" des opérateurs, c'est à dire que l'on fait d'abord les multiplications et les divisions, puis les additions et les soustractions.
- le PICAXE lui, fait les opérations de la gauche vers la droite sans aucune préséance.

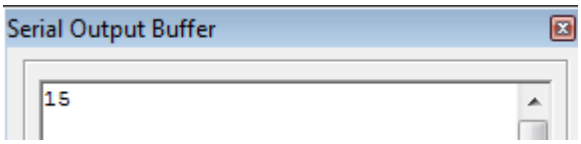
Ce n'est ni mieux, ni moins bien, mais il faut en tenir compte. Donc notre PICAXE a effectué :

(2+3) * 5 ce qui fait bien 20

2.7.4 Opérations sur les bits

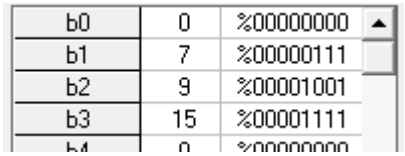
2.7.4.1 Le programme



qui donne 

Mais cé quoi ce truc ?

Pour avoir l'explication, jetez un œil dans :



b0	0	%00000000
b1	7	%00000111
b2	9	%00001001
b3	15	%00001111

b3 est le **OU** binaire de b1 et de b2

Si vous regardez les chiffes 0 ou 1 colonne par colonne, vous constatez que les chiffres de b3 sont a 1 quand le chiffre correspondant de B1 **OU** de b0 est à 1;

Autrement dit, le chiffre dans B3 n'est à 0 que si le chiffre correspondant dans B1 **ET** B2 sont à 0.

On peut établir la table de vérité de l'opérateur OR :

b1	0	0	1	1
b2	0	1	0	1
b3=b1 OR b2	0	1	1	1

Je vous laisse essayer les autres opérateurs (et au besoin établir les tables de vérité ad hoc...)

AND OR XOR NAND NOR XNOR ANDNOT ORNOT

2.7.5 Pour aller plus loin

Il y a une autre limitation : les calculs intermédiaires ne peuvent dépasser 65535 ni être négatifs.

Donc il faut faire bien attention à chaque opération pour "rester dans les clous", mais si le résultat final est entre 0 et 255.

Impossible toutefois de faire une démo : le simulateur ne simule pas ce problème !

Il s'appuie bêtement sur le processeur de votre ordinateur (genre Pentium...) qui lui gère parfaitement bien toutes ces broutilles... Si bien que les calculs qui donneraient des résultats "fantaisistes" sur un PICAXE sortent "impec" sur le simulateur.

2.8 Choix multiples

Supposons que nous souhaitions construire un générateur de fonctions, capable d'afficher 5 types d'ondes :

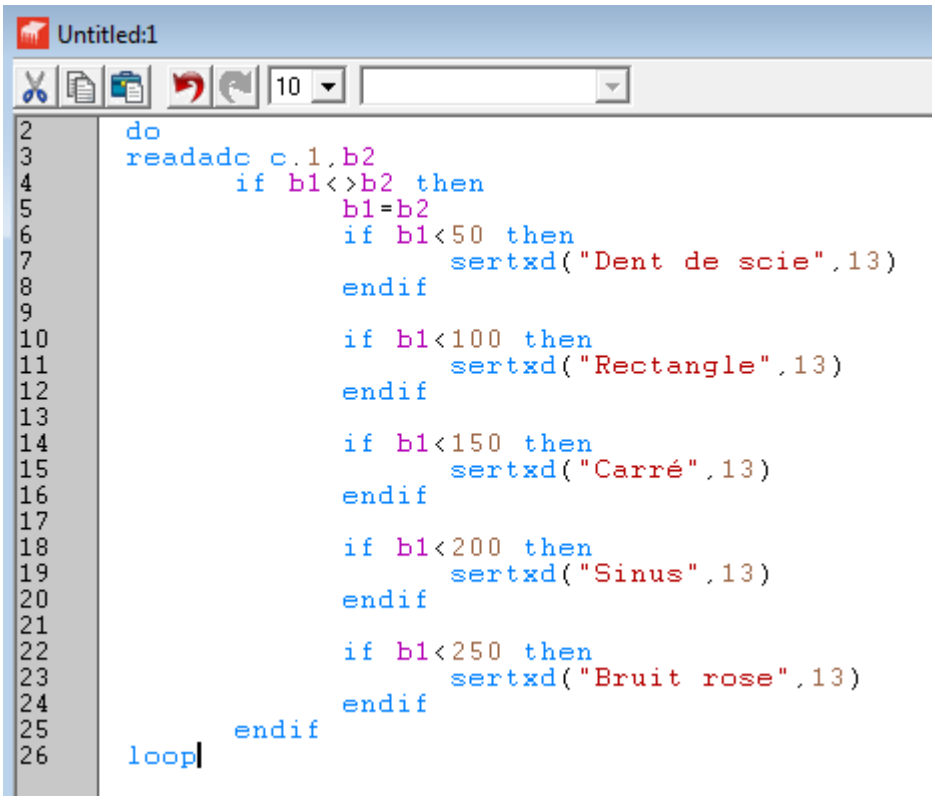
- Dent de scie
- Rectangle
- Carré
- Sinus
- Bruit rose

Le cahier des charges précise également que la sélection sera réalisée par un potentiomètre.

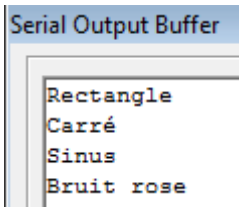
2.8.1 Le programme

Classiquement, nous allons lire la tension obtenue en divisant les 5V de l'alimentation à l'aide du potentiomètre. Les valeurs produites en sortie de l'ADC vont donc varier de 0 à 255.

Sur cette base on peut envisager un programme du style :



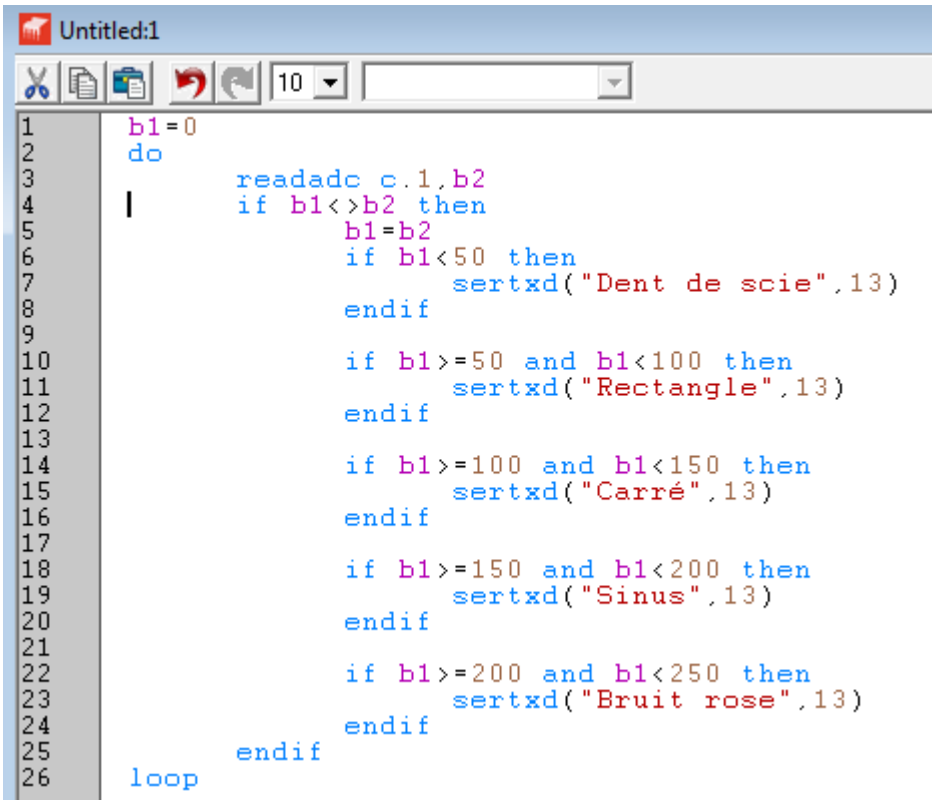
En passant C.1 à la valeur 50, on obtient :



En appliquant les conseils de déverminage, on identifie le problème :

- si b1 est inférieur à 100, il est aussi inférieur à 150, 200 et 250 !

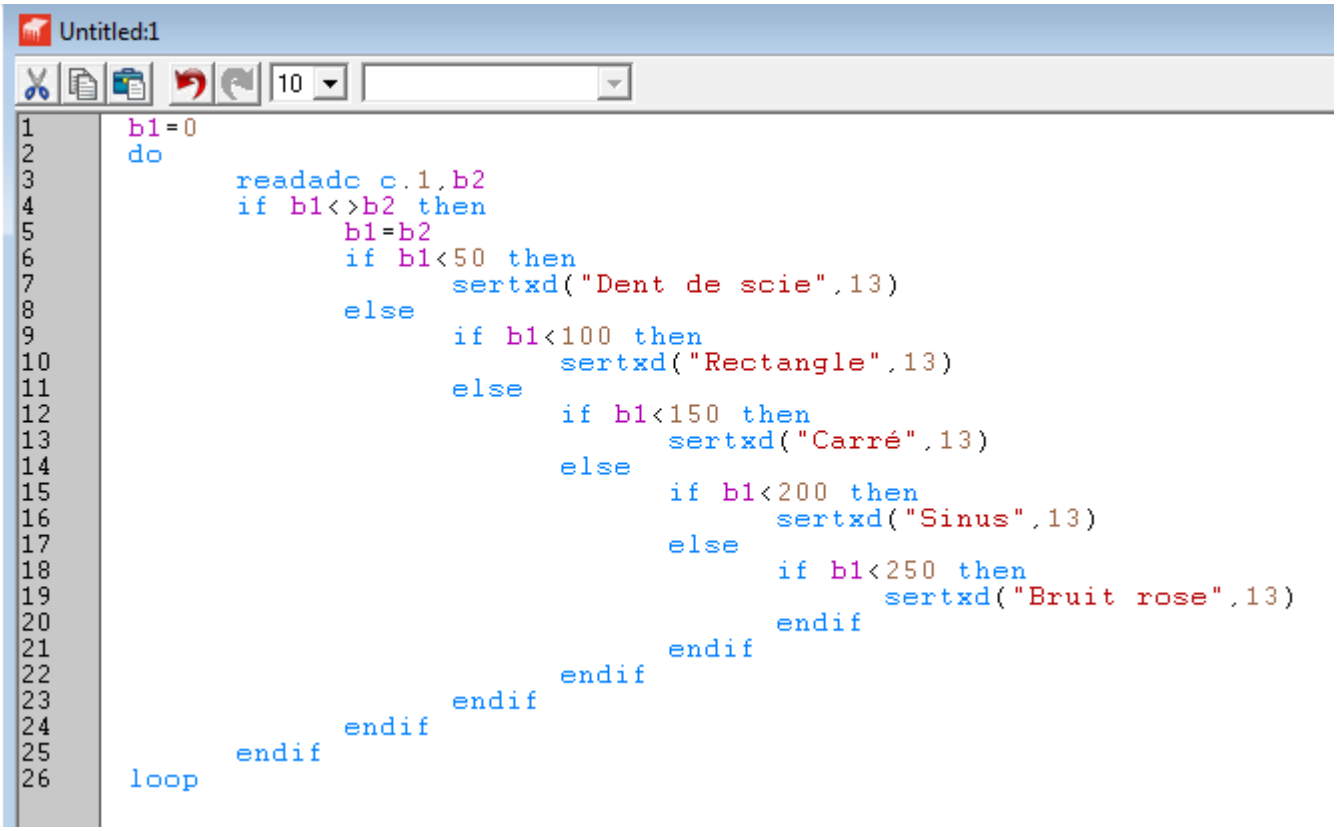
Une première solution consiste à faire des tests plus complets :




```
b1=0
do
      readadc c.1,b2
      if b1<>b2 then
            b1=b2
            if b1<50 then
                  sertxd("Dent de scie",13)
            endif

            if b1>=50 and b1<100 then
                  sertxd("Rectangle",13)
            endif

            if b1>=100 and b1<150 then
                  sertxd("Carré",13)
            endif

            if b1>=150 and b1<200 then
                  sertxd("Sinus",13)
            endif

            if b1>=200 and b1<250 then
                  sertxd("Bruit rose",13)
            endif
      endif
loop
```

Ca marche.

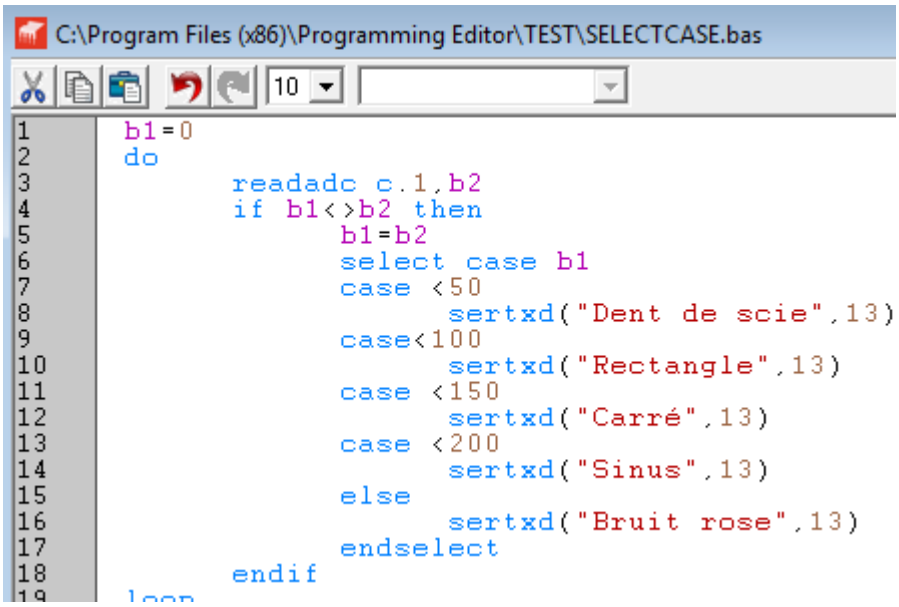
Autre solution plus phosphorée :




```
b1=0
do
      readadc c.1,b2
      if b1<>b2 then
            b1=b2
            if b1<50 then
                  sertxd("Dent de scie",13)
            else
                  if b1<100 then
                        sertxd("Rectangle",13)
                  else
                        if b1<150 then
                              sertxd("Carré",13)
                        else
                              if b1<200 then
                                    sertxd("Sinus",13)
                              else
                                    if b1<250 then
                                          sertxd("Bruit rose",13)
                                    endif
                              endif
                        endif
                  endif
            endif
      endif
loop
```

Ca marche aussi.

Le PICAXE dispose d'une structure spécialement dédiée à ce type de cas de figure :




```
1   b1=0
2   do
3         readadc c.1,b2
4         if b1<>b2 then
5               b1=b2
6               select case b1
7               case <50
8                     sertxd("Dent de scie",13)
9               case<100
10                    sertxd("Rectangle",13)
11              case <150
12                    sertxd("Carré",13)
13              case <200
14                    sertxd("Sinus",13)
15              else
16                    sertxd("Bruit rose",13)
17              endselect
18        endif
19  loop
```

Avouez que c'est quand même plus simple et que ça marche tout aussi bien.

2.8.2 Les explications

Il s'agit d'une structure à N blocs que l'on peut traduire par :

CHOISIS SUIVANT b1

SI <50

faire ceci

SI <100

faire cela

SI <150

...

SI <200

...

SINON

Faire cela dans les autres cas

FIN DE LA LISTE DE CHOIX

Remarque

On aurait pu écrire **case <250** à la place du **ELSE**. C'est vrai.

Mais, suite aux remarques sur la "solidité" des programmes, prenez l'habitude de toujours traiter le cas **ELSE**.

Avec un **case <250** que se passerait-il pour les valeurs comprises entre 250 et 255 ?

2.9 Sous programmes

Comme expliqué plus haut, il est fréquent que la boucle principale doive faire différentes choses.

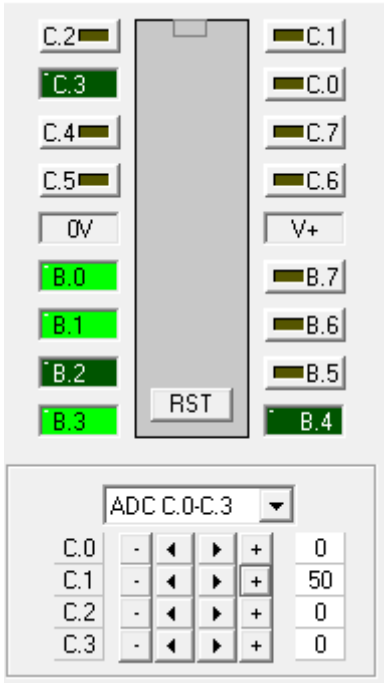
Supposons par exemple que notre générateur de fonction doivent en plus faire clignoter une LED correspondant au type de signal choisi :

B.0 pour le signal en dent de scie, **B.1** pour le Rectangle, etc...

2.9.1 Le programme

C:\Program Files (x86)\Programming Editor\TEST\SELECTCASE.bas

```
1   b1=0
2   do
3         toggle b3
4
5         readadc c.1,b2
6         if b1<>b2 then
7               b1=b2
8               select case b1
9               case <50
10                    sertxd("Dent de scie",13)
11                    b3=0
12              case<100
13                    sertxd("Rectangle",13)
14                    b3=1
15              case <150
16                    sertxd("Carré",13)
17                    b3=2
18              case <200
19                    sertxd("Sinus",13)
20                    b3=3
21              else
22                    sertxd("Bruit rose",13)
23                    b3=4
24              endselect
25        endif
26  loop
```



Ca marche à peut prêt ... sauf que les leds ont tendance à ne pas s'éteindre...

Il faudrait penser à éteindre la lumière avant de sortir. (oui :j'ai une fibre écolo...)

C:\Program Files (x86)\Programming Editor\TEST\SELECTCASE.bas

```
do
      toggle b3

      readadc c.1,b2
      if b1<>b2 then
            b1=b2
            select case b1
            case <50
                  sertxd("Dent de scie",13)
                  low b3
                  b3=0
            case<100
                  sertxd("Rectangle",13)
                  low b3
                  b3=1
            case <150
                  sertxd("Carré",13)
                  low b3
                  b3=2
            case <200
                  sertxd("Sinus",13)
                  low b3
                  b3=3
            else
                  sertxd("Bruit rose",13)
                  low b3
                  b3=4
            endselect
      endif
loop
```

Vous constatez que maintenant une seule led clignote.

2.9.2 Simplifions

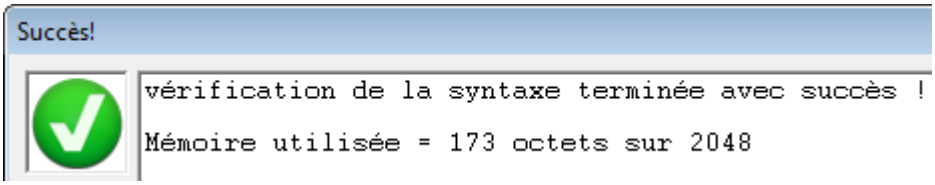
Petit à petit, notre programme prends de la consistance.

Imaginez ce qui se passe quand vous avez vingt choix possibles, 10 boutons à traiter, des tas de leds à faire clignoter, etc...

Vous pouvez bien sur mettre tout bout à bout et ça va marcher.

Mais ça devient pénible à lire, et accessoirement la taille du programme n'est pas illimitée (ça dépends du Picaxe choisi).

Pour avoir l'info, cliquez sur le bouton 



Ah quand même ! Avec ce bête exemple, on a déjà utilisé près de 10% de la mémoire.

Remarquons que dans chaque cas de figure, on fait exactement la même chose :

- envoyer un CR (retour à la ligne)
- mettre à 0V la broche dont le N° est contenu dans la variable b3

Dans les cas réels, il y a souvent des multitudes de choses à faire systématiquement.

L'idée serait donc de confier ce travail répétitif à un sous-programme avec les avantages suivants:

- c'est plus concis et plus lisible,
- l'encombrement mémoire diminue,
- en cas de modification, on ne doit changer le code qu'à un seul endroit. L'expérience prouve que si on doit changer le code dans chaque **case**, on se trompe au moins une fois et ce n'est pas facile à déverminer parce que l'on est persuadé d'avoir fait 5 fois exactement la même chose...

Mais trêve de paroles (écrites) : passons aux actes :

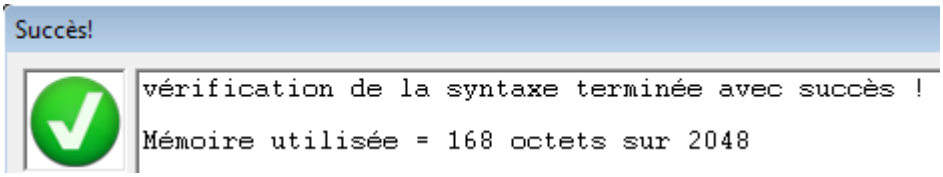
C:\Program Files (x86)\Programming Editor\TEST\SELECTCASE2.bas

```
1   b1=0
2   do
3         toggle b3
4
5         readadc c.1,b2
6         if b1<>b2 then
7               b1=b2
8               select case b1
9               case <50
10                    sertxd("Dent de scie")
11                    gosub ValiderChoix
12              case<100
13                    sertxd("Rectangle")
14                    gosub ValiderChoix
15              case <150
16                    sertxd("Carré")
17                    gosub ValiderChoix
18              case <200
19                    sertxd("Sinus")
20                    gosub ValiderChoix
21              else
22                    sertxd("Bruit rose")
23                    gosub ValiderChoix
24              endselect
25        endif
26  loop
27
28
29
30  ValiderChoix:
31        low b3
32        b3=b2/50 max 4
33        sertxd (13)
34  return
```

Je vous laisse vérifier que ça marche toujours (ça nous change des exemples sur l'arithmétique...)

2.9.3 Explications

Question gain mémoire, ce n'est pas terrible :



On va néanmoins dans le bon sens. En effet, notre exemple est trop simple, et il y a trop peu de choses répétitives pour compenser les lignes chargées d'appeler le sous-programme. Dans un cas réel, le gain est bien plus important.

Ah mais oui au fait, j'ai oublié de vous présenter ce cher **GOSUB**, et son inséparable compagnon, j'ai nommé monsieur **RETURN**

Leurs collègues français se nomment **VASY** et **REVIENTVITE**

Vous remarquez aussi le mot "ValiderChoix". Ca s'appelle un LABEL, comprenez : une étiquette.

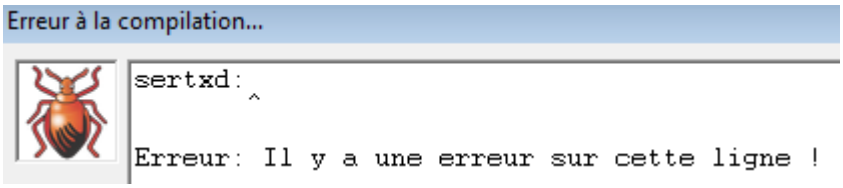
Si vous demandez à quelqu'un d'aller quelque part, c'est commode que le quelque part porte une pancarte avec son nom dessus.

Evidement, il est hautement conseillé de donner un nom qui explique bien ce que fait le sous-programme.

Mais vous pouvez donner n'importe quel nom à un LABEL (sauf ceux réservés par le langage BASIC) :

- ValiderChoix
- AllumerLaLED
- RepeindreLePlafond
- DeCadix
- EtLaBete

- ~~SerTxd~~ ' non non non : ça c'est interdit !



Vous avez sans doute compris que lorsque le PICAXE exécute la commande **GOSUB**, il se déroute vers le LABEL "ValiderChoix" et fait le travail demandé.

Arrivé au **RETURN**, il revient tout bonnement à l'endroit d'où il était parti (à l'instruction juste après en pratique), c'est à dire dans notre cas au prochain **case**

2.9.4 Allons plus loin

On peut également utiliser un **GOSUB** uniquement pour organiser le programme :

C:\Program Files (x86)\Programming Editor\TEST\SELECTCASE2.bas

```
1   b1=0
2   do
3         toggle b3
4
5         readadc c.1,b2
6         if b1<>b2 then
7               b1=b2
8               gosub TraiterChoix
9         endif
10  loop
11
12
13  TraiterChoix:
14        select case b1
15        case <50
16              sertxd("Dent de scie")
17              gosub ValiderChoix
18        case<100
19              sertxd("Rectangle")
20              gosub ValiderChoix
21        case <150
22              sertxd("Carré")
23              gosub ValiderChoix
24        case <200
25              sertxd("Sinus")
26              gosub ValiderChoix
27        else
28              sertxd("Bruit rose")
29              gosub ValiderChoix
30        endselect
31  return
32
33
34  ValiderChoix:
35        low b3
36        b3=b2/50 max 4
37        sertxd (13)
38  return
```

Vous constatez que ça marche toujours aussi bien.

Mais la boucle principale est maintenant toute simple. A la lecture du programme, on comprends facilement ce qui se passe "en boucle" et ce qui est déroulé uniquement en cas d'action sur le potentiomètre.

2.10 Le renégat...

Prononcez le mot **GOTO** devant un programmeur expérimenté, de la catégorie des purs et durs : vous constaterez immédiatement que son visage s'empourpre, que ses oreilles fulminent et que ses cheveux se hérissent. Il finira même probablement par éructer : "GOTO = CACA" (oui ces gens là ont souvent peu de vocabulaire en dehors de i:=0;i++; ^a > ^^b)

Pourtant le **GOTO** ressemble très fort au **GOSUB** (qui jouis d'un réputation acceptable).

C'est un **VAS LA BAS** qui n'aurait pas de compagnon genre **REVIENTVITE**

Donc une fois parti, on n'est jamais sur de revenir au programme principal. Il faut penser à mettre un autre **GOTO** vers un **LABEL** du programme principal, et ceci dans tous les cas de figure...

De plus, ce retour peut être choisi n'importe où dans le programme principal, au milieu d'un **FOR NEXT**, dans un **IF THEN ELSE**,...

Un programme écrit comme cela peut fort bien marcher, ou même "tomber en marche", mais son comportement est en général assez imprévisible.

Le déverminage est aussi spécialement corsé.

Vous voulez des exemples ?

Ca va pas non : pas envie de me faire lyncher moi !

Essayez si vous voulez : je n'ai personnellement aucun problème avec le **GOTO**.

2.11 Variables : explication (plus) complète

Ah oui : je vous l'avais promis : il faut que je vous parle des autres types de variables du PICAXE.

2.11.1 La théorie :

Pour cela référons nous au cultissime Tome 2 du Manuel anglais :

General Purpose Variables.

	Bytes	**Bit Name**	**Byte Name**	**Word Name**
X2 parts	56	bit0-31	b0-55	w0-27
X1 parts	28	bit0-31	b0-27	w0-13
M2 parts	28	bit0-31	b0-27	w0-13
Older	14	bit0-15	b0-13	w0-6

Accrochez vous bien au pinceau : j'enlève l'échelle...

General Purpose Variables = variables d'usage général

Donc il y a aussi des variables d'usage spécifique... Nous verrons plus tard.

La première colonne n'a aucun nom : c'est la famille de PICAXE considéré, présentés depuis les plus performants jusqu'aux plus anciens = older

Byte : traduisez ça par **mot** (au sens informatique du terme)

Ne confondez pas avec **bit** (que l'on ne traduit jamais, j'y peux rien).

Le **bit** est la plus petite information qu'un microprocesseur sache gérer. En clair **0** ou **1**

Un **bit** n'a que deux état possibles : 0V ou 5V, 0 ou 1, VRAI ou FAUX, ECLAIRE ou ETEINT,...

Un **mot** (informatique) est un paquet de **bits** que le microprocesseur est capable de traiter en une seule fois.

Pour les microprocesseurs 8 bits dont font partie les PICs et donc les PICAXES, le mot fait 8 bits.

Il y a 256 possibilités que le PICAXE utilise pour représenter les nombres de 0 à 255.

Mais dans d'autres cas, ça peut aussi bien représenter les nombres de -128 à +127

Ou les lettres du code ASCII comme nous l'avons vu plus haut.

On utilise aussi l'**octet** qui contrairement au mot, fait toujours 8 **bits**. Sur un PICAXE, un **Byte** ou un **octet**, c'est la même chose. Mais ce n'est pas vrai sur tous les microcontrôleurs.

Donc la colonne bytes indique le nombre de mots de 8 bits que le PICAXE considéré peut mémoriser.

Pour les plus anciens modèle (older) pas plus de 14, mais 56 pour un X2.

Je passe directement à la colonne Byte Name : c'est tout simplement le nom de la variable qui désigne la variable (définie par défaut) correspondant à une mémoire.

b0-b13 désigne les variables b0, b1, b2, b3, b4,..., b13 que nous avons déjà utilisé.

Passons maintenant à la dernière colonne : **Word Name**

Elle contient les noms des variables "larges" (wide en anglais)

Par exemple, pour un X2 : w0, w1, w2, w3,...,w27

Une variable "large" est composée de 2 mots de 8 bits ce qui nous fait dans les 16 bits donc.

Il y a 2^{16} possibilités que le PICAXE utilise pour représenter les nombres de 0 à 65535.

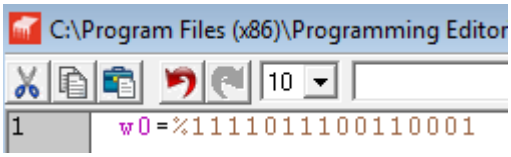
Kolossale subtilité :

Les variables **w...** désignent les mêmes emplacement mémoire que les variables **b...** de même rang.

Par exemple, la variable w0 désigne la même mémoire que les variables b0 et b1.

2.11.2 La pratique

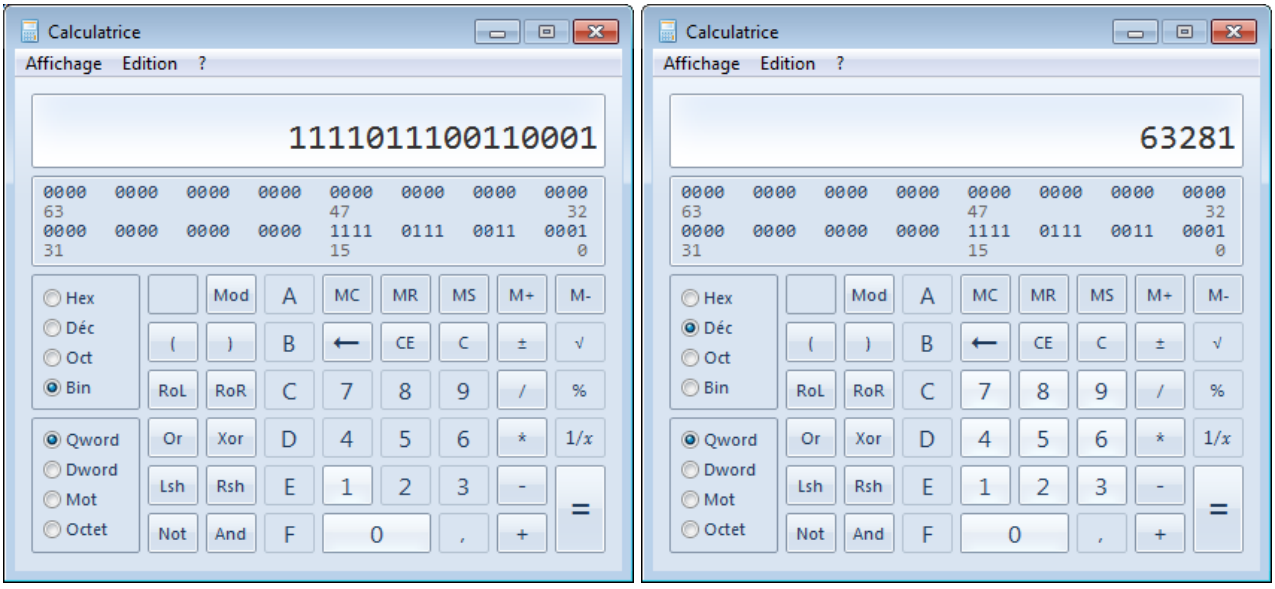
Pour y voir plus clair, je vous suggère d'écrire ce fabuleux programme :



puis d'ouvrir le simulateur et la case >>

La notation %1111011100110001 sert à introduire un nombre binaire.

Le menu Affichage/calculator donne accès à la calculette de Windows qui permet fort commodément de traduire en binaire ou en hexadécimal n'importe quel nombre décimal (et réciproquement)



Revenons à notre simulateur :

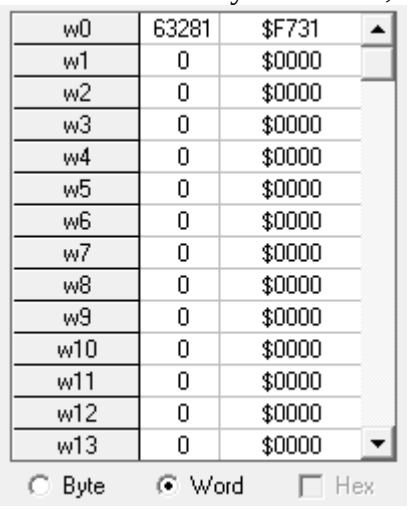
b0	49	%00110001
b1	247	%11110111
b2	0	%00000000
b3	0	%00000000

Vous constatez que notre programme a bel et bien modifié les variables b0 et b1

Plus précisément, b0 a reçu la partie droite du nombre binaire (on parle des bits les moins significatifs).

b1 à reçu la partie gauche : ce sont les bits les plus significatifs.

En jonglant avec les boutons Byte et Word, vous pouvez passer de l'affichage des variables **b...** à celui des variables **w...**



Constatez que la calculette Windows ne vous a pas raconté des blagues...

2.11.3 Les champs de bits sont-ils constitués de Terrabits ?

Ma foi : il ne reste plus que la colonne Bit Name

Ce sont les noms des variables qui ne peuvent contenir que des bits.

Il n'y a que deux possibilités : 0 ou 1

Et là encore, le PICAXE utilise les mêmes emplacements mémoire pour stocker ces variables.

La variable b0, c'est donc exactement la même chose que les variables

bit7, bit6, bit5, bit4, bit3, bit2, bit1, bit0

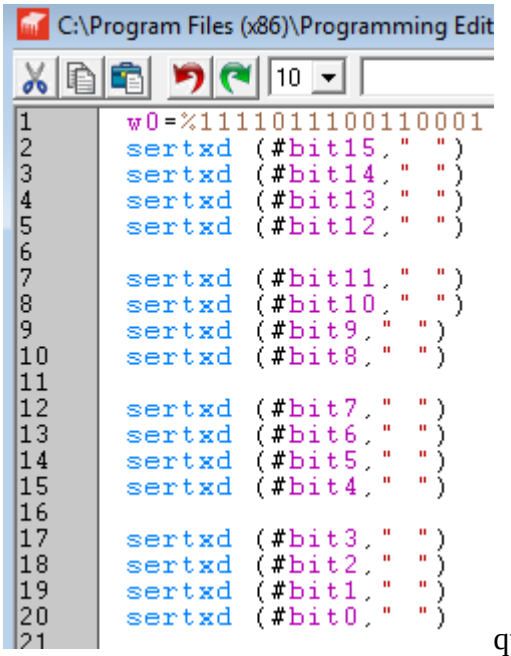
b1, c'est exactement la même chose que

bit15, bit14, bit13, bit12, bit11, bit10, bit9, bit8

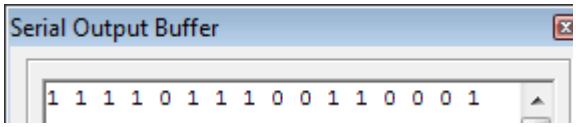
et w0, c'est b1, b0 ou bien

bit15, bit14, bit13, bit12, bit11, bit10, bit9, bit8, bit7, bit6, bit5, bit4, bit3, bit2, bit1, bit0

Pour vous en convaincre, vous pouvez écrire (fastidieusement, je vous l'accorde) le programme suivant :



qui donne



2.11.4 Soyez méthodiques !

Toutes ces variables qui partagent les mêmes emplacement mémoires peuvent être bien commodes.

Par exemple si vous avez besoin des bits individuels d'un nombre (en gros de convertir un nombre en binaire), il suffit de mettre le nombre dans b0, et pouf, instantanément les bits sont disponibles dans les variables b0 à b7.

2.11.4.1 Le problème

Mais vous risquez aussi un terrible "effet de bord".

On désigne de cette façon les conséquences d'un modification d'un programme dans une partie éloignée, à priori sans aucun lien avec l'endroit modifié.

Par exemple, vous avez une partie qui lit l'ADC sur 8 bits et une autre bien séparée qui transmet des données sur un port série.

Vous décidez de passer l'ADC à 10 bits, et donc d'utiliser une variable **w...** pour enregistrer une valeur supérieure à 255 (oui 10 bits, ça va de 0 à 1023)

Et là, c'est le drame : le module qui envoie les données se met à merdouiller joyeusement...

En fait, le module qui envoie les données utilise la variable b7.

Et pour lire l'ADC, vous utilisiez b3, et tout marchait très bien.

Donc en passant à 10 bits, vous prenez à la place de b3 ➔ w3

GROSSE ERREUR ! w3, en fait, c'est b6 et **b7**

Pour peu que les deux sous modules fonctionnent en même temps, vous avez l'explication des dysfonctionnements.

2.11.4.2 Une solution

Vous avez remarqué que je n'ai pas utilisé la variable b0 dans les exemples précédents. C'est justement pour éviter un effet de bord si jamais j'ai besoin des bits 0 à 7.

Si vos besoins en variables binaires (on dit aussi booléennes) sont plus importants, vous pouvez aussi bannir les variables b1 à b3.

Au delà, ça ne sert plus à rien car les variables bits ne vont que de 0 à 31.

Dans ce cas, il faut aussi bannir w0 et w1

D'autre part, il est recommandé de choisir les variables **w...** en partant de **w27** pour un X2 (ou de **w13** pour un M2), puis w26, w25, etc...

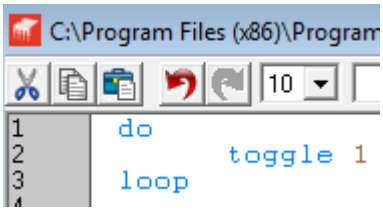
Attention toutefois, si vous êtes "short petrol" il va falloir utiliser des ruses de sioux quand les deux courbes se croiseront, en général vers b30 et w15.

2.12 Pauses

Il faut bien souffler un peu de temps en temps...

2.12.1 Philosophie N°1

Revenons à un programme tout simple :



qui provoque le clignotement d'une LED branché sur B.1, en tout cas sur le simulateur.

Si vous injectez ça sur un PICAXE réel et que vous branchez la LED (via une résistance),

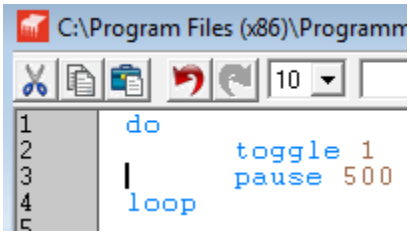
vous allez constater que la LED est éclairée de façon fixe.

En tout cas, c'est ce que qu'il vous semble avec vos yeux. Avec une caméra super rapide, vous pourriez peut être discerner un clignotement super rapide.

Le problème, c'est que le simulateur ne peut faire au mieux que 4 à 5 boucles par seconde, et comme la LED est une fois éteinte, une fois allumée, elle ne s'éclaire pas plus de 2 fois par seconde : vos yeux s'en sortent très bien.

Mais sur un PICAXE réel, ça peut tourner à 1000 fois par seconde. Autant vous dire que vos yeux sont dans les choux.

Donc, c'est tout bête, il suffit d'attendre un peu entre chaque boucle, et ça devrait aller mieux :



et notre LED bat gentiment la seconde !

La commande **pause** sert à arrêter le PICAXE un certain temps, exprimé en millièmes de secondes.

Donc 500, ça fait une demie seconde. Comme expliqué plus haut, il faut tenir compte du fait que la LED est une fois allumée, une fois éteinte ; donc la LED s'allume en gros toutes les secondes.

En gros ? pourquoi donc ?

Parce que le PICAXE a aussi besoin d'un peu de temps pour effectuer le **DO LOOP** et le **TOGGLE**.

Certains se sont "amusés" à mesurer très précisément le temps d'exécution de chaque commande du PICAXE. Il peuvent ainsi ajuster le temps de **pause** pour allumer la LED avec une fréquence un peu plus précise.

Dans tous les cas, il faut diminuer la valeur 500 pour améliorer la précision.

Il existe une variante spécialisée dans les temps très courts de la commande **pause** : c'est la commande **pauseus** suivie d'une durée en micro-secondes.

Vous verrez des tas de programmes criblés de pauses de partout.

Par exemple un PICAXE transmet des données sur un port série. Un autre est censé les recevoir.

Mais ça merdouille gentiment (oui ça peut aussi merder gravement : ça c'est quand on ne reçoit rien du tout).

Non : là ça merdouille juste un peu : de temps en temps, certains caractères sont perdus...

Qu'à cela ne tienne : une bonne petite pause (comme une bonne guerre...) dans le programme émetteur entre chaque caractère envoyé et pouf : miracle, ça ne merdouille plus !

Bon d'accord, ça va deux fois moins vite, MAIS CA NE MERDOUILLE PLUS on vous dit.

Bon OK , je force le trait : c'est pour bien faire comprendre ou je veux en venir.

Quand le programme n'est pas critique (le PICAXE a tout le temps pour faire ce qu'on lui demande) vous pouvez utiliser autant de pauses que vous voulez.

Mais quand vous commencez à rajouter des pauses et que "ça va mieux" sans que vous compreniez vraiment pourquoi, gare...

2.12.2 Philosophie N°2

En réalité, tant qu'il est alimenté, le PICAXE ne sait absolument pas "ne rien faire"

Au minimum, il se "tourne les pouces" (y compris dans une instruction pause) c'est à dire qu'il boucle comme un fou sur une instruction qui n'a aucun effet visible.

Si vous voulez vraiment arrêter le PICAXE, par exemple pour diminuer la consommation de courant parce que votre montage doit fonctionner pendant des jours sur une pile minuscule, utilisez la commande **hibernate**.

Si vous êtes "time critical" (en français : que la réactivité du montage est primordiale)

c'est ballot de laisser le PICAXE "se tourner les pouces" alors que vous avez besoin de lui pour faire autre chose !

Supposons que nous souhaitions éclairer une LED toutes les 2 secondes et une autre toutes les 3 secondes.

Je vous laisse réfléchir.

Oui oui : en même temps

Alors, alors, vous me la vendez à combien la pause ?

to be continued

2.13 Ecrire un beau programme

2.13.1 Pourquoi

2.13.2 Comment