Tutoriel : Apprendre l'ASM Z80 pour TI

Table des matières

Apprendre l'ASM Z80 pour TI

Salut à tous !
Vous en avez marre du TI-Basic ? Ou vous voulez directement profiter de toutes les capacités de votre TI (83+, 83+ SE, 84+, 84+ SE) ?
Eh bien vous êtes sur le bon tutoriel ! C'est parti ! :)

> Ce tutoriel ne traitera pas de l'Assembleur pour TI-83 ; les différences sont peu nombreuses mais leurs conséquences sont importantes. Il est donc déconseillé de suivre ce tutoriel si vous possédez cette calculatrice, mais je ne peux pas vous empêcher de le faire !

Quelques trucs à savoir...

Saviez-vous que le microprocesseur présent dans votre chère calculatrice est le même qui équipait les ordinateurs des années 60 ?!
Et que la rapidité de la calculatrice peut être augmentée (ou diminuée... :'( ) via l'ASM ?
Ça devient intéressant là, non ?
On attaque !

Pourquoi l'ASM ?

Je précise que, étant parfois amené à parler du TI-Basic, je peux aussi bien l'appeler « Basic ». Que cela ne vous surprenne pas, il s'agit ici du même langage.

Pour le comparer au TI-Basic, il est plus rapide, permet d'utiliser pas mal de fonctions système mais nécessite un PC pour son édition (ce qui ne vous rebute pas, j'espère :p) ; il est en outre d'une complexité assez importante (face au TI-Basic bien sûr !).
En fait, lorsque la calculatrice lance un programme Basic, elle doit à chaque instruction se référer à une table qui lui explique comment la réaliser dans son propre langage. En effet, le TI-Basic est un langage interprété, ce qui implique une relative lenteur.
Avec l'ASM, rien de tout cela ! La calculatrice peut directement exécuter les instructions du programme puisqu'elle n'a pas à les traduire ! Il en résulte bien évidemment une rapidité importante et **constante**. Les programmeurs Basic ont en effet peut-être pu constater une « baisse de régime » au cours de l'exécution du programme.
De plus, bon nombre de fonctions sont accessibles à l'ASM et pas au Basic, et vice-versa.

> Mais ? Si la calculatrice peut interpréter les fonctions Basic, on doit pouvoir les utiliser en Assembleur, non ?

En fait, je préfère pour le moment éviter de parler des liens entre Basic et Assembleur car pour débuter, il vaut mieux ne considérer qu'un seul langage sans faire de rapprochement entre les deux.
Tant pis pour les programmeurs en TI-Basic ! :diable:
Mais la diversité des fonctions ne vous fera pas regretter ce choix !

Introduction à la notion de bases

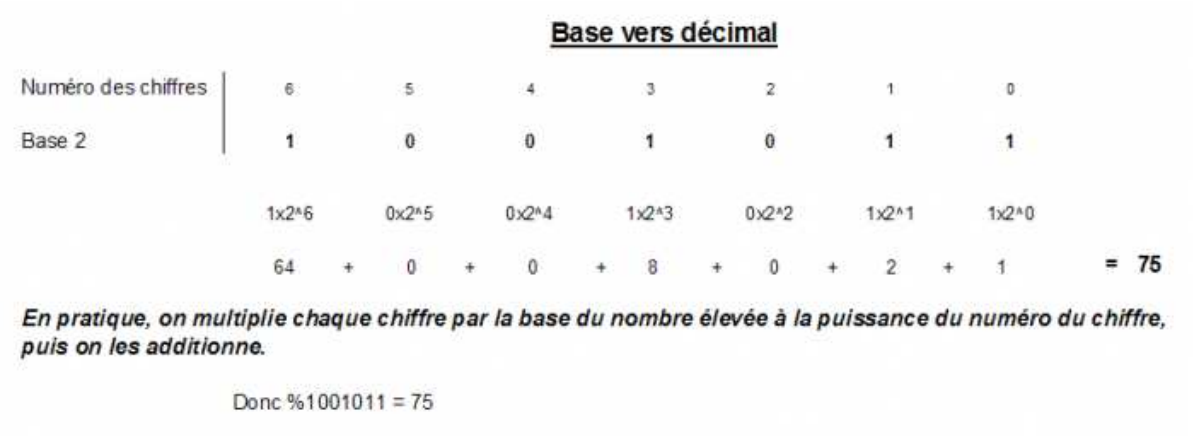
OK, vous devez sûrement vous taper ça à chaque tuto ; mais là je considère que la connaissance des autres bases est plus importante. Et puis vous êtes des Zéros, oui ou non ?! o_O

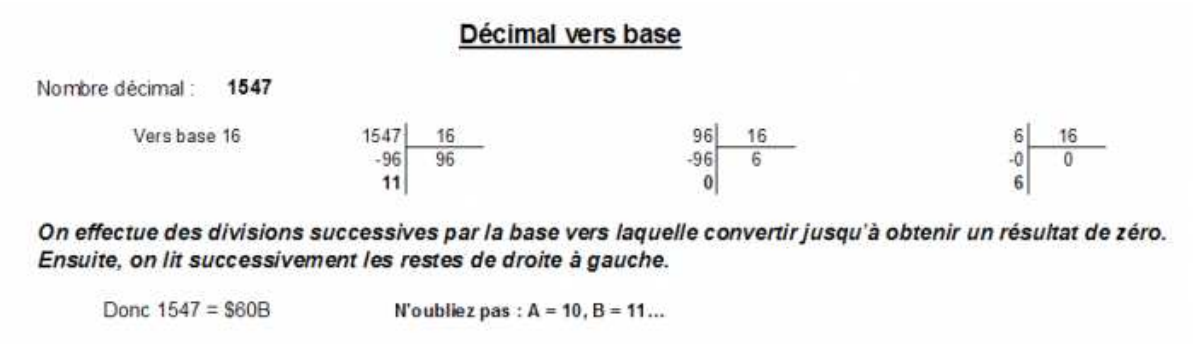
Le binaire correspond à la base 2 et l'hexadécimal à la base 16.
Nous avons pour habitude l'utilisation du système décimal, de base... 10 !
Le binaire n'est composé que de 0 et 1 (2 caractères <=> base 2) tandis que l'hexa se note de 0 à F.
On a donc A correspondant à 10, B à 11, etc.
Bien évidemment, on peut utiliser toutes les bases jusqu'à 36 mais leur intérêt est très limité, pour ne pas dire inexistant !
Pour que TASM puisse différencier les bases, on utilise le symbole « % » pour le binaire et « $ » pour l'hexa.
En effet, 101 peut aussi bien être binaire que décimal... Si vous ne mettez rien, le nombre sera considéré comme décimal.

```
%10011001 ou 10011001b
```

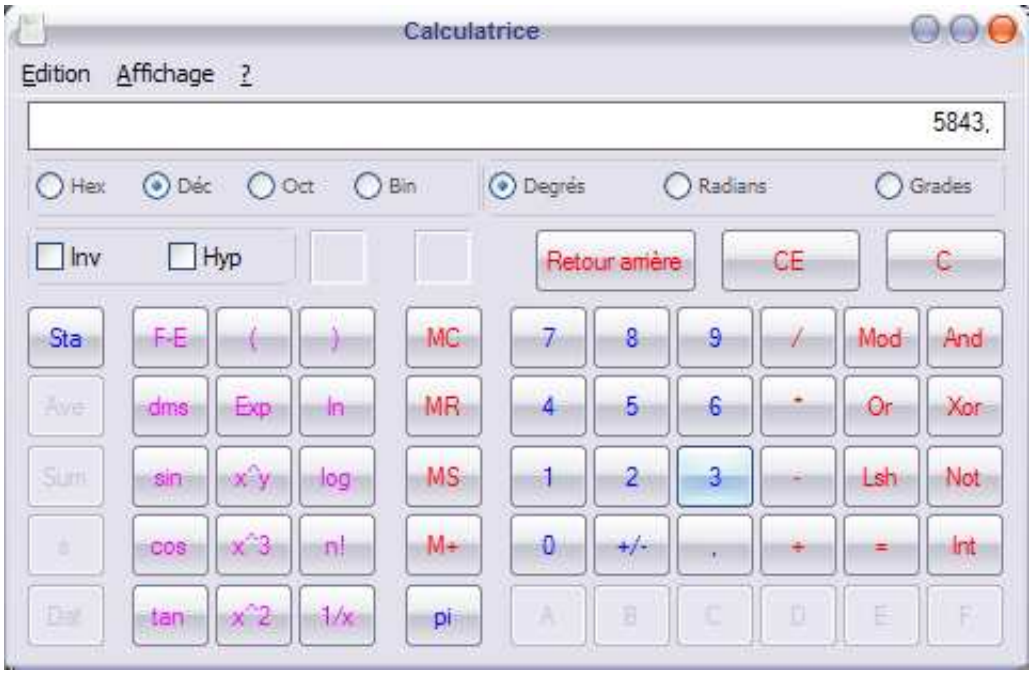
```
$5C ou 5Ch
```

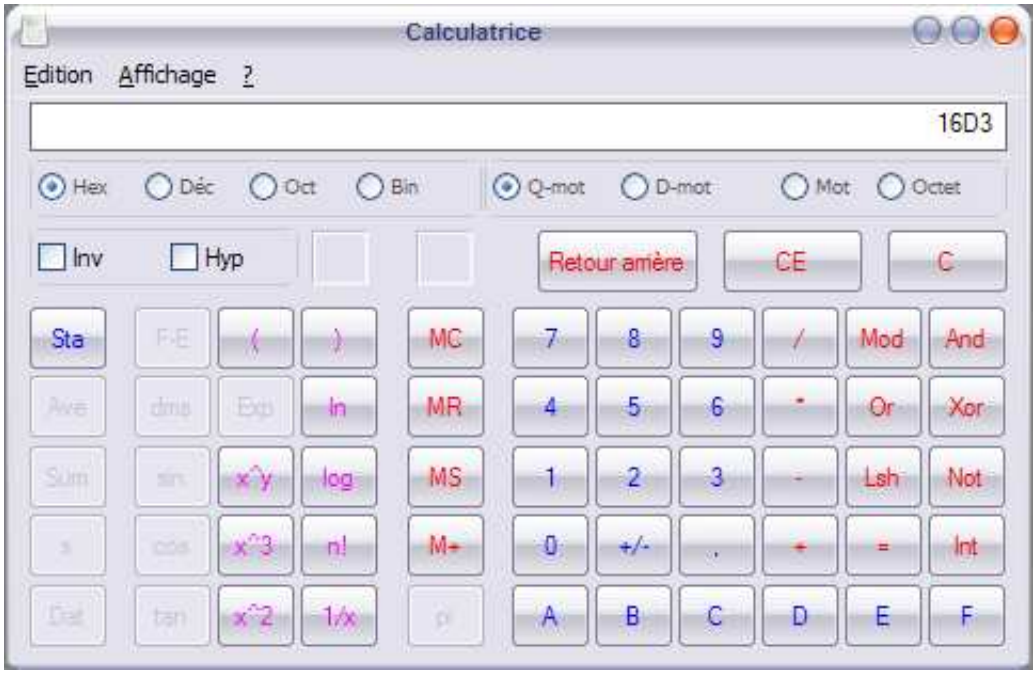
La conversion d'une base vers une autre est assez simple :





Vous pouvez aussi utiliser la calculatrice de Windows en affichage scientifique :





L'utilité de ces bases vous sera expliquée plus tard... :p
Retenez juste les notations pour le moment !

Vous n'avez normalement plus qu'à écrire un programme pour pouvoir l'utiliser...

Matériel nécessaire

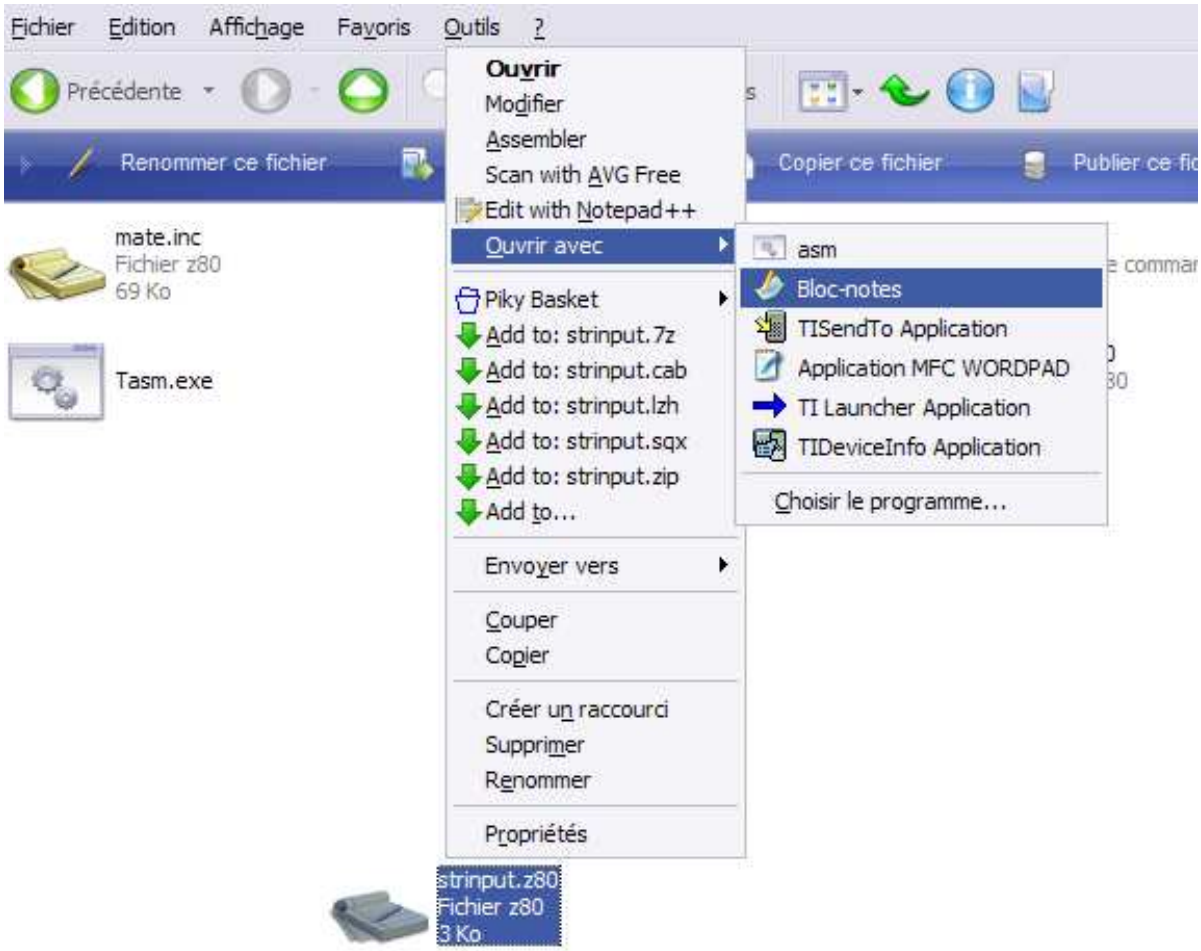
C'est bien beau de programmer, mais généralement ça ne suffit pas ; un matériel minimum est requis.
Vous connaissez peut-être la galère de rechercher ces bons fichiers sur le net ?
Eh bien comme je suis très gentil je vous éviterai cette souffrance ! ^^

Édition

De nombreuses IDE sont présentes sur le net mais personnellement, je leur préfère de loin le Bloc-notes ; lui n'a aucune prétention !
En plus, vous n'avez pas à le télécharger, et ses fonctions seront bien suffisantes pour l'usage que vous en ferez.
Vous devez donc créer un nouveau document texte à l'aide d'un clic droit et modifier son extension en **.z80**.
Répondez « oui » à la question idiote que vous pose votre ordinateur et créez si nécessaire une association de programme :

1. clic droit sur votre fichier z80 ;
2. **Ouvrir avec** ;
3. **Choisir le programme** ;
4. le Bloc-notes devrait déjà être présent dans la liste suivante.

Sélectionnez-le, cochez **Toujours utiliser ce programme pour ouvrir ce type de fichier**, et cliquez sur **OK**.
Cela permet de lancer directement le Bloc-notes quand vous double-cliquerez sur un fichier z80.



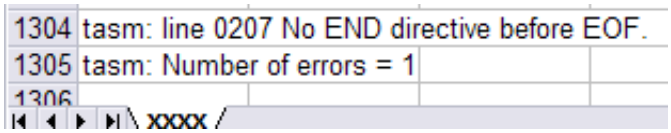
Vous êtes maintenant prêts pour l'édition. :)

Compilation

Bon, là y a pas 36 solutions ! Le compilateur le plus connu (et le seul que je connaisse...) est sans conteste **TASM**, qui fonctionne avec une table de référence, un fichier nommé **TASM80.TAB**.
Cette application permet de transformer votre texte en un fichier binaire... inutilisable !
Heureusement il y a Findus un autre compilateur qui permet de compiler ce fichier **.bin** en programme directement exportable sur la calculatrice, ô joie ! :D
Ce compilateur s'appelle... **DEVPAC8X.COM** et comme son nom l'indique, il produit un fichier en **.8XP** lisible par la calculatrice ou par un émulateur.

> C'est bien beau toutes ces applications, mais comment on les utilise ?

Je supposerai que vous préférez lancer une application plutôt que de créer plusieurs fichiers pour gérer ces deux compilateurs.
Et c'est pour ça qu'a été créé le fichier **asm.bat** qui gère le tout pour ne vous laisser que le programme en cas de réussite.



En cas d'échec (oui, ça arrive :p), vous obtiendrez un fichier **.XLT** s'ouvrant avec Excel qui détaillera vos erreurs.

Et comme promis, voilà tous ces fichiers dans un package :

Télécharger le kit de développement (http://www.mediafire.com/download.php?bsbvzxeictb)

> Tous ces fichiers doivent **impérativement** être dans le même dossier.

Pour utiliser **asm.bat** facilement, créez un fichier **lancement.bat** (appelez-le comme vous voulez, il faut juste que l'extension soit en .bat. Utilisez un fichier texte puis changez le .txt) et tapez le code suivant :

```
asm [nom du fichier z80 sans l'extension]
```

Par exemple, pour compiler le fichier « MATHS.z80 », tapez

```
asm MATHS
```

Ensuite, enregistrez votre .bat et double-cliquez dessus.

> Votre programme (fichier z80) doit également se trouver dans le même dossier que les applications.

Et tant qu'on est dans les téléchargements, je vous propose les derniers (au moment où j'écris ces lignes !) OS :

- Pour TI-83+ ou TI-83+ SE (http://www.mediafire.com/download.php?19acmbynyxv)
- Pour TI-84+ ou TI-84+ SE (http://www.mediafire.com/download.php?fqjbyunqbzq)

> Et c'est quoi au juste un OS ? Parce que Operating System, ça me parle pas vraiment...

Personne ne saurait expliquer ça mieux que ses créateurs je suppose... ^^

Citation : Texas Instruments

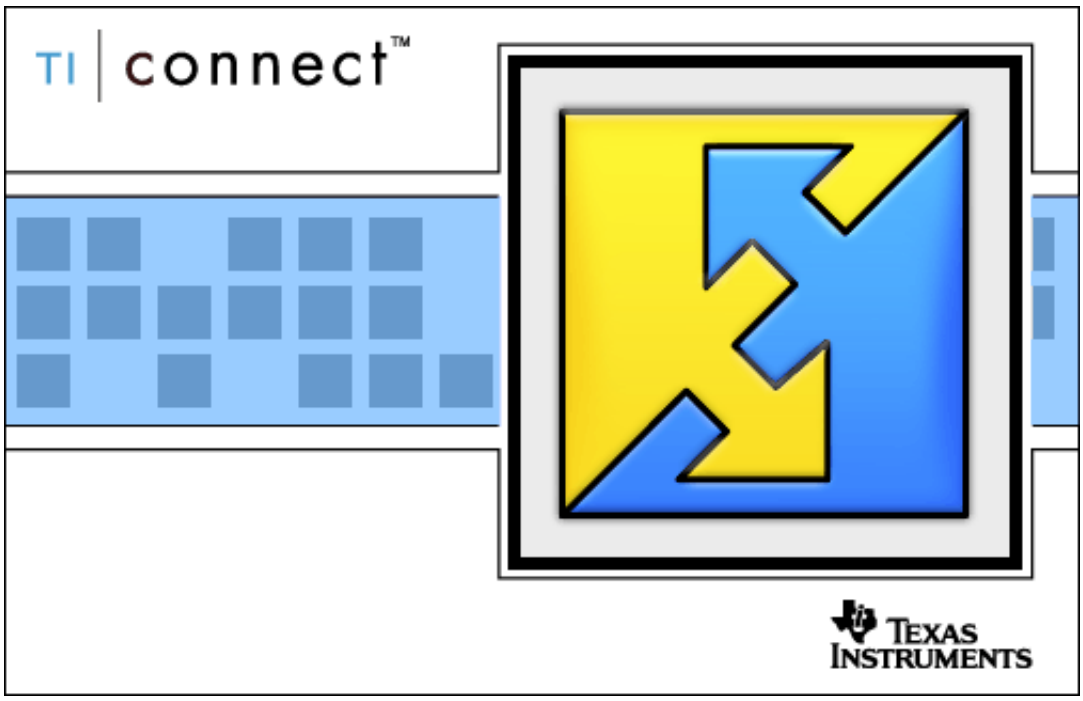
> Le système d'exploitation (OS) de votre calculatrice fait référence au programme de configuration qui permet à votre calculatrice de fonctionner. Comme pour les ordinateurs, le système d'exploitation de votre calculatrice peut être mis à jour.

Transfert

OK, vous avez un programme pour calculatrice sur votre ordinateur.
Il ne serait pas mieux sur une calculatrice ? :p
Il vous faut donc un logiciel de transfert pour échanger des données de l'ordinateur vers votre TI, ou vice-versa.

Mon choix s'est porté sur le logiciel TI-Connect qui propose en plus quelques options sympa comme la capture d'écran, la conversion et le transfert d'images...
De plus, la version Windows ne fait pas de discrimination ; toutes les calculatrices (de TI-83+ à TI-84+ SE) sont acceptées.

Télécharger TI-Connect (http://www.mediafire.com/download.php?dub1dca3mgj)

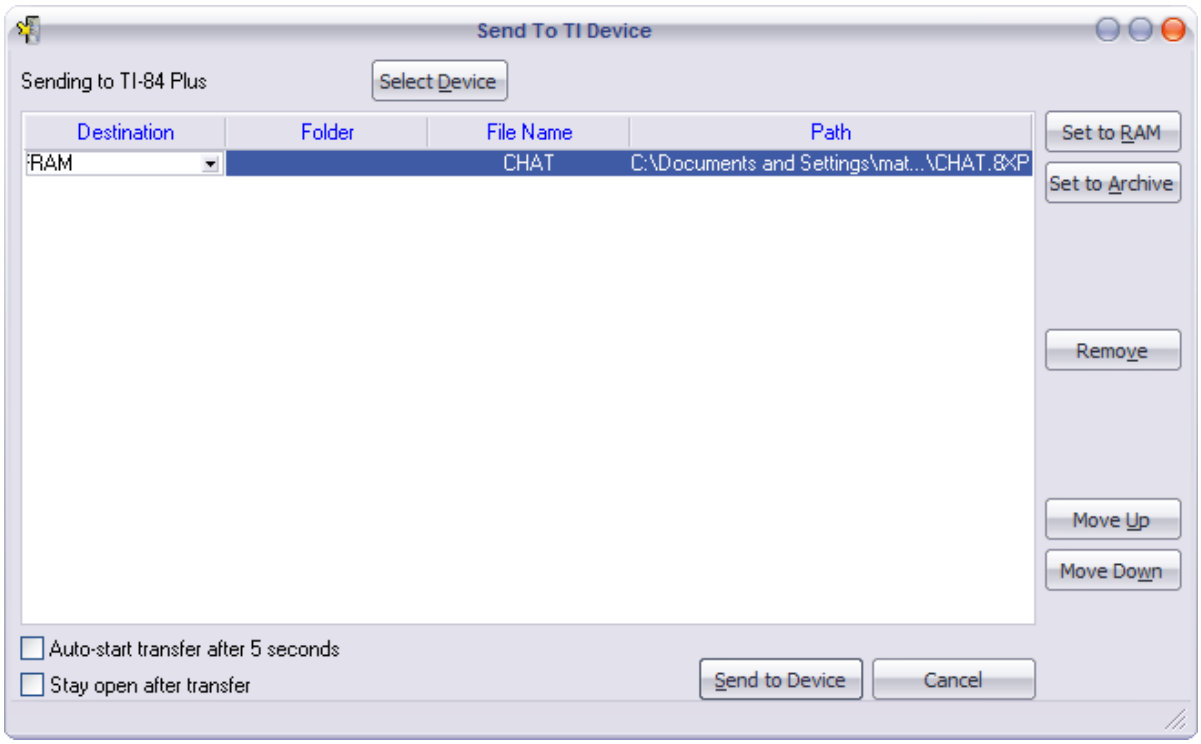


L'installation est extrêmement simple et intuitive, je ne perdrai pas de temps à l'expliquer.

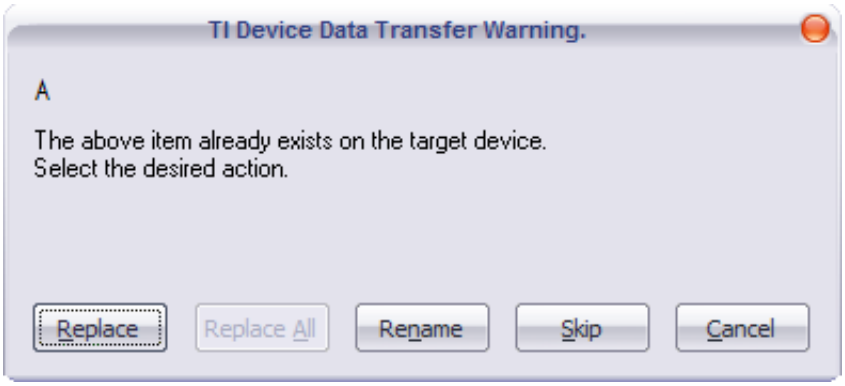
Ce qui nous intéresse le plus pour le moment, c'est l'application de transfert ordinateur -> calculatrice.
À la base, si vous double-cliquez sur un fichier en .8XP, l'ordinateur vous sortira un truc du genre « il n'y a pas de programme associé à cette action ».
Il faut lui associer l'application **TISendTo.exe** présente par défaut dans le dossier **Program files\TI Education\TI Connect**.
Vous démarrerez ainsi le logiciel de transfert en double-cliquant sur un fichier 8XP.

> Bien sûr, c'est ce que je préconise, mais rien ne vous empêche d'associer un autre programme.

Voici un screenshot :



Comme vous pouvez le remarquer, plus intuitif, tu meurs ! :D
De plus, si vous essayez de transférer un programme dont le nom est déjà présent, aucun bug ne sera généré !



J'en profite pour préciser que le nom du programme n'est pas modifiable après la compilation. Si vous changez le nom du fichier 8XP, c'est tout de même le nom de départ qui apparaîtra dans vos programmes. La preuve : j'avais compilé ce fichier en l'appelant « A.z80 » puis renommé le fichier final en « B.8XP » et regardez... le programme est transféré sous le nom A !

Voilà le plus important. Les autres applications sont moins utiles, je vous laisse les découvrir seuls comme des grands ! :p

---

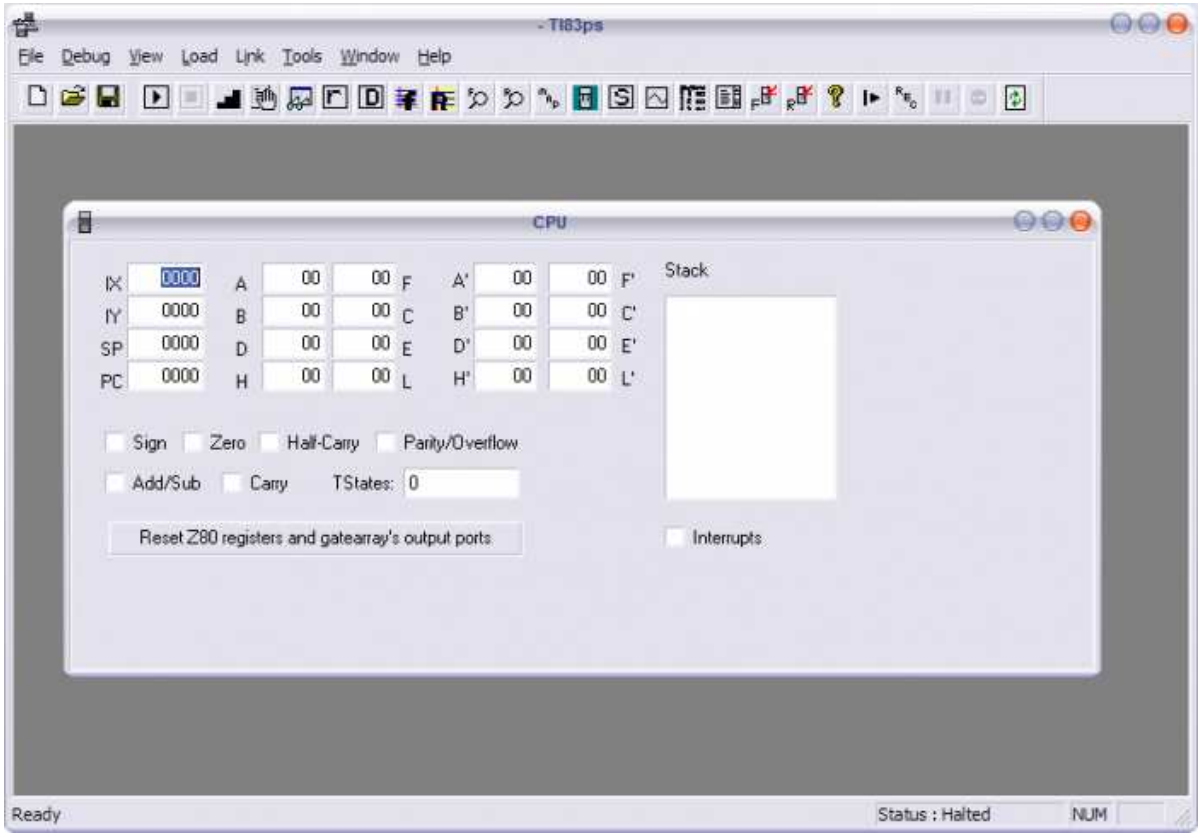
Émulation

Si vous n'êtes pas au courant, il est temps que je vous prévienne : l'ASM génère des bugs beaucoup plus « élaborés » que le TI-Basic ! :-°
Ça passe par l'impossibilité de lancer des programmes, le sempiternel « Mem cleared », le blocage pur et simple, ou encore un écran bleu (oui ! Bleu !)...
Vous n'avez pas fini de vous amuser ! :lol:
Je vous rassure, il suffit généralement d'enlever les piles d'alimentation et de les remettre pour résoudre le problème.
Au pire, vous devrez carrément retirer la « back-up battery » et laisser patienter quelques jours.
Heureusement, il existe des émulateurs qui prennent des risques à la place de votre TI. En cas de plantage, vous avez juste à fermer le logiciel !

> Je précise que je ne pourrai en aucun cas être tenu responsable d'éventuels dégâts causés à votre calculatrice puisque je compile et vérifie moi-même les programmes d'exemple complets.

Voyons... je propose TI-83+ flash debugger !
Télécharger (http://www.mediafire.com/download.php?e9vyjumnttf)
Pourquoi « flash » ?
Parce qu'il prend des photos supporte les applications. Cela s'avère très utile pour tester un programme sous **MirageOS** par exemple.
Une fois le logiciel lancé, **File/New**, et choisissez votre calculatrice.

> Il n'y a quasi aucune différence en ce qui concerne les programmes entre une TI-83+ ou une TI-84+. Si vous possédez cette dernière, vous pouvez quand même utiliser le logiciel.

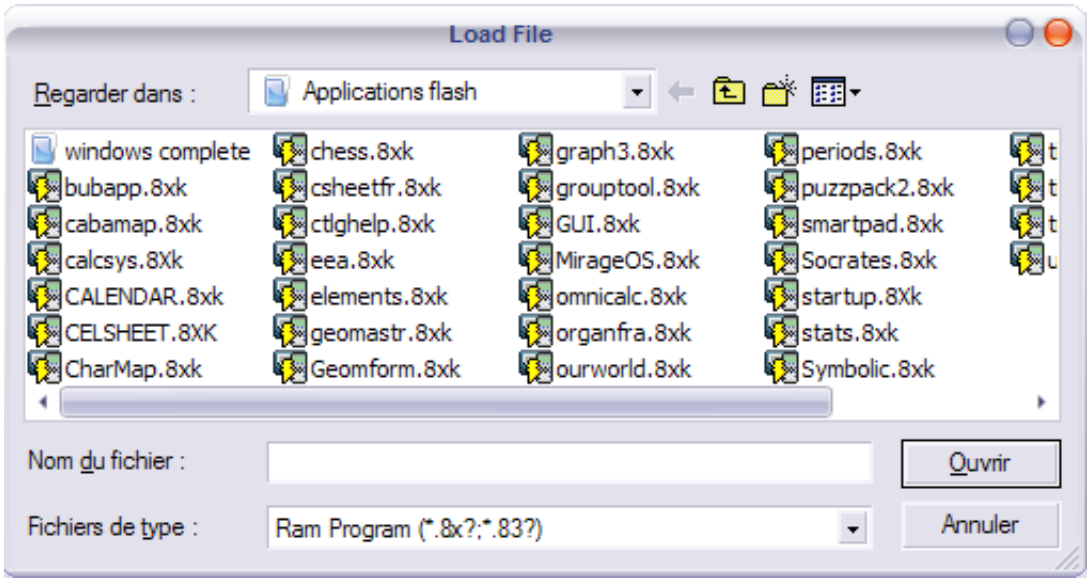


Pour lancer la calculatrice virtuelle, appuyez sur la flèche noire (4e icone en partant de la gauche), et sur le carré pour la stopper.

> Ces commandes ne correspondent en aucun cas à l'allumage ou à l'extinction de la TI ; elles n'agissent que sur l'application.

Pour charger un programme ou une application flash, choisissez **Load** puis Application**Application...** ou RAM file**RAM file...**.

La calculatrice doit être stoppée pour pouvoir utiliser cette fonction.



Je déconseille **fortement** de charger un programme dont le nom est déjà présent dans la calculatrice virtuelle...

> Après plusieurs utilisations, mon logiciel refusait de se lancer. Si cela vous arrive, je n'ai comme solution que le redémarrage. >_

Les principales fonctions sont le chargement de programmes et l'émulation de la TI. Je n'expliquerai donc rien d'autre, d'autant plus que je n'ai pas encore tout compris !
Mais cela devrait vous suffire. ;)

Et GNU/Linux alors ?!...

Cette sous-partie a été créée par **saimoun**, merci à lui !

1. **z80asm à la place de Tasm**, il est dans les paquets de la plupart des distributions, sinon faîtes une recherche sur Google. Son gros défaut (ou celui de Tasm, tout dépend de quel point de vue on se place, niark niark :diable:) étant qu'il a une syntaxe légèrement différente de Tasm. Voir la doc ici : http://www.nongnu.org/z80asm/usage.html (http://www.nongnu.org/z80asm/usage.html).
   Je donne quand même les équivalences :
   - #define Nom_de_la_macro (arg1, arg2, ...) code_de_la_macro_sur_une_ligne avec Tasm contre

     ```
     Nom_de_la_macro: macro arg1, arg2, ...
             code_de_la_macro
             qui_peu_s_etendre_sur_plusieurs_lignes
             ; Genial_non ?
     endm
     ```

     avec z80asm.
   - _fonction = XXXXh avec Tasm contre _fonction: equ XXXXh avec z80asm.
   - .dw, .db, .org, .etc... avec Tasm contre dw, db, org, etc... (enlevez tous les points ".") avec z80asm.

   Si avec tout ça vous trouvez encore des bugs avec un programme qui marchait avec Tasm, regardez sur la doc en ligne (lien un peu plus haut), sinon tapez **man z80asm** en console et vous aurez une documentation plus complète.

2. **Tilem à la place de VirtualTI**, fonctionne très bien, téléchargeable sur http://lpg.ticalc.org/prj_tilem/ (http://lpg.ticalc.org/prj_tilem/).

3. **bin2var (http://www.ticalc.org/archives/files/fileinfo/143/14394.html) à la place de Devpac8x** (je l'ai cherché celui-là...), en plus, il peut convertir en 82p, 83p, 8xp, 85s, etc. Note : le programme est distribué pour Windows, mais le code source est distribué, il faut juste le recompiler pour Nux (**gcc -o bin2var bin2var.c** dans le dossier où l'archive est décompressée).

4. **Tilp à la place de TI-Graph Link** du programme de transfert, fonctionne très bien avec le câble TI GraphLink, mais aussi le GreyLink, les câbles séries et parallèles (voir le site ftp83plus.net (http://www.ftp83plus.net/software.htm)).

Ci-dessous le makefile que j'utilise pour compiler (bande de veinards j'ai fait tout le boulot à votre place :p). Pour le faire fonctionner : vous le mettez dans le même dossier que votre fichier source (portant l'extension *.z80), vous tapez **make** dans une console (dans le bon dossier, bien sûr) et il vous produira un fichier du même nom ayant l'extension *.8xp, ce dernier étant directement ouvrable par tilem.
Note : vous pouvez aussi taper **make clean** pour effacer les fichiers intermédiaires, et **make mrproper** pour effacer les fichiers intermédiaires et les fichiers finaux (avec l'extension *.8xp).

```
COMP = z80asm
FLAGS = --verbose
CONVERT = bin2var

SRC = $(wildcard *.z80)
BIN = $(SRC:.z80=.bin)
EXE = $(SRC:.z80=.8xp)

all: asm

asm: ${SRC}
        @${COMP} ${FLAGS} --input ${SRC} --output ${BIN}
        @${CONVERT} ${BIN} ${EXE}

.PHONY: clean mrproper

clean :
        @rm -f *.bin
        @rm -rf *~
        @echo "Nettoyé."

mrproper: clean
        @rm -f *.8xp
        @echo "C'est vide !"
```

Voilà, je trouvais que ça manquait, partout sur le Net, on ne voit que des tutos (même ceux en anglais et réputés « très bons ») qui balancent du Tasm et Cie, pas un seul n'explique qu'on peut très bien faire ça avec GNU/Linux !

Vous devriez à présent être en mesure de compiler vos programmes **sans bugs** en fichiers directement transférables, et de les transférer.
Je vous sens impatients de commencer... :)

OK, maintenant on peut passer à l'installation.

Le commencement

Eeeeeeehhhhh oui, il faut bien commencer par le commencement... :-°
Surtout, n'hésitez pas à vous attarder, la compréhension est essentielle pour la suite.

Le header... incontournable !

C'est un bon début...
En effet, votre programme commencera **toujours** par un header.
Voyons sa structure :

```
#define end .end
#define END .end
#define equ .equ
#define EQU .equ
#include "ti83plus.inc"
.org 9D93h
.db $BB,$6D
```

> Ce header vient d'un code d'exemple mais il en existe de nombreux. Votre programme ne commencera donc pas forcément par celui-ci.

Étudions ça progressivement ; vous connaissez peut-être le rôle des #define [texte1] [texte2] mais je considérerai que non.
Ils servent à remplacer toutes les occurrences de [texte1] par [texte2] dans le programme.
TASM est une application très tatillonne qui reconnaît des termes très précis. Par exemple, la fin du programme lui est spécifiée par **.end** et pas end ou .END (Oui, la casse aussi a une importance >_).
Selon vos envies ou vos habitudes, vous serez peut-être amenés à utiliser d'autres termes et dans ce cas, pensez aux #define !

> O.K., **.end** est une instruction de fin de programme mais assez importante pour que j'en parle maintenant, car si vous l'omettez, vous aurez droit à **No END directive before EOF**. :o Mignon non ?

Voyons la suite...
Un **#include** ; ce terme n'est peut-être pas inconnu si vous connaissez d'autres langages. Son utilité est d'insérer le contenu d'un fichier à l'endroit de son appel avant la compilation du programme.

```
#include "ti83plus.inc"
```

Ceci inclura le texte du fichier **ti83plus.inc** entre **#define EQU .equ** et **.org 9D93h** dans l'exemple.
Je vous le remets si vous êtes fainéants. :-°

```
#define end .end
#define END .end
#define equ .equ
#define EQU .equ
#include "ti83plus.inc"
.org 9D93h
.db $BB,$6D
```

Les fichiers en .inc (include) sont en fait de simples fichiers texte dont l'extension a été modifiée pour distinguer son usage.
Ils stockent généralement des données relatives à la calculatrice. J'y consacrerai une partie plus tard ; ne vous en souciez pas trop pour l'instant.

Dernière instruction (enfin !) : **.org**
C'est un peu plus complexe. Indispensable, mais vous pouvez oublier son utilité. Je vous la donne quand même (:D) : il spécifie où le programme doit être chargé dans la mémoire pour son exécution.

> Je profite de ceci pour prévenir les utilisateurs de TI-83 qui auraient mal lu l'introduction ; l'adresse du lancement des programmes varie pour cette calculatrice. À titre indicatif, je la donne quand même : $9327

N'oubliez **SURTOUT PAS** la ligne **.db $BB,$6D**.
Si elle est omise, le programme sera compilé mais refusera de se lancer en vous reprochant un **ERR : INVALID** !
.db sert à spécifier une donnée ; j'expliquerai la signification de cette ligne plus tard. :p

Essayez aussi de vous imposer une mise en forme : pas de tabulation pour le header, les labels et ce qui commence par un point, et une tabulation pour le reste.

J'allais oublier les .equ...

Cela fonctionne sous la forme `[texte] .equ [nombre]`
Il remplace dans le programme tous les [texte] par les [nombre].

> Et concrètement, ça sert à quoi ?

À faire le café o_O Naaaaaaaan !
Ça sert dans la plupart des cas à **définir des romcalls** et, euh, bah... C'est le sujet suivant ! :p

> L'ordre des instructions précédentes est totalement libre, on peut même les placer après le **.org** ; l'important est qu'elles définissent des valeurs avant leur utilisation.

Les romcalls

Bon, concrètement, vous savez maintenant faire des programmes vides. :euh:
Retenez la forme :

```
header

#define ------
#include ------
.org ------
.db $BB,$6D

Corps du programme

Instructions diverses et variées...

.end
```

Les romcalls sont un exemple d' « instructions diverses et variées ».

Ce sont des « mini-programmes » stockés dans la mémoire de la calculatrice qui permettent de réduire considérablement la complexité du programme. On peut faire la comparaison avec les fonctions en C ; une romcall prend des arguments en entrée, et retourne une ou plusieurs données en sortie.
Elles peuvent aussi bien avoir un effet visible (comme effacer l'écran) qu'invisible (on verra ça plus tard !).

On les appelle (à prendre dans le sens « appel de fonction ») différemment selon les habitudes, et j'ai pour habitude d'utiliser **bcall(xxxx)** où « xxxx » est le nom de la romcall.
Ça n'a pas d'importance de toute façon, utilisez un **#define** avec le texte que vous voulez.
Le code correspondant est le suivant :

```
rst 28h
.dw xxxx
```

Je ne vous demande pas de le comprendre.

On a donc un texte à remplacer par un autre, à moins que vous ne préfériez taper deux lignes de code au lieu d'une ! :p
Ça ne vous rappelle rien ?

Il faut utiliser #define. :)

On inclura donc dans le header :

```
#define bcall(xxxx) rst 28h \ .dw xxxx
```

> « \ » permet de placer deux lignes l'une à la suite de l'autre.

Vous avez peut-être remarqué que cette ligne n'apparaît pas dans le header de la partie précédente ; eh bien c'est juste qu'elle est déjà présente dans le fichier include. Vous commencez à comprendre leur utilité, non ? :)

> Bon, « xxxx » désigne un nom, alors que les romcalls correspondent à des adresses mémoires ?

En fait, « xxxx » peut aussi bien désigner un texte qu'un nombre. À la base, ce sont bien des adresses mémoires, donc des nombres que nous utilisons. Mais avouez que **bcall(_ClearDraw)** est plus explicite que **bcall($4540)**.

On remplace un nombre par du texte. Le compilateur doit donc remplacer du texte par un nombre, ce qui nous amène à utiliser **.equ** ! :D
Rappelez-vous...

```
[texte] .equ [nombre]
```

Donc pour continuer sur ma lancée, on utilisera dans le header : `_ClearDraw .equ $4540`

> Par convention, les noms de toutes les romcalls débutent par un underscore « _ ». Après c'est vous qui voyez !

Eh oui, $4540 (inutile de vous rappeler que c'est de l'hexadécimal ^^) n'a pas été pris au hasard ; il correspond à l'adresse mémoire où se trouve la romcall qui gère l'effacement de l'écran ; elle ne nécessite aucun argument.

> Et quand il y en a, on les passe comment ? Et comment on fait pour savoir à quelle adresse correspond tel effet ?

Si j'ai pris un exemple où il n'y en a pas besoin, c'est que je vais en parler plus tard (:ange:), patience (à moins que je décide de ne pas en parler du tout, yêk :diable:) !
En ce qui concerne les adresses, je vous les donnerai en fonction des besoins du tutoriel. Si vous êtes pressés, jetez donc un coup d'œil à des **fichiers include** !

> Certaines romcalls « détruisent » des registres. C'est-à-dire qu'elles ne conservent pas leur valeur de départ. Par exemple, _ClearDraw (qui se nomme réellement _clrlcdfull) détruit **tous** les registres !

Vous en savez maintenant assez pour compiler votre 1er programme (Tatsiiiin !!) !

Votre 1er programme !

Si j'ai bien suivi (^^) , vous en connaissez maintenant assez sur le header et le corps du programme.
D'accord, pas beaucoup mais assez !
Nous allons donc créer un programme qui efface l'écran et qui ne plante pas.
Voyons... Nous n'avons besoin que d'une romcall qui a été définie précédemment.
Et j'allais oublier... L'instruction ASM pour la fin d'un programme est **ret**, qui nécessite donc une tabulation.

> Bah ? Je croyais que c'était « .end » pour la fin du programme ?

Attention, **.end** indique à TASM où il doit s'arrêter de compiler ; tandis que **ret** agira en tant que fermeture du programme lors de son exécution.

N'hésitez pas à relire les 1er chapitres jusqu'à bien les comprendre, je ne vous en voudrai pas ! :ange:

Selon moi, on obtient quelque chose comme ça :

```
#define bcall(xxxx) rst 28h \ .dw xxxx
_ClearDraw .equ $4540
.org $9D93
.db $BB,$6D
        bcall(_ClearDraw)
        ret
.end
```

> TASM laisse tout de même la possibilité de laisser des commentaires. Il suffit d'utiliser un point-virgule devant son texte ; il ne sera ainsi pas considéré comme du code.

O.K., j'autorise des différences au niveau des « **appellations** », c'est-à-dire « _ClearDraw » et « bcall(»(et des **#define** si vous utilisez une autre notation).

> Au secours !! Ça ne marche pas !!

Si votre compilation échoue, retentez-la en utilisant le code ci-dessus tel quel, pour voir si votre erreur vient du code ou des applications. Si ça ne marche toujours pas, éclatez votre ordinateur à coup de relisez la première partie, vérifiez que vous avez toutes les applications, qu'elles correspondent à votre calculatrice... Si malgré cela la compilation échouait une fois de plus, envoyez-moi un message avec l'erreur indiquée. :-°

Une fois que vous avez un beau fichier transférable... transférez-le !
Considérons « xxxx » comme le nom de votre programme (vous l'appelez comme vous voulez, je n'impose rien !).
Pour l'exécuter :

```
Asm(prgmXXXX)
```

La commande **Asm(** se trouve dans le catalogue (2ND+0).

```
                Done
```

Logiquement, votre écran devrait ressembler à ça (après appui sur **ENTER** :p) :

> Ne vous inquiétez pas pour le Done, nous verrons comment le faire disparaître plus tard.

Quelle émotion, non ?
Non ?
O.K. on continue. :pirate:

C'est bien beau d'avoir fait un programme, mais je suis sûr que vous l'avez trouvé léger et que vous en voulez plus ?
Nous allons donc voir les essentiels, c'est-à-dire les variables (enfin !), les fichiers include et les instructions du z80.
Courage ! ^^

Qui peut se passer des variables ?

> Certainement pas moi. Et comme c'est moi qui fait le tuto... :lol:
> En effet, c'est bien beau les romcalls, mais vous voudrez peut-être stocker et utiliser des valeurs, ou même utiliser des romcalls plus complexes (y a intérêt ! :pirate:) ?
> Eh bien il vous faut des variables !
> Ce chapitre sera l'un des plus longs mais sa compréhension est **essentielle** !

Variables prédéfinies

Vous êtes au courant qu'il faut utiliser le z80 sur une TI ? Oui ? Parfait. :D
Sachez que ces deux « objets » possèdent chacun leurs propres variables.

- Pour le processeur (z80), il s'agit des **registres**.
- Pour la TI, ce sont des « **system-defined ram variables** ».

Les registres

Vous vous rappelez les fameux arguments à passer ? Eh bien ils le sont grâce aux registres.
Il en existe de 8 et de 16 bits.

> 8 bits = 1 octet = 1 byte.

Ils sont représentés par une ou deux lettres (vous devinez pourquoi ? :euh:).
On dispose en 8 bits des registres **a**, **b**, **c**, **d**, **e**, **f**, **h** et **l** (très original... :D).

Logiquement, ils peuvent donc contenir des valeurs de %00000000 à %11111111, c'est-à-dire entre 0 et 255.
Votre programme ne sera pas compilé en cas de dépassement.

Voyons les registres 16 bits ; il existe **af**, **bc**, **de** et **hl**.
Eh oui, ce sont des combinaisons de deux registres 8 bits, qui peuvent donc prendre des valeurs entre 0 et 65535.

> Il existe également ix et iy, mais ils sont un peu différents (ce sont des index de registre), ne vous en souciez pas pour l'instant.

Quel rapport y a-t-il alors entre les registres 8 et 16 bits ? Eh bien l'un fait partie de l'autre, ce qui m'amène à justifier l'utilisation du binaire et de l'hexadécimal.
En effet ces bases sont adaptées à l'utilisation d'octets ; regardez :

Si **h** contient %10110010 et **l** %00100101, alors on aura **hl** = %1011001000100101.

De même :

Si **b** contient $05 et **c** $8A, alors **bc** = $58A.

Compris ? On va voir ça ! :diable: .
Si **d** contient 54 et **e** 200, **de** = ...

Attention, c'est du décimal !
On a :

- 54 = 00110110b = 36h ;
- 200 = 11001000b = C8h.

D'où **de** = 0011011011001000b = 36C8h = **14024**.

Et ça marche aussi dans l'autre sens (encore heureux !).

Pour traiter des nombres de 16 bits, on utilise généralement l'hexadécimal, et le décimal pour les nombres de 8 bits, mais c'est vous qui voyez !
Non, je n'oublie pas le binaire ! Sa plus grande utilité est ailleurs, patience...

Autre chose : les registres ne se contentent pas tous de ne servir qu'au stockage.
Voyons leurs caractéristiques particulières.

a

Il est appelé **accumulateur**, c'est le plus important des registres 8 bits. Il permet de réaliser les opérations de base et d'accéder à la mémoire.

f

C'est le registre des **f**lags, on ne peut pas y charger de valeur.
J'aborderai les flags plus tard.

hl

Équivalent du registre **a** en 16 bits : il permet de réaliser des opérations et est utilisé pour l'adressage. Sur deux <bytes on aura donc le « **h**high » et le « **l**low byte » d'une adresse mémoire !

bc

Il est utilisé dans des instructions comme un compteur d'octets. En anglais ça donne **b**yte **c**counter ! :)

de

Ce registre sert à mémoriser des adresses mémoire qui serviront de **de**stination.

> Bien sûr, n'importe quel registre (mis à part **f** et donc **af**) peut être utilisé pour du simple stockage de données ; ce que j'ai présenté décrit leurs caractéristiques propres.

Pour charger une valeur dans un registre, on utilise l'instruction **ld** (abréviation de load si vous n'avez pas deviné ;)).

```
ld [destination],[valeur à charger]
```

On peut charger n'importe quelle valeur - en tenant compte de la capacité tout de même - dans n'importe quel registre utilisable (oubliez **af** et **f** pour le moment).

```
ld hl,$547A
ld a,%11001
etc.
```

On peut également charger d'un registre 8 bits vers un autre registre 8 bits sans problème.

Un registre 16 bits doit être chargé dans un autre registre 16 bits en deux étapes.
On n'écrira pas **ld hl,de** mais :

```
ld h,d
ld l,e
```

Possibilités restantes :

- **D'un 8 bits vers un 16 bits** :

  possible, mais le 1er octet du registre 16 bits ne changera pas. Vous devez donc le mettre à 0.

- **D'un 16 bits vers un 8 bits** :

  impossible ; même si le registre 16 bits contient une valeur inférieure à 256, vous aurez une erreur de compilation.

Par exemple, pour stocker **a** dans **de**, on fera :

```
ld e,a
ld d,0
```

Retenez juste les méthodes de chargement pour l'instant, c'est la principale utilité des registres.

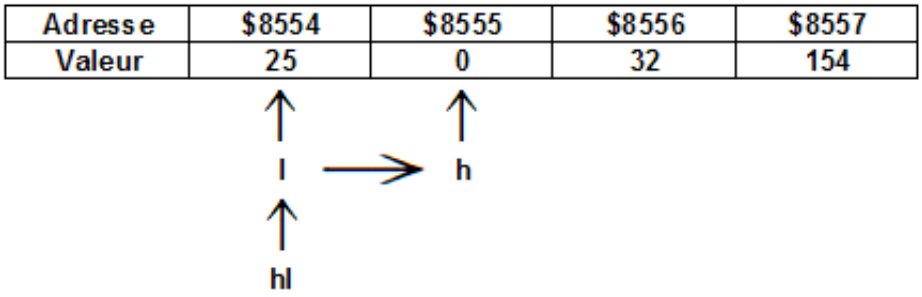
Variables pré-définies

Pour le bon fonctionnement de l'OS, la TI possède dans sa mémoire des informations comme les coordonnées du curseur, le contraste, la liste des programmes, ...
Avec l'ASM, nous pouvons récupérer et / ou modifier leurs valeurs.

> Comment on fait ?

Il faut **absolument** comprendre le fonctionnement de la mémoire avant de lire la méthode.

On peut comparer la mémoire - dans notre cas, la RAM - à une succession de cases les unes à la suite des autres, chacune de ces cases étant identifiée par une **adresse**.
Il faut donc distinguer cette adresse du contenu de la case.
Ces cases ont une capacité d'un octet. Si on place 16 bits à une adresse, 8 bits vont forcement « déborder », selon le schéma suivant :



À noter que les valeurs ont été prises **au hasard**.

> Cela ne doit pas être vu comme un inconvénient, au contraire, car il sera donc possible de modifier deux valeurs côte à côte en une seule fois !

> Une erreur fréquente consiste à croire que c'est le premier octet (high byte -> **h** dans l'exemple) qui sera à l'adresse indiquée, alors qu'il s'agit du contraire.

Si on possède l'adresse d'une valeur, on obtient cette valeur par la notation :

```
(adresse)
```

> Le nombre entre parenthèses est un **pointeur**.
> Le nombre **avec** les parenthèses est la valeur contenue dans la RAM à l'adresse « pointée ».

Il suffit de mettre l'adresse entre parenthèses pour récupérer la valeur contenue à cette adresse.
On utilise également **ld** pour charger des valeurs à des adresses précises, avec quelques restrictions :

- on **ne peut pas** transférer le contenu d'une adresse vers une autre directement, il faut passer par un registre ;
- en 8 bits, **seul l'accumulateur** (registre **a**) peut être utilisé pour récupérer ou pour charger le contenu d'une adresse ;
- on peut utiliser **n'importe quel registre 16 bits** pour récupérer ou charger le contenu d'une adresse ;
- si on récupère le contenu d'une adresse avec un registre 16 bits, alors son premier octet prend **la valeur contenue dans l'adresse suivante** (la valeur contenue dans l'adresse demandée se trouve dans le deuxième octet du registre).

Revenez au schéma ci-dessus ; on aura :

```
ld a,($8556)
-> a = 32
ld hl,($8554)
-> h = 0, l = 25, donc hl = $0019
```

Ce sera tout pour le moment ; retenez bien les restrictions car elles sont nombreuses en ASM ! :p

---

Variables définies par l'utilisateur

C'est bien beau les registres, mais on les utilise tout le temps. Leur valeur est donc toujours changeante. Et vous voudrez peut-être créer vos propres variables, auxquelles vous seuls aurez accès.
Vous devez donc les **définir**.

> Ah, tiens ! 'Faut utiliser **#define** alors ?

NON !!!!
Bon, OK, c'était un piège !
En fait, vous allez devoir stocker vos variables dans la RAM. Et la RAM est constituée... d'adresses, qui sont des... nombres ! :lol:
Après lui avoir trouvé un nom, vous pourrez donc utiliser un **.equ** pour faire correspondre ce nom à une adresse libre.

> Ce fameux **.equ** peut être remplacé par... le signe égal ! Par convention, on utilise **.equ** pour les romcalls et « = » pour les variables.

De plus, les variables peuvent être définies à n'importe quel endroit du programme, tant qu'on ne les utilise pas avant. Par convention, on les place toutes dans le header, avant le **.org** ; il est plus facile de s'y retrouver quand elles sont toutes définies ensemble au début du code.

> OK, mais comment savoir à quel endroit de la RAM les placer ?

Eh bien il existe quatre zones de la RAM dédiées au programmeur :

- saferam1 : commence à l'adresse **$8265** et s'étend sur 768 octets ;
- saferam2 : commence à l'adresse **$858F** et s'étend sur 531 octets ;
- saferam3 : commence à l'adresse **$80C9** et s'étend sur 128 octets ;
- saferam4 : commence à l'adresse **$8039** et s'étend sur 66 octets.

Le terme « saferam » est évidemment une convention, vous pouvez l'appeler comme vous voulez.

> Si vous avez plusieurs variables à définir, prenez garde à la taille de celles-ci ; il faudra passer 2 octets pour définir une variable à la suite d'une de 16 bits.

```
saferam1 .equ $8265
variable_16_bits = saferam1
vaiable_suivante = saferam1+2
```

Mais il est bien sûr possible de stocker ces variables n'importe où dans les saferam.

Vous pouvez sauter des octets en utilisant **+[valeur constante]**, c'est-à-dire que

```
variable_suivante = saferam1+a
```

ne sera pas accepté, contrairement à

```
variable_suivante = saferam+2
```

Bon, vous avez maintenant défini votre variable comme **son adresse dans la RAM.**
On a donc un pointeur sur variable_suivante qui vaut $8267.
Pour récupérer la valeur contenue à l'adresse du pointeur dans un registre 8 bits, vous devez écrire :

```
ld a,(variable_suivante)
```

> N'oubliez pas que **a** est le seul registre 8 bits capable de récupérer la valeur contenue à une adresse.

Pour ce qui est des restrictions, vos variables se gèrent comme les « **system-defined ram variables** » vues plus haut. Il n'est donc pas nécessaire de les rappeler, vous avez juste à lire la partie précédente en cas d'oubli.

Opérations de base

Vous savez maintenant comment stocker et rappeler des nombres, qu'ils soient dans des registres ou dans la RAM.
Mais des nombres sans opération, c'est comme un Mickey sans oreilles ! :D
Vous en avez besoin d'un minimum ; et c'est là que l'ASM devient assez frustrant...

Je précise d'entrée que les instructions qui vont suivre n'agissent **que sur les registres**. Pour effectuer des opérations avec vos variables, vous devrez donc tout d'abord les stocker dans les registres adaptés.

Addition

On utilise l'instruction **add** (incroyable... ^^) avec la syntaxe `add [registre],[registre]` ou `add [registre],[valeur constante]` et quelques restrictions, bien sûr ! :-°

> Je précise que ce que j'appelle valeur constante représente simplement un nombre quelconque ; ce n'est pas un registre ni une variable.

Premièrement, pas d'addition entre registre 8 bits et registre 16 bits.
Et ce n'est pas fini !

Registres 8 bits

Vous pouvez vous y attendre, on ne peut additionner une valeur qu'au registre **a** (ce n'est pas pour rien qu'il s'appelle l'accumulateur ! :lol:).
On peut lui ajouter n'importe quel registre 8 bits ou valeur constante (pas de contenu d'adresse, je le

répète) ; le résultat sera stocké dans l'accumulateur.

Registres 16 bits

Je vous avais dit que **hl** était l'équivalent de **a** en 16 bits ? Malgré cela, on ne peut additionner que les registres 16 bits à **hl**.
Le résultat sera mémorisé dans... **hl** !

Accepté :

```
add a,b
add a,54
add hl,de
```

Refusé :

```
add hl,2145
add a,hl
add hl,a
```

Incrémentation, décrémentation

Je ferai court !
Les instructions correspondantes sont respectivement **inc** et **dec** qui fonctionnent sur tous les registres.

```
inc [registre]
dec [registre]
```

Le résultat est stocké dans le registre qui a servi au calcul.
Cette instruction est plus rapide qu'une addition ou soustraction de 1. Ne l'oubliez pas ! ^^

Soustraction

Les restrictions sont exactement les mêmes que pour l'addition.
On utilise **sub** pour soustraire une valeur constante ou un registre 8 bits à **a** et **sbc** pour soustraire un registre 16 bits à **hl**.
Le résultat est stocké respectivement dans **a** ou **hl**.

```
sub b ; a = a - b
sub 5 ; a = a - 5

sbc hl,de ; hl = hl - de
```

Multiplication, division

Très énervant ! :colere2:
En effet, on dispose seulement de quelques instructions des niveaux de bits, qui n'agissent que sur les registres 8 bits...
On dispose de **sla**, qui multiplie par deux et de **sra** qui divise par deux (ces instructions agissent comme telles sur les nombres **signés** que l'on verra un peu plus tard).

```
sla [registre 8 bits]
sra [registre 8 bits]
```

> Si le résultat de la division ne tombe pas juste, il est arrondi à l'entier inférieur.

Vous l'avez compris, rien que pour diviser par 5, c'est déjà la galère, d'autant plus que les registres ne gèrent pas les nombres décimaux !
À la base, sla et sra ne servent pas vraiment au calcul d'ailleurs...
Pour multiplier, vous pouvez toujours vous arranger avec l'addition.

Multiplier **a** par 9 :

```
ld b,a
sla b ; b = 2a
sla b ; b = 4a
sla b ; b = 8a
add a,b ; a = a+b = 9a
```

Mais ne vous inquiétez pas, il existe une autre méthode pour les calculs, beaucoup plus complète mais plus lente ; je l'aborderai évidemment ! :)

> Le « lent » en Assembleur est relatif ; il reste bien plus rapide que n'importe quel algorithme en Basic, mais retenez que les instructions sont beaucoup plus rapides que les romcalls, même si la différence n'est pas visible à votre niveau.

Le stack

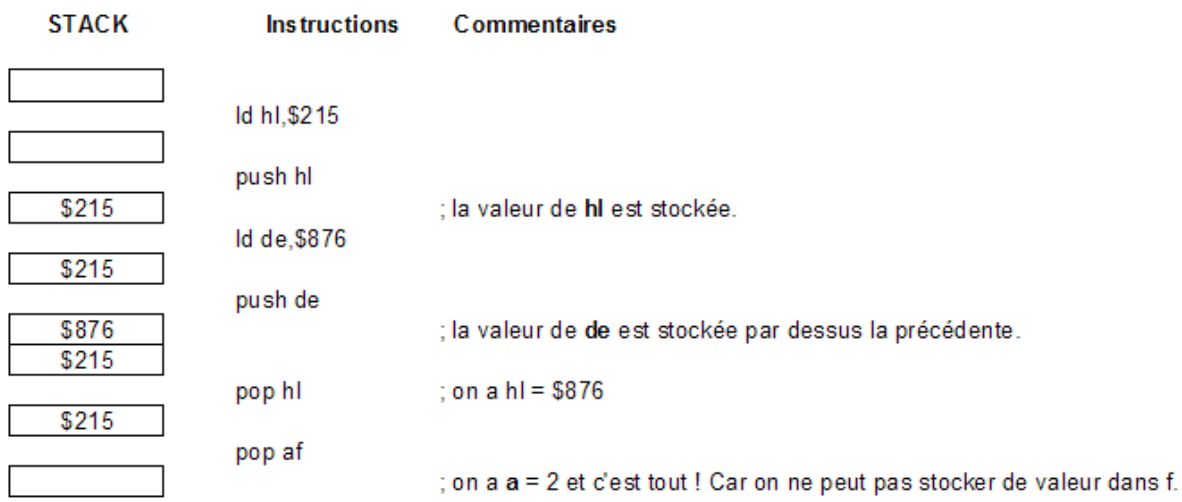
Yeah ! Ça fait style, non ? :soleil:
On va quand même traduire !
« Stack », c'est la pile. Attention, pas la pile de la calculatrice (:p) mais plutôt une pile d'objets.
Ainsi, vous n'avez accès qu'au haut de la pile. Si vous voulez un objet placé plus bas, il faudra d'abord enlever ce qu'il y a dessus.
Concrètement dans notre cas, il s'agit d'une pile de valeurs que vous déciderez d'« empiler ».

Seuls des registres 16 bits peuvent être empilés et retirés.
Mais comme vous savez que les registres 8 bits sont contenus dans les 16 bits, cela n'a pas dû vous effrayer ! :lol:
On utilise les instructions **pop** pour retirer et **push** pour empiler.

```
pop [registre_16_bits]
push [registre_16_bits]
```

Dans le cas de **push** [**registre**], [registre] est sauvegardé sur le haut de la pile.
Quand on a **pop** [**registre**], alors la valeur du haut de la pile est stockée dans [registre].

Voici un petit schéma pour vous permettre d'y voir clair :



Le plus dur à comprendre est que le registre sous lequel est retirée une valeur est totalement indépendant de celui qui a stocké la valeur retirée.

Le stack n'est pas souvent utilisé mais il reste utile lorsque vous devez par exemple passer un argument tout en conservant une valeur dans le même registre.

Il existe des solutions plus rapides pour la sauvegarde mais elles sont beaucoup plus complexes ; je les expliquerai peut-être plus tard.

Nombres négatifs

J'entends déjà des hauts cris : « oui, on me dit que les registres peuvent contenir des nombres entre 0 et 255 alors je vois pas ce que les négatifs viennent faire là ! ».
Eh bien vous avez en partie raison ! :p
En fait, cette affirmation est juste. Le signe d'un registre vient de l'interprétation de ses bits.
En d'autres termes, vous pouvez choisir de voir un registre comme positif ou négatif !

Vous vous souvenez de la structure des registres ?
8 bits ou 16 bits, utilisés entièrement pour le stockage de nombres positifs.
Dans ce cas, on dit que le nombre est unsigned (non signé, quoi...).

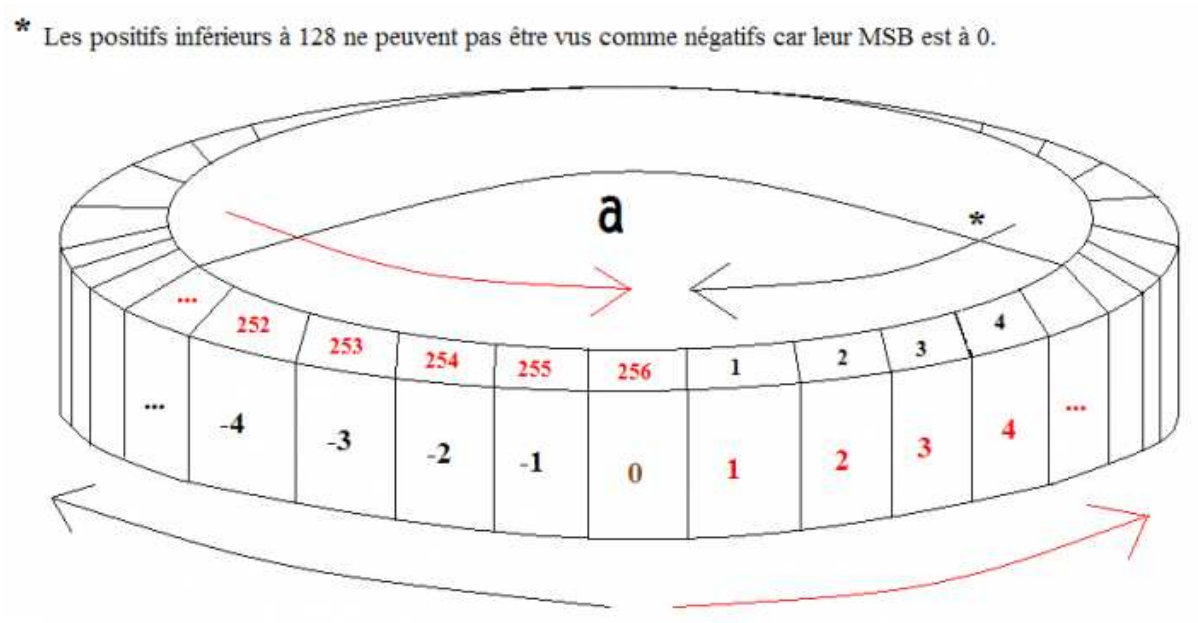
Et dans le cas contraire, alors ?
Eh bien soit un nombre est positif, soit il est négatif, n'est-ce pas ?!
Deux états possibles... On peut donc faire tenir cet état dans un seul bit.
L'heureux élu est le « most significant bit », le bit de poids fort, autrement dit celui qui se trouve à **l'extrême gauche**.
On a dans ce cas un nombre « **signé** ».
Évidemment, on a alors un bit disponible de moins.

Par exemple, un registre 8 bits signé pourra contenir des valeurs de -127 à 127.

Le bit de poids fort est à 1 dans le cas d'un nombre négatif, et à 0 dans le cas contraire.
Les bits suivants composent le nombre.

> Cette composition diffère entre les signed et les unsigned.

Pour vous aider, imaginez le registre **a** (pour l'exemple) composé de 255 cases correspondant aux valeurs possibles disposées en cercle. On pourra alors dire que 0 et 256 correspondent au même nombre.
Donc -1 correspond à 256-1, soit 255, et cætera.



Soit :

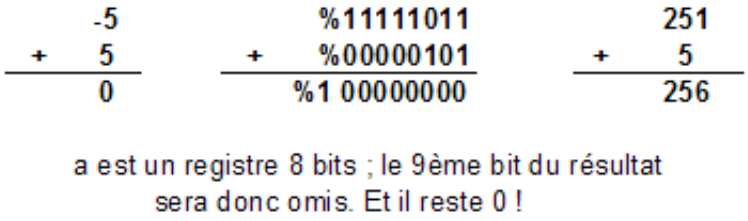
```
-1 = %11111111
120 = %01111000
-120 = %10001000 (ou 136...)
```

Pour changer le signe d'un nombre, vous devez changer l'état de tous ses bits puis lui ajouter 1. Ici se trouve l'importance de la taille du registre.
Sur 8 bits, le complément de %00101101 sera %11010010 alors que sur 16 bits, il sera %1111111111010010.
Faites-y bien attention !
Plus simplement, vous pouvez utiliser l'instruction **neg** qui change le signe de l'accumulateur.

Cette partie est assez déroutante mais après l'avoir lue plusieurs fois, le principe devrait rentrer ! :)

> Pour le processeur, le fait que le registre soit signed ou unsigned ne change absolument rien puisqu'il ne différencie pas les deux ! Seule votre décision influe sur le nombre. -12 ou 244 ? C'est à vous de choisir...

Si vous ne comprenez pas pourquoi, ce schéma devrait vous éclairer :



Les variables constituent un pilier pour la plupart des langages, et l'ASM z80 n'échappe pas à la règle !
N'hésitez pas à relire ce chapitre jusqu'à avoir bien tout compris car vous vous en servirez dans tous vos programmes.
Je considèrerai pour la suite qu'il est acquis.

Un peu de texte...

OK, c'est bien beau les cours théoriques, mais ça saoule, non ? :-°
Je pense qu'alterner avec des programmes ne vous lassera pas trop vite.
On va donc en créer un 2e, youpi ! ^^

Ressources nécessaires

Bon, en Basic, afficher du texte, c'était simple.
En Assembleur... c'est moins simple ! :p
Vous devez déjà savoir qu'il y a deux types d'écriture sur la calculatrice : la « grande » et la « petite » police.
Eh bien il existe une romcall pour afficher du texte dans chacune de ces polices.
Il s'agit respectivement de **_puts** ($450A) ou **_vputs** ($4561).
À votre avis, de quelles données ont besoin ces romcalls ?

Coordonnées

Il faut que la calculatrice sache où le texte doit être placé. Pour cela, elle utilise les variables **curcol** et **currow** (grande police) ou **pencol** et **penrow** (petite police).
« col » définit une colonne (-> l'abscisse), et « row » une ligne (-> l'ordonnée).
Voici leurs adresses :

- currow : **$844B** ;
- curcol : **$844C** ;
- pencol : **$86D7** ;
- penrow : **$86D8**.

Vous ne remarquez rien ? Les adresses se suivent deux à deux dans la RAM !
On pourra donc définir les deux coordonnées en une seule fois : pratique, non ? ;)

Texte

Le contraire m'aurait surpris !
Bah oui, il faut évidemment un texte à afficher.
Pour cela, vous devez charger dans **hl** le pointeur sur votre texte.

> Comment inclut-on le texte dans le programme ?

Vous vous souvenez peut-être du **.db** (chapitre du header) ?
En fait, toutes les données fixes dont vous aurez besoin au cours du programme doivent être « stockées » sous la forme **.db** [**donnée**] entre le « .end » et la fin du programme (généralement un **ret**), et repérées par un label. C'est ce label qui contiendra l'adresse de votre donnée ; c'est un pointeur ! ;)
Vous pouvez choisir le label que vous voulez, du moment qu'il vous rappelle votre donnée.
Comme ici, c'est une chaîne de caractères, on pourra utiliser du texte entre guillemets.

> Ne **SURTOUT pas oublier** un « **,0** » après la fermeture des guillemets ; c'est ce qui permet au programme de reconnaître la fin de la chaîne.

```
formule_de_politesse:
.db "bonjour !",0
```

Je reviens maintenant sur les coordonnées.
À votre niveau, vous avez juste besoin de savoir que l'assembleur sur la TI n'utilise qu'un écran qui sert à tout l'affichage ; il n'y a pas de distinction entre l'écran graphique et l'écran principal en Basic.
Il représente un tableau de 96 x 64 pixels, numérotés de haut en bas et de gauche à droite :

(0,0)	(0,1)	...	(0,94)	(0,95)
(1,0)	(1,1)	...	(1,94)	(1,95)
(2,0)	(2,1)	...	(2,94)	(2,95)
...	...	...	...	...
(62,0)	(62,1)	...	(62,94)	(62,95)
(63,0)	(63,1)	...	(63,94)	(63,95)

Retenez bien cette numérotation, car c'est elle qui est utilisée dans une majorité de cas.
Toutefois, les variables **curcol** et **currow** qui gèrent les grands caractères voient l'écran comme un tableau de 8 x 16 cases. Chaque case peut contenir un caractère.
Je crois que j'ai tout dit... On passe au programme ? ^^

« Hello world! », vous connaissez ?!

Le fameux « hello, world » ! Que de nostalgie ! :lol: .
Le principe du programme est simple : on efface l'écran, on affiche « HELLO, WORLD! » en grand au milieu et on stoppe le programme.

Je rappelle les notations des romcalls et leurs adresses :

- _clrlcdfull : $4540 ;
- _puts : $450A ;
- currow : $844B ;
- curcol : $844C.

À présent, c'est à vous !
Si vous avez bien suivi les chapitres précédents, cela ne devrait vous poser aucun problème ! ;)
Bon, comme c'est le début, je vous mâche un peu le travail ; pour être centré, ce texte doit être placé aux coordonnées (ligne : 3, colonne : 1).
Perso, j'obtiens ça :

```
#define bcall(xxxx) rst 28h \ .dw xxxx
_clrlcdfull .equ $4540
_puts .equ $450A
currow .equ $844B
.org $9D93
.db $BB,$6D
        bcall(_clrlcdfull)
        ld de,$0103
        ld (currow),de
        ld hl,texte
        bcall(_puts)
        ret
texte:
.db "HELLO, WORLD! ",0
.end
```

Encore une fois, les notations peuvent changer mais essayez d'utiliser les miennes ; ça vous servira plus tard... ^^
Remarquez l'espace après le point d'exclamation ; il sert simplement à éviter que le « Done » ne se superpose au texte.

Voilà... Ne vous inquiétez toujours pas pour le Done !

Maintenant que vous avez une forme de programme bien définie, essayez de changer le texte, de changer la police, la position...

Quelques erreurs peuvent subvenir sans que vous compreniez pourquoi ; voici les plus fréquentes.

Maximum number of args exceeded

C'est vrai, quand on ne sait pas quoi faire, ça énerve beaucoup !
Pour contrer ceci, séparez votre texte en plusieurs parties sous le même label.
Concrètement, ça donne ça :

```
Ne faites pas

texte:
.db "Ce texte est beaucoup trop long !",0

mais

texte:
.db "Ce texte est beaucoup trop lon"
.db "g !",0
```

Compris ?! :)

Unrecognized directive

Il est très probable que vous ayez oublié le **.db**. :p

```
texte:
"HELLO, WORLD!",0
```

Unrecognized argument

Vous avez tenté d'entrer une valeur dans **currow** sans passer par un registre.

```
ld (currow),$0103
```

> Si vous oubliez le « ,0 » à la fin d'un texte, vous risquez d'avoir des surprises même si le code est correctement compilé. En effet, tant que le compilateur ne trouvera pas un zéro, il considérera que les données des adresses suivant votre texte en font également partie... :-°

Je précise également que le Bloc-notes ne code pas tous les caractères comme TASM, ce qui peut entraîner des différences entre le texte du code et celui du programme. Par exemple, « **é** » ne s'affichera pas comme tel ; il faudra utiliser **150** à la place.
J'expliquerai pourquoi plus tard...

OK, un chapitre court, mais qui j'espère vous donne envie d'en faire plus !
L'écriture est tout de même fondamentale pour n'importe quel programme, alors entraînez-vous. ^^

Trucs utiles...

> OK, encore du cours théorique, mais cela simplifiera beaucoup vos prochains programmes, et vous pourrez les complexifier (paradoxal, non ?!).
> Bah oui, un programme linéaire qui affiche du texte n'est pas super utile ! :D
> Mais ne vous inquiétez pas, on apprend... :ange:

Flags

Attention... Avec les sauts, voilà un autre pilier de la programmation ! :)
Jusqu'à présent, vos programmes étaient linéaires, c'est-à-dire qu'ils se déroulaient tranquillement du début du code à sa fin.
Mais si vous voulez reproduire plusieurs fois une suite d'instructions, vous n'allez pas la taper chaque fois que vous en avez besoin.
Et si vous voulez exécuter telle commande seulement si votre variable vaut telle valeur, vous avez besoin des instructions de saut et des conditions. Ces conditions sont vérifiables grâce aux flags.

Flags du processeur

Ça vous rappelle quelque chose ? ;) Eh oui, le registre **f** !
Voici sa structure :

Structure du registre ***f***

Bit	7	6	5	4	3	2	1	0
Flag	S	Z	-	H	-	P/V	N	C

Les flags sont des **bits**, ils ne peuvent donc prendre que deux valeurs : 0 ou 1.
Voyons la signification de ceux-ci, propres au processeur.

S - Signe

Ce flag indique le signe de l'accumulateur ; il est mis à 1 lorsque l'accumulateur contient un nombre négatif, et à 0 quand ce nombre est positif.
En fait, si vous avez bien suivi, il contient le bit n°7 de l'accumulateur !

Z - Zéro

Simple ! Si a = 0, alors z = 1, et vice-versa. ^^

-

Les bits 5 et 3 ne sont pas utilisés.

H - Half-Carry

Il est seulement utilisé avec l'instruction DAA.
Vous pouvez l'oublier...

P/V - Parité / Dépassement

Un dépassement se produit lorsqu'une valeur dépasse en positif ou négatif la capacité d'un registre 8 bits signé (moins de -127 ou plus de 127).
Dans ce cas, le flag est mis à 1.
La parité représente simplement le bit de poids faible. Il sera donc à 1 si l'accumulateur contient un nombre impair et à 0 s'il contient un nombre pair.
La parité ou le dépassement est concerné en fonction de l'instruction utilisée.

N - Addition / Soustraction

Il s'utilise exclusivement avec DAA. Vous pouvez donc l'oublier aussi !

C - Carry

Il agit un peu comme le flag**P/V**, à la différence qu'il détecte le dépassement d'un « unsigned 8 bits » (moins de 0 ou plus de 255).
Il est mis à 1 en cas de dépassement.

Voilà, ça ne doit pas trop vous parler ; on verra des utilisations concrètes un peu plus tard.

Flags de la calculatrice

Vous vous rappelez le fameux « Done » qui survient à la fin de l'exécution d'un programme ?
Pour le renier (:lol:), il faut utiliser la commande **res doneprgm,(iy+doneflags)**.

Étudions cette commande...
res est le contraire de **set** ; ces instructions ne sont peut-être pas inconnues car elles sont très utilisées.
set s'utilise pour une mise à 1 et **res** (pour reset :)) pour une mise à 0.

Je suis déjà passé très rapidement sur **IY**, juste pour dire qu'il s'agissait d'un index de registre. En fait, vous pouvez l'utiliser comme un registre 16 bits standard mais son utilité n'est pas là, puisqu'il pointe à la base sur les flags !

> Sa valeur doit normalement être $89F0, souvenez-vous en si vous la changez.

Vous voyez le signe « + » ? Il sert pour l'**offset**.
En effet, iy pointe sur les flags. Il contient donc l'adresse du 1er octet contenant des flags. Pour utiliser les autres, on devra donc sauter un certain nombre d'octets dans la mémoire.

doneflag représente donc le nombre d'octets à sauter pour arriver à la bonne série de flags.
Ceux-ci ne représentent que des bits, perdus dans un octet. Le rôle de **doneprgm** est de « pointer » la position du bit dans l'octet.

> Encore une fois, j'utilise des notations mais le programme ne se compilera pas si vous n'affectez pas de valeur à **doneprgm** et **doneflags**.

Regardez d'ailleurs la « vraie » commande : `res 5,(iy+0)`

En clair, vous devrez ajouter dans vos programmes :

```
doneflags .equ 0
doneprgm .equ 5
```

Et

```
...
        res doneprgm,(iy+doneflags)
ret
```

Je précise que nous voyons ces instructions à l'occasion de l'étude des flags mais elles peuvent s'utiliser d'une autre manière.
Tout ce que nous avons, ce sont deux valeurs !
OK, il y a quand même des restrictions. :colere: La première valeur doit être un entier constant et la deuxième un registre 8 bits.

```
set/res [entier constant],[registre 8 bits]
```

ou

```
set/res [entier constant],([index de registre]+[entier constant])
```

Exemple :

```
ld a,1 ; a = %00000001
set 4,a ; a = %00010001
```

> Tout ça c'est bien beau mais comment je fais pour savoir si tel flag est actif ou non ?

Eh bien soit vous ne cherchez pas à savoir et vous le définissez dans tous les cas ; soit vous utilisez l'instruction **bit**.
Ses arguments sont exactement les mêmes que res et set, et elle retourne le bit dans l'accumulateur.

`bit 5,b` ou `bit 5,(iy+0)`

> Notez bien que le « +0 » n'est pas indispensable, l'important est d'avoir un index de registre entre parenthèses.

Sauts / Sauts conditionnels

Les labels

Vous en avez déjà utilisé... Ça vous revient ?
Pour repérer votre texte dans le chapitre précédent !
Eh bien on a fait le tour ; l'utilité du label est bien sûr de repérer.
En fait, votre programme est obligatoirement stocké dans la RAM pour être exécuté ; il peut donc être pointé. Le label est donc le pointeur d'un endroit du programme, et contient toujours une adresse sur 16 bits.
Encore une fois, c'est ce que vous en faites qui détermine son utilité ; rien ne vous empêche de stocker des variables aux adresses des labels (base du SMC), de l'utiliser pour repérer une donnée ou une partie du programme.
De toute façon, ça ne change rien à son fonctionnement !

```
ld a,(texte) ; a = 162
...
ret
texte:
.db 162,150,0
.end
```

```
#include "ultimate.inc"
start_prgm:
ret
.end
```

start_prgm contient l'adresse de ret et (start_prgm) correspond à la « traduction » en chiffres de ret.
Si vous ne comprenez pas ça, ce n'est pas bien grave ! ^^

> Un label entre deux parties du programme ne modifiera pas l'ordre d'exécution puisque celui-ci se déroule normalement du haut vers le bas.

Sauts et sauts conditionnels

Maintenant qu'on a repéré des endroits du programme, il faut des instructions pour y accéder.
On utilise **jp** ou **jr**.

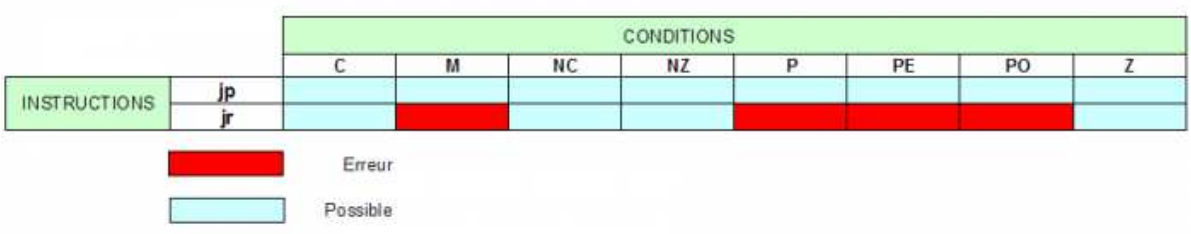
```
jp [label]
jr [label]
```

Leurs caractéristiques diffèrent sur leur taille et leur portée.
jp occupe un octet de plus que **jr**, mais permet d'accéder à n'importe quel endroit du programme, alors que jr ne peut « sauter » que de 128 octets vers le haut ou le bas (par comparaison, **jp** peut sauter 32 Ko, donc toute la RAM...).

Maintenant, voyons les conditions.
Vous avez peut-être essayé de tester les états des bits composant le registre **f** avec l'instruction **bit** ; et ça n'a pas marché !
Je rappelle que **f** est inaccessible en lecture comme en écriture (cependant, il l'est bit par bit en fonction de l'état des flags).
En fait, des notations dérivent de ces flags ; voici leur liste :

- **NZ** - non zéro ;
- **Z** - zéro ;
- **NC** - no carry ;
- **C** - carry ;
- **PO** - parity odd (impair) ;
- **PE** - parity even (pair) ;
- **P** - positif ;
- **M** - minus (négatif).

Ces notations se couplent aux instructions sous la forme `jp/jr [condition],[label]`
Cela signifie **aller à [label] si [condition] est vraie.**
Cependant, toutes les conditions ne peuvent pas s'associer à toutes les instructions...



On dispose également de l'instruction **djnz** qui s'utilise simplement avec la syntaxe `djnz [label]`
Ceci décrémente le registre **b** et va à [label] si **b** diffère de zéro. Dans le cas contraire, le code se poursuit comme s'il n'y avait pas cette instruction.
Très utile pour les boucles puisqu'il n'y a pas d'équivalent de For, ni de While, ni de Repeat... en ASM. :'(

Procédures

Ce terme désigne simplement un sous-programme, également appelé « routine ».
Le principe d'une routine est d'éviter la répétition d'une suite d'instructions sans perturber l'ordre d'exécution (en théorie !).

Elle est repérée par un label et se termine par un **ret**. Pour l'appeler, on utilise l'instruction **call** qui peut - au même titre que **ret** - être jointe à n'importe quelle condition.

```
call [condition],[label]
ret [condition]
```

```
...
call setvputs
...

setvputs:
ld (pencol),de
bcall(_vputs)
ret
```

Regardez cette routine ; lorsque le **call** est exécuté, il se passe la même chose que si les commandes de la routine - à l'exception du **ret** - étaient copiées à l'endroit de l'appel.
Autrement dit, ce code peut être remplacé par :

```
...
ld (pencol),de
bcall(_vputs)
...
```

Bien entendu, dans ce cas la routine n'a plus aucun intérêt, mais si vous devez utiliser plusieurs fois les commandes, la taille du programme se trouve réduite en utilisant un sous-programme.

> Mais alors **ret** ne sert pas à quitter le programme ?

Pas dans la cas d'une routine, où il sert à retourner dans le programme principal.
J'expliquerai comment et pourquoi dans un chapitre sur les optimisations, car des applications intéressantes peuvent en être tirées.

> Vous vous souvenez du stack ? Eh bien une procédure ne doit **jamais** retirer une valeur qui y était présente avant son appel.

Comparaison

Maintenant que vous connaissez les conditions, il ne reste plus que les instructions qui les déclenchent.
La plus utilisée est **cp** : `cp [valeur constante]` ou `cp [registre 8 bits]`.
Cette commande soustrait virtuellement [valeur constante] ou [registre 8 bits] à **a** et adapte les flags en fonction du résultat. L'accumulateur restera donc inchangé avant et après **cp**.

Exemple :
Si a>8
Alors aller à suite

```
cp 8
jp p,suite
```

Imaginons que **a** vaut 12 ; on lui soustrait 8.
Son résultat sera 4 ; donc le flagpositif sera activé, et **a** restera à 12.
Dans le cas général, le résultat sera positif si **a** est supérieur à 8. Compris ?! :)

Autres

La comparaison n'est pas la seule à affecter les flags ; en fait, la majorité des instructions agissent dessus. Par exemple, une addition avec l'accumulateur mettra P/V à 1 en cas de dépassement, P à 0 si le résultat est positif... etc.
Je pense mettre en annexe la liste des instructions du z80 avec leur description et leurs effets, mais ce n'est pas pour tout de suite !
De toute façon, **cp** convient dans la majorité des cas, et vous pouvez toujours me demander si vous avez un problème. :ange:

Les fichiers include

Vous vous demandez peut-être pourquoi je vous impose des notations de romcalls et d'adresses depuis le début alors que vous pouvez utiliser n'importe quel nom, du moment que vous connaissez l'adresse ?
Eh bien simplement car ces notations sont celles du fichier include que je possède et que je vous conseille également de posséder car je suivrai le cours avec.

Télécharger le fichier include (http://www.mediafire.com/download.php?blfix2x2gti)

C'était à la base le fichier ti83plus.inc, puis j'ai ajouté des trucs, des machins... Et finalement ça va beaucoup simplifier nos programmes !

Jetez un coup d'oeil, ce fichier est encore assez bien organisé !
Vous pouvez voir qu'il est découpé en plusieurs parties.

Defines

Le seul que vous avez utilisé à présent était **bcall(xxxx)** ou éventuellement pour des notations. Cette commande peut également s'associer à des conditions.
Pour l'instant, oubliez **bjump** et **errhandon**.

Zones libres

Eh oui, les saferam ! Pas grand-chose à dire ; relisez la partie Variables définies par l'utilisateur si vous avez oublié ce que c'est...

Ion functions / MirageOS subroutines

Si vous ne connaissez pas Ion ni MirageOS, ce n'est pas grave, vous n'aurez pas à vous servir de ces routines.

> Ce sont des **routines**, pas des romcalls. Elles seront appelées avec **call**.

System Variable Equates

Bah... Les équations des variables système !
J'en profite pour préciser que ce fichier include couvre une majorité - pour ne pas dire tout - des possibilités de programmation ; il est donc normal que vous ne compreniez pas certaines choses. :honte:
Mais j'essaierai de parler de tout.

Run indicators

Pas grand-chose à dire... Permet de gérer l'apparence de l'indicateur ; vous savez, la petite barre en haut à droite !

Macro to call base code from an app

Oubliez ça aussi sec !

> Bah pourquoi ?

Nous, on fait des programmes, pas des applications !
J'en parlerai peut-être si j'arrive à en compiler une, raaaaaaaaahhh. :colere:
Sinon, sachez que le langage est le même mais les commandes ne se gèrent pas toutes de la même manière (essayez de quitter une application avec **ret**, ha ha !!).

Common subroutine RST numbers

Je ne pense pas encore avoir tout compris, mais certaines romcalls peuvent être appelées avec l'instruction **rst** seule, ce qui implique un gain de rapidité.

Entry points

Aaaaaaaaaaaahhh !
Bah si ! C'est tout simplement la liste des romcalls !
Y en a un bon paquet, hein ?!
Bon, certaines sont explicites, d'autres incompréhensibles...
De toute façon j'expliquerai comment fonctionnent celles que j'utiliserai.
Vous pouvez regarder si vous voyez des trucs intéressants : ça motive !

System-defined RAM Variable Address Equates

Un nom pareil, ça ne s'oublie pas, si ?
Vous avez donc la localisation de toutes les variables utilisées par la calculatrice.
Je dois avouer que je n'en ai pas utilisé beaucoup jusqu'à présent, alors l'utilité de la plupart me reste inconnue...

Language localization equates

Je pense que vous devinez à quoi ça sert mais je ne sais **pas du tout** comment on l'utilise... :honte:

System and State Flags

Liste des flags, et effets pour des valeurs données. Pas mal de possibilités, vous pouvez les tester, normalement vous savez le faire !

Character Font equates

Vous vous rappelez l'accent aigu du chapitre sur le texte ? Il n'apparaissait pas car son codage était modifié.
Eh bien maintenant vous pourrez remplacer les caractères rebelles par leur « traduction » sur TI.
Si on regarde à « leAcute » (« e » accent aigu si vous n'avez pas compris), on trouve $96 qui équivaut à... 150 !
Preuve que je ne raconte pas de bêtises !

Keypress Equates

Liste des valeurs renvoyées par la romcall _getkey.
Je l'expliquerai dans le chapitre des entrées de l'utilisateur.

TI-83 Plus Context Equates

Des notations supplémentaires à la partie précédente...

Scan Code Equates

Encore une liste de valeurs retour d'une romcall ; il s'agit cette fois de _getcsc.
Même remarque que pour _getkey.

Tokens

Notion assez subtile ; j'y consacrerai une sous-partie. De toute façon nous n'en avons pas l'utilité pour l'instant.

Data Type Equates

Pour différencier les listes des matrices, ou les équations des programmes... la calculatrice accorde à chaque objet un octet « type ».
C'est cet octet qui détermine la nature de l'objet ; on a ici la liste des valeurs avec leur correspondance.

Parser Equates

Similaire à la liste précédente, mis à part qu'il s'agit ici des valeurs d'octets types des fonctions, genre fmin(, bal(...

I/O equates

I/O est l'abréviation de Input/Output, soit Entrée / sortie.
Ces octets servent pour la transmission et la réception de données entre calculatrices.

Interrupt equates/Memory paging equates/DEVICE CODES

Je ne sais pas encore à quoi ça sert... :'(

LCD Driver equates

Adresse des drivers permettant la communication directe avec l'écran LCD.

System Error Codes

Liste des identifiants des erreurs.

EQUATES TO RAM LOCATIONS FOR STAT VARS

C'est assez clair ; adresse mémoire des variables statistiques...

Vous aurez remarqué, les noms appartenant à une partie commencent généralement par la même lettre, ce qui permet de les identifier rapidement. Certaines notations, comme les touches, restent les mêmes.

Par exemple : yequ correspond à l'appui sur **Y=** et est précédée d'un **k** pour _getkey, d'un **sk** pour _getcsc... etc.

Remarquez également que les lignes

```
.org $9D93
.db $BB,$6D
```

sont présentes. En clair, après avoir inclus ce fichier au début de votre programme, vous n'avez plus qu'à écrire vos instructions !

Une dernière chose ; le .nolist du début, combiné au .list de la fin, permet de ne pas afficher tout le fichier include dans le document XLT. C'est quand même plus confortable !

OK, une partie assez barbante, mais maintenant vous devez avoir tout ce qu'il vous faut en matière d'adressage. C'est toujours ça dont je n'aurai plus à m'occuper. :lol:

Parlons de bits...

Le langage de base de n'importe quel processeur, c'est le binaire.
Si les instructions d'opérations décimales étaient assez rares et limitées, on peut agir beaucoup plus facilement sur les bits qui composent un octet.

Le langage de base de n'importe quel processeur, c'est le binaire.
Si les instructions d'opérations décimales étaient assez rares et limitées, on peut agir beaucoup plus facilement sur les bits qui composent un octet.

Opérateurs logiques

Ils étaient déjà présents en Basic et les revoilà !
Nous disposons en ASM des opérateurs **and**, **or** et **xor**. Le not n'est plus de la partie, du moins sous ce nom-là !
Ils agissent **tous** sur l'accumulateur.

Autrement dit, `xor b` effectuera l'opération **a xor b**.
b peut être remplacé par une variable.

Le résultat est toujours stocké dans **a**.

Comment agissent ces opérateurs ?
Vous connaissez peut-être les tables de vérité ? Il s'agit d'un tableau qui représente une fonction avec en entrée tous les états possibles pour chaque variable et en sortie, les états de la variable (généralement notée S) qui en résultent.

> Ici, le « b » peut être soit une valeur constante, soit un registre 8 bits.

and

a	b	Sortie
0	0	0
0	1	0
1	0	0
1	1	1

or

a	b	Sortie
0	0	0
0	1	1
1	0	1
1	1	1

xor

a	b	Sortie
0	0	0
0	1	1
1	0	1
1	1	0

À quoi ça sert ?
Bah pas à grand-chose, en fait. :honte:
Disons qu'ils sont là si vous en avez besoin...

Leur principale utilité est le masquage de bits de différentes façons.
Ainsi, pour mettre à 0 un certain nombre de bits, nous allons utiliser l'opérateur **and** avec une valeur binaire qui présentera des zéros aux mêmes bits que ceux qui doivent être masqués.
Comment ça, c'est pas clair ?!

Imaginons... Vous voulez masquer les 4 derniers bits en les mettant à zéro.

	%	1	1	0	1	0	0	1	1
and	%	1	1	1	1	0	0	0	0
		1	1	0	1	0	0	0	0

De même, pour masquer des bits à 1, il faudra utiliser **or** avec des bits à 1.
xor, quant à lui, est utile pour changer l'état d'un bit. Regardez sa table de vérité ; la sortie est à 1 quand les entrées sont différentes.
Par exemple, pour changer l'état de **a** en suivant le schéma logique (c'est-à-dire 0 ou 1), on utilisera **xor 1**.
Pourquoi ?

Si **a** est à 0, alors xor 1 renvoie 1 -> **a** change d'état.
De même, si **a** est à 1, alors xor 1 renvoie 0.
Compris ? :)

Allez, une autre utilité. Disons plutôt une optimisation : au lieu de **ld a,0**, utilisez **xor a**. Comme **a** = **a** (dingue o_O), la fonction renverra toujours zéro, quelle que soit la valeur de **a** avant l'instruction.

Une dernière chose, j'ai dit que not n'était pas disponible... Il faut utiliser **cpl** [**registre 8 bits**] qui retourne le complément de la valeur présente dans le registre dans ce même registre.

Ex. : cpl %11001011 = %00110100

> Notez bien que les instructions sous forme d'opérateurs sont parmi les plus rapides de l'ASM.
> N'hésitez donc pas à remplacer d'autres commandes par des opérateurs (dans la mesure du possible :-°).

Décalage / rotation

Ça ne sert pas tous les jours mais si vous en avez besoin, vous serez bien contents !
Un exemple ? Pour faire une routine de multiplication. Ça suit mieux déjà, non ? :lol:

> Ils ne fonctionnent que sur des variables ou des registres 8 bits.

SRL

Décalage logique à droite ; on déplace tous les bits d'un cran vers la droite.

> Ouais, et les bits 7 et 0, on en fait quoi ?

Mis à part le sens du décalage, c'est sur ce point que se distinguent les différentes instructions.
Ici, le bit 7 est mis à 0 et le bit 0 va dans le carry.
Donc si on a

```
#include "ultimate.inc"
ld a,%10010101
srl a ; a = %01001010
jp c,aff1
ret
aff1:
ld hl,text
ld de,0
bcall(_vputs)
ret
text:
.db "Le carry est à true",0
.end
```

<gras> la calculatrice affichera « Le carry est à true » et quittera le programme.
En décimal, cette commande revient à diviser par deux en arrondissant à l'entier inférieur.

Bon : si vous avez compris celui-là, je ne vais pas m'éterniser sur les autres !

SLL

Bon, elle a quelque chose de spécial... Pas définie dans TASM80.tab.
:o
Nan, revenez !
Pour l'utiliser, ajoutez

```
.addinstr    SLL (HL)    CB36 2 NOP 1
.addinstr    SLL (IX*)   DDCB 4 ZIX 1 3600
.addinstr    SLL (IY*)   FDCB 4 ZIX 1 3600
.addinstr    SLL A       CB37 2 NOP 1
.addinstr    SLL B       CB30 2 NOP 1
.addinstr    SLL C       CB31 2 NOP 1
.addinstr    SLL D       CB32 2 NOP 1
.addinstr    SLL E       CB33 2 NOP 1
.addinstr    SLL H       CB34 2 NOP 1
.addinstr    SLL L       CB35 2 NOP 1
```

<gras> au début de votre code dans le programme.

Là, on décale vers la gauche en mettant le bit 7 dans le carry et 1 dans le bit 0.
Conséquence, cela revient à multiplier par deux et à ajouter un.

SRA

Décalage arithmétique à droite.
Les bits se décalent d'un cran vers la droite, le bit 7 reste inchangé et le bit 0 va dans le carry.

```
ld a,%10011011
sra a ; a = %11001101 et carry à 1
```

Cela correspond à une division par deux d'un registre signé en arrondissant à la valeur inférieure.

SLA

Décalage arithmétique à gauche.
Le bit 7 va dans le carry et le bit 0 est mis à zéro.

Multiplication par deux standard, attention au dépassement !

RR

Rotation vers la droite.
Quand vous utilisez cette instruction, les bits sont décalés vers la droite, le bit 7 prend la valeur du carry présente à l'exécution de la commande, et ensuite le bit 0 est stocké dans le carry.

RL

Vous l'auriez deviné tout seuls, je parie !
OK, vous êtes des zéros. :p
Rotation à gauche ; décalage à gauche, le bit 0 prend la valeur du « vieux » carry, puis le bit 7 va se loger dans le carry.

RRC

Voilà une vraie rotation (vers la droite) !
Les bits sont décalés d'un cran vers la droite, le bit 7 et le carry prennent la valeur du bit 0 de la valeur d'origine.

RLC

La même, mais dans l'autre sens. C'est donc le bit 0 et le carry qui prennent la valeur à l'exécution du bit 7.

Il existe encore d'autres instructions de ce type, mais je préfère les passer sous silence !

Et les 16 bits ?

Y a quelques moyens, je n'explique rien ; vous réfléchirez calmement dessus !

```
srl d
rr e
```

```
sla b
rl l
```

* Juste pour montrer que les registres n'ont aucune importance.

Graphisme

Comme vous avez longtemps attendu ce chapitre, voici un petit quelque chose pour me faire pardonner. :ange:

compil.bat (http://www.mediafire.com/?sharekey=ac3f421d78db7adad2db6fb9a8902bda)

C'est un petit script qui remplace les deux que vous devriez avoir. Son fonctionnement est très simple : il suffit de le lancer, de renseigner le nom du fichier, et voilà !

J'en profite pour préciser qu'un nouveau passage concernant les linuxiens se trouve dans le chapitre Matériel nécessaire (http://www.siteduzero.com/tutoriel-3-15-0-materiel-necessaire.html). Un grand merci à **saimoun** qui s'en est occupé. :)

Bon, si vous lisez ceci c'est que vous voulez du graphisme, alors je ne vous fais pas plus mariner ! ^^

Alors... 96 x 64 pixels... Allumés ou éteints... Ça fait...
:waw: 4.26 x 10^(1820) possibilités !!

Même en noir et blanc, un programme doit avoir de beaux graphismes pour être attirant. Ça tombe bien, notre chère TI dispose des fonctions nécessaires pour toute réalisation !
Si on ajoute la rapidité de l'assembleur, des screens comme celui-ci peuvent être obtenus instantanément.

C'est pas beau, tout ça ?!
Allez, c'est parti ! :ninja:

L'écran graphique

(Re)présentation

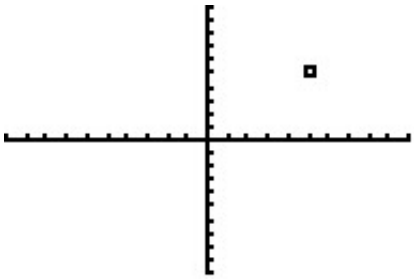
Vous vous souvenez (j'en suis sûr :)) que j'ai dit qu'il n'y avait qu'un écran en Assembleur ?
Eh ben en fait ce n'est pas vrai...

J'apporte donc une rectification ; il y a également deux écrans disponibles en Assembleur.
On dispose de l'écran que vous utilisez depuis le départ, et de **l'écran graphique standard** qui gère les tracés de courbes ou statistiques.

> Et c'est seulement maintenant que tu le dis ?!

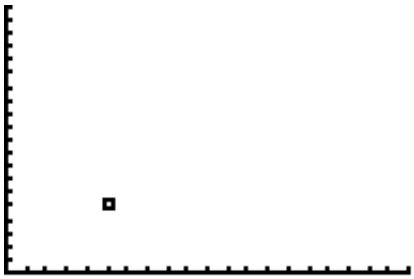
Oui, mais j'ai mes raisons. :p

L'utilisation en Assembleur de cet écran est assez limitée, autant pour la technique que pour l'usage.
Premièrement, le positionnement est **relatif**, c'est-à-dire que les coordonnées dépendent non pas de l'écran en lui-même mais du zoom actif au moment du dessin.
Regardez : les screens ci-dessous représentent tous les trois un point aux coordonnées (5,5)...



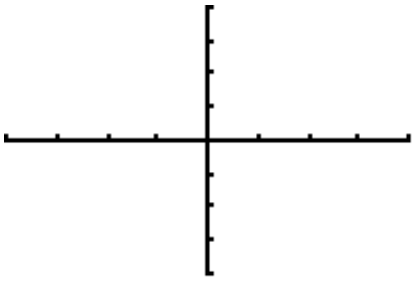
Zoom standard
XMin = -10
XMax = 10
YMin = -10
YMax = 10

Vous remarquerez que la position du point n'a rien à voir avec le pixel de mêmes coordonnées.



Ah ?! Là, ça change déjà ! La position du point change sur l'écran, mais pas par rapport au graphique courant. Si vous comptez avec le repère, le point se trouve bien à (5,5).

XMin = 0
XMax = 20
YMin = 0
YMax = 20



Ici... pas de point. Pourtant il est bien là, fidèle à ses coordonnées. J'illustre ici qu'un point dessiné sur l'écran graphique ne génère pas d'erreur de positionnement même s'il sort de l'écran.

XMin = -4
XMax = 4
YMin = -4
YMax = 4

Il faut donc être au courant du zoom avant le dessin pour être sûr du résultat. Dans le cas contraire, l'absence d'erreur de positionnement évite les plantages. :)

> Le **mode de l'écran** (Horiz ou G-T) influe également sur les valeurs du repère.

Deuxièmement, à part pour des programmes de maths ou de physique, l'utilisation des graphiques n'est pas courante ; et il vaut mieux utiliser le Basic pour les calculs !

OK, n'importe quel calcul peut être fait en Assembleur mais c'est parfois beaucoup plus long à coder du fait qu'on ne peut faire qu'un calcul par ligne entre autres. L'accès aux variables ajoute aussi pas mal de code.

Enfin, il me semble que ce chapitre est le bon endroit pour en parler. ;)

Bon, en fait je me rends compte que j'ai quasiment tout dit... :-°

Si, une dernière chose : cet écran est en partie géré par la calculatrice elle-même. Par exemple, quand vous voulez tracer une fonction, vous avez juste à rentrer son équation puis à demander son tracé, et la calculatrice s'occupe du reste.
En Assembleur, c'est la même chose ; en fait, hormis les techniques de dessin, l'écran graphique se gère à l'identique dans les deux langages.

Cela implique parfois des mises à jour des graphes, des vérifications de modes, l'activation des « plots » statistiques...

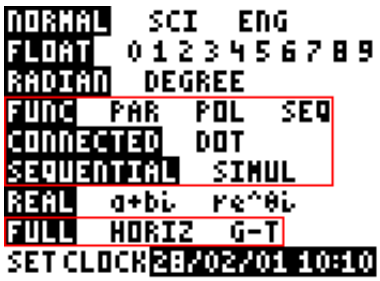
> Il n'y a pas d'erreur de positionnement, OK. Mais cela ne veut pas dire qu'aucune erreur ne peut être générée par l'écran graphique ; citons par exemple **WINDOW RANGE** et **DIM MISMATCH**.

En conclusion, **seule la méthode de dessin « direct » varie de l'Assembleur au Basic** (même si le Basic utilise l'Assembleur :lol:).

Flags

Voici la liste des flags attribués à l'écran graphique.

[MODE]



Nom du flag	Groupe correspondant	Description de la valeur booléenne
grffuncm	grfmodeflags	1 : Graph en mode fonction
grfpolarm	grfmodeflags	1 : Graph en mode polaire
grfparamm	grfmodeflags	1 : Graph en mode paramétrique

grfseqm	grfmodeflags	1 : Graph en mode séquentiel
grfdot	grfdbflags	0 : mode connecté 1 : mode point par point
grfsimul	grfdbflags	0 : Tracé de fonctions séquentiel 1 : Tracé de fonctions simultané
grfsplit	sgrflags	1 : Mode Horiz
vertsplit	sgrflags	1 : Mode G-T

Remarquez que lorsque seulement deux choix sont disponibles, on a qu'un seul flag puisque celui-ci peut prendre deux valeurs.

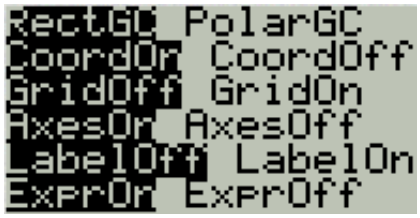
> Le partage d'écran ne se gère pas avec les flags ; ils servent ici uniquement d'information.
> Je n'aborderai pas la technique car elle est assez complexe ; pour l'instant, il est préférable de n'utiliser que le plein écran.
> Vous pouvez « forcer » ce réglage grâce à **_forcefullscreen**.

Pour ce qui est des autres réglages - quand ils concernent plus d'un flag -, il faut tout d'abord désactiver le flag courant et ensuite activer le flag voulu, ou l'inverse. L'important est de ne pas laisser deux flags activés (ni aucun).

```
set grfparamm,(iy+grfmodeflags)
        res grffuncm,(iy+grfmodeflags)
        res grfpolarm,(iy+grfmodeflags)
        res grfrecurm,(iy+grfmodeflags)
```

Ce code active le mode paramétrique ; comme vous pouvez le constater, le flag activé n'a pas besoin d'être connu.

[FORMAT]



Nom du flag	Groupe correspondant	Description de l'état
grfpolar	grfdbflags	0 : RectGC 1 : PolarGC
grfgrid	grfdbflags	0 : GridOff 1 : GridOn

grfnocoord	grfdbflags	0 : CoordOn 1 : CoordOff
grfnoaxis	grfdbflags	0 : AxesOn 1 : AxesOff
grflabel	grfdbflags	0 : LabelOff 1 : LabelOn

Il est apparemment impossible d'agir sur ExprOn...

Il reste deux flags « non visibles » intéressants :

- **graphdraw** du groupe **graphflags** qui sert d'indicateur. S'il est à 1, c'est que l'écran graphique doit être mis à jour ;
- **smartgraph_inv** du groupe **smartflags**. À 1, la calculatrice gère seule les mises à jour au besoin.

Les réglages restant ne se gérant pas simplement avec une romcall comme les limites d'affichage sont assez complexes et utilisent des techniques que nous n'avons pas encore vues.
Mais ne vous inquiétez pas, un autre chapitre sera consacré aux applications de l'écran graphique !

Les romcalls graphiques

Dans ce chapitre, nous allons voir les romcalls nécessaires à l'élaboration des graphismes.
Il existe des techniques de dessin sans romcalls, donc beaucoup plus rapides, mais elles sont plus complexes ; on les verra donc plus tard.

Les romcalls présentes permettent de tracer des **cercles**, des **rectangles**, des **lignes**, des **points** avec des options comme l'inversion de pixels, le remplissage...

Que du bonheur ! :D

Les points

> Il s'agira de points sur l'écran graphique et de **pixels** sur l'écran standard.

La base de tout graphisme...

_cpoint

Cette romcall permet d'allumer, d'éteindre ou d'inverser un point (sur l'écran graphique donc) selon la valeur de l'accumulateur.
S'il vaut :

- 0 : le point s'éteint ;
- 1 : le point s'allume ;
- 2 : le point s'inverse.

Notez que dans les deux premiers cas, l'état précédent du point n'influera pas sur le résultat.

Bien évidemment, la romcall a besoin des coordonnées du point ; celles-ci sont passées via les **OP** (Operating Points).

> Mais c'est quoi ça, les OP ?

Je l'expliquerai dans un chapitre qui leur sera consacré. Vous ne pouvez donc pas utiliser cette romcall pour le moment mais si elle est là, c'est pour que vous vous en rappeliez si jamais vous en avez besoin ; quand vous connaîtrez le fonctionnement des OP. ^^
Un code d'exemple est donc pour le moment inutile.

On utilise OP1 comme abscisse et OP2 comme ordonnée.

Cette romcall ne met pas **le graph** à jour et **détruit tout les registres**.

_darkpnt

Facile ! Cette romcall agit exactement comme la précédente, à la seule exception qu'elle ne permet pas de choisir l'état du point spécifié.
En effet, comme son nom l'indique, le point de coordonnées (OP1,OP2) sera allumé.
Tous les registres sont détruits et **le graph** n'est pas mis à jour...

_ipoint

Permet d'allumer, d'éteindre, d'inverser, de tester ou de copier un pixel (donc indépendant du zoom).
Les coordonnées du pixel sont passées par **b** (ligne/ordonnée) et **c** (colonne/ordonnée) et l'action à effectuer est représentée par **d** comme ci-dessous :

- 0 : éteint le pixel ;
- 1 : allume le pixel ;
- 2 : inverse le pixel ;
- 3 : teste le pixel ; son état est renvoyé dans le flag**Z** ;
- 4 : copie le pixel du graph buffer sur l'écran.

Le graph buffer sera vu plus tard, ne vous en préoccupez pas pour l'instant.

<tableau>
<ligne>
<cellule>

```
#include "ultimate.inc"
start:
        bcall(_clrlcdfull)
        ld bc,$2010 ; b=32
        ld d,1
        bcall(_ipoint)
        bcall(_getkey)
quit:
        ret
.end
```

</cellule>
<cellule>

</cellule>
</ligne>
</tableau>
bcall(_getkey) sert juste ici à figer l'affichage et permettre la capture d'écran ; cette romcall sera vue dans le chapitre suivant.

Aucun registre n'est détruit, **sauf** dans le cas du test où ils le sont tous.

_pointon

Elle est à _ipoint ce que _darkpnt est à _cpoint.
Explications ?! o_O
Allume le pixel de coordonnées (**b**,**c**) et... détruit le registre **d** !

Quelques remarques

Avant de continuer la liste des romcalls, je tiens à faire quelques précisions :

- qu'un registre soit « détruit » veut simplement dire que sa valeur après l'appel de la romcall n'est plus forcément la même qu'avant ; vous pouvez toujours l'utiliser !
- il existe un autre type de romcalls qui utilisent les FPS (First Person Shoot Floating Point Stack) mais comme vous ne savez pas ce que c'est et que leur utilité est assez limitée, je n'en parlerai pas ici ;
- l'écran graphique et l'écran standard sont en fait comme superposés ; vous verrez une modification sur n'importe quel écran. Toutefois, la mise à jour de l'écran graphique efface seulement les dessins de cet écran ; ceux de l'écran standard restent alors visibles ;
- en théorie, la ligne du bas (ordonnée 0) et la colonne de droite (abscisse 95) sont « intouchables », mais pas avec l'Assembleur ! Pour pouvoir dessiner dessus, utilisez **set fullscrndraw,(iy+apiflg4)** ;
- un pixel sera toujours plus rapide à utiliser qu'un point puisque le zoom ne sera pas pris en compte.

Lignes

Une ligne, c'est tout bonnement deux points reliés ; à part un point à ajouter on n'a donc pas vraiment de différences avec les romcalls précédentes... :-°

_cline

J'espère que vous remarquez les conventions de nom !
Cette romcall dessine une ligne sur l'écran graphique en utilisant les coordonnées de deux points passées via les OP :

- OP1 : abscisse du deuxième point ;
- OP2 : ordonnée du deuxième point ;
- OP3 : abscisse du premier point ;
- OP4 : ordonnée du premier point.

Notez qu'il existe 6 OP ! :D

Tous les registres sont détruits et le graph n'est pas mis à jour.

_darkline

Trace une ligne sur l'**écran standard** entre deux **pixels** dont les coordonnées sont passées par les **registres**.

- **b** : abscisse du premier pixel ;
- **c** : ordonnée du premier pixel ;
- **d** : abscisse du deuxième pixel ;
- **e** : ordonnée du deuxième pixel.

Tous les registres sont préservés.

_iline

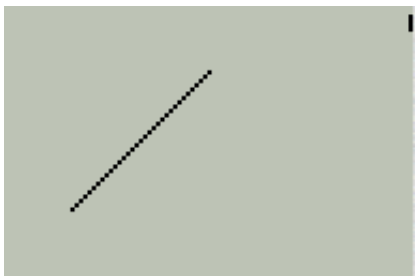
Agit exactement comme _darkline en ajoutant des options choisies avec **h** :

- 0 : éteint les pixels composant la ligne ;
- 1 : trace une ligne normalement ;
- 2 : inverse l'état des pixels composants la ligne.

<tableau>
<ligne>
<cellule>

```
#include "ultimate.inc"
start:
        bcall(_clrlcdfull)
        ld bc,$1010
        ld de,$3030
        bcall(_darkline)
        ; 1ère capture
        ld h,2
        bcall(_iline)
        bcall(_getkey)
quit:
        ret
.end
```

</cellule>
<cellule>



</cellule>
<cellule>

</cellule>
</ligne>
</tableau>
Je n'ai pas utilisé de _getkey pour la première capture car cette romcall détruit les registres.
Ici, on voit bien qu'ils ont été préservés car l'inversion efface la ligne précédente ; les coordonnées des pixels n'ont donc pas changé.

Cercles

Si on exclut les romcalls utilisant les FPS, il ne reste pas grand-chose... >_

_grphcirc

Trace un cercle sur l'écran standard en utilisant des variables prédéfinies :

- **curgx2** : abscisse du centre du cercle ;
- **curgy2** : ordonnée du centre du cercle ;
- **curgx** : abscisse d'un pixel du cercle ;
- **curgy** : ordonnée de ce pixel.

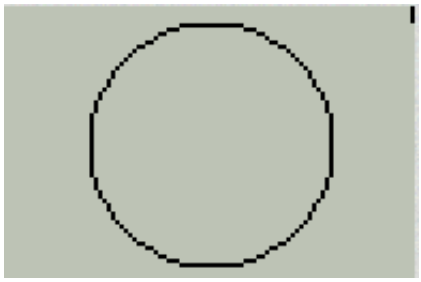
Détruit tous les registres.

> Vous pouvez accélérer le dessin d'un cercle avec la commande **set usefastcirc,(iy+plotflag3)** qui impose à la calculatrice de dessiner plusieurs sections du cercle simultanément.

<tableau>
<ligne>
<cellule>

```
#include "ultimate.inc"
start:
        bcall(_clrlcdfull)
        ld hl,$301F
        ld (curgy2),hl
        ld hl,$4433
        ld (curgy),hl
        bcall(_grphcirc)
        bcall(_getkey)
quit:
        ret
.end
```

</cellule>
<cellule>



</cellule>
</ligne>
</tableau>

_drawcirc2

Je donnerai juste un exemple tiré de la doc. officielle puisqu'il utilise les FPS ; c'est malheureusement le seul moyen de tracer un cercle sur l'écran graphique.

```
bcall(_zoodefault)  ; standard window
  bcall(_pdspgrph)  ; get current graph to the display
;
  bcall(_op1set0)   ; OP1 = 0
  rst  rpushrealo1
  rst  rpushrealo1  ; (0,0) pushed onto FPS
;
  bcall(_op1set3)  ; radius is 3
  rst  rpushrealo1  ; 3 pushed onto FPS
;
  bcall(_drawcirc2)
  ret
```

Pour info, on stocke le rayon du cercle dans FPST et les coordonnées du cercle dans FPS2 (abscisse) et FPS1 (ordonnée).
Si je dois parler des FPS, ce sera beaucoup plus tard, donc pour l'instant libre à vous de vous documenter ailleurs. ;)

Rectangles

On dispose de nombreuses possibilités pour leur tracé... mais seulement sur l'écran standard.

_clearrect

Efface une aire rectangulaire dont les coordonnées sont les suivantes :

- **h** : ligne du pixel supérieur gauche ;
- **l** : colonne du pixel supérieur gauche ;
- **d** : ligne du pixel inférieur droit ;
- **e** : colonne du pixel inférieur droit.

On doit logiquement avoir **d** >= **h** et **e** >= **l** sinon... plantage ! :(
Tous les registres sont détruits.

> Les coordonnées des points et la destruction des registres sont identiques pour toutes les romcalls de dessin de rectangle ; j'expliquerai donc maintenant seulement l'effet pour chaque romcall.

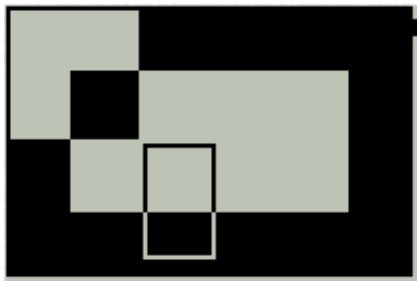
- **_drawrectborder** - Trace la bordure du rectangle.

- **_drawrectborderclear** - Trace la bordure du rectangle et efface son intérieur.
- **_eraserectborder** - Efface la bordure du rectangle.
- **_fillrect** - Dessine un rectangle plein.
- **_invertrect** - Inverse les pixels de la zone définie par le rectangle.

<tableau>
<ligne>
<cellule>

```
ld hl,0
        ld de,$3E5E
        bcall(_fillrect)
        ld hl,$0101
        ld de,$1E1E
        bcall(_clearrect)
        ld hl,$2020
        ld de,$3A30
        bcall(_eraserectborder)
        ld hl,$0F0F
        ld de,$2F4F
        bcall(_invertrect)
        bcall(_getkey)
```

</cellule>
<cellule>



</cellule>
</ligne>
</tableau>

Tests

_ibounds

Renvoie un booléen en fonction des coordonnées d'un point passées en paramètre(s).
Si ce point n'est pas affichable sur l'écran en fonction du zoom courant, renvoie **1** dans l'accumulateur, et **0** sinon.
Les registres sont préservés.

_iboundsfull

La même, mais inclut cette fois la ligne 63 et la colonne 95.
Le « plein écran » (fullscrndraw) doit être activé.

Écran graphique

Voilà maintenant les romcalls spécifiques à l'écran graphique qui agissent sur les tracés, le zoom, le type d'équation...

_alleq

Sélectionne ou désélectionne toutes les équations selon la valeur de l'accumulateur.
Si **a** = 3, sélection, et si **a** = 4, désélectionne.

Tous les registres sont détruits et OP1 et OP2 sont utilisés.

_pdspgrph

(Prononcez pour voir ?!! :D)
Cette romcall teste si l'écran graphique a besoin d'être mis à jour.
S'il l'est déjà, elle le copie dans le **graph buffer**. Dans le cas contraire, elle le met à jour (c'est-à-dire efface les dessins, trace les équations et les plots statistiques sélectionnées et éventuellement les axes) et l'affiche sur l'écran.

> Les données de l'écran ne sont pas effacées pour autant ; elles peuvent être recouvertes mais ne peuvent pas disparaître.

Tous les registres sont détruits.

Le graph buffer ? C'est la sous-partie suivante. ;)

_regraph

Agit comme la romcall précédente à l'exception que les équations sont toujours retracées même si le graph est déjà à jour.
L'écran graphique est donc également stocké dans le graph buffer.

Détruit tous les registres sauf **af**...

_setallplots

Sélectionne ou désélectionne tous les plots statistiques selon la valeur de **b**.
S'il vaut 1, alors ils sont tous sélectionnés, et inversement si **b**=0.

Si un changement de sélection survient, l'écran graphique est marqué comme « bon pour une mise à jour » !

Détruit tous les registres (pour changer... !). :D

_setfuncm, _setparm, _setpolm, _setseqm

Ces romcalls automatisent respectivement le passage des équations en mode fonction, paramétrique, polaire et séquentiel.
Détruit **a**, **bc**, **de** et **hl**.

Elles prennent moins de place dans le programme que l'utilisation des flags en cas de test, mais détruisent la quasi-totalité des registres ; à vous de choisir. ;)

_settblgraphdraw

Marque l'écran graphique ainsi que la table des valeurs pour une mise à jour complète au prochain affichage.
Si vous avez bien suivi, cela revient à activer un flag ; les registres sont donc préservés.

> En fait la table des valeurs possède elle aussi un « marquage » effectué par le flag**retable** du groupe **tblflags** qui est donc utilisé ici.

_zmdecml, _zmfit, _zmint, _zmprev, _zmsquare, _zmstats, _zmtrig, _zmusr, _zoodefault

Romcalls de zoom agissant comme dans le menu **ZOOM** de la calculatrice. Pas grand-chose à expliquer donc, excepté **_zmusr** qui rappelle le zoom enregistré par le dernier **zoomSto**.
Ah oui, et aussi **zmint** qui utilise respectivement OP1 et OP5 en tant qu'abscisse et ordonnée relatives du centre du graph...

Tous les registres sont détruits et l'écran graphique est marqué pour une mise à jour.

Affichage de données

Oui, on va pouvoir afficher des chiffres ! :D

_disphl

Vous pouvez vous en douter, cette romcall affiche le contenu de **hl**. :lol:
Elle utilise les grands caractères (et donc currow et curcol).

Comme **hl** contient à la base un nombre, c'est ce nombre qui est affiché.

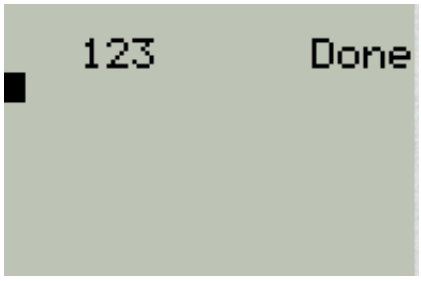
Détruit **af**, **de** et **hl**.

<tableau>
<ligne>
<cellule>

```
#include "ultimate.inc"
start:
        bcall(_clrlcdfull)
        ld hl,123
        ld de,$0101
        ld (currow),de
        bcall(_disphl)

quit:
        ret
.end
```

</cellule>
<cellule>



</cellule>
</ligne>
</tableau>

_dispop1a

Eh oui, encore un OP... ! :-°
Cette fois, il sert à afficher en petits caractères le contenu d'OP1 en chiffres.
On utilise donc penrow et pencol comme coordonnées.

Et **a** ?
Il contient le nombre maximum de chiffres à afficher !

Je ne m'étendrai pas sur cette romcall car le contenu des OP n'est pas forcément un nombre...

<tableau>
<ligne>
<cellule>

```
#include "ultimate.inc"
start:
        bcall(_clrlcdfull)
        bcall(_op1set1)
        ld a,2
        ld de,$0101
        ld (pencol),de
        bcall(_dispop1a)
quit:
        ret
.end
```

</cellule>
<cellule>

</cellule>
</ligne>
</tableau>

Ce code stocke le nombre 1 dans OP1 puis l'affiche.
Vous remarquerez que si la valeur dans **a** est supérieure au nombre de chiffres qui composent la valeur dans OP1, celle-ci est simplement affichée entièrement.

Détruit tous les registres.

Le graph buffer

Voici... le troisième écran. :p

Non, non, du calme !
Cet écran est **virtuel**, vous ne le verrez jamais directement.

Quel intérêt, me direz-vous, puisqu'on en a déjà deux ?!
Cet écran sert à dessiner des graphismes qui n'apparaîtront généralement que plus tard dans le programme.

> Moi, je ne vois toujours pas l'intérêt...

Les intérêts, auriez-vous dû dire. ^^

Mais une petite présentation s'impose avant.

Description du graph buffer

Concrètement, le graph buffer est une image de l'écran LCD placée dans la RAM.
Elle débute à l'adresse « plotsscreen ».
Comme notre TI ne gère pas la couleur, 1 bit suffit pour chaque pixel ; s'il vaut zéro le pixel est éteint et s'il vaut un, bah... je vous laisse deviner !

Par commodité, on le représente avec un tableau de... Allez, cherchez un peu. :p

On a au total 96 x 64 pixels, ce qui représente 6144 bits, soit **768 octets** avec 12 lignes et 8 colonnes.

Si vous connaissez la taille d'une variable Pic, vous pouvez noter le rapport !

Preuve finale, le fichier ultimate.inc :

```
plotsscreen          equ  9340h
seed1                equ  9640h
```

Vous voyez que 300h octets sont laissés à plotsscreen, et 300h ça fait... 768 !!

Voilà, une zone de 768 octets dans la RAM. C'est tout ce qu'est le graph buffer...
Rassurez-vous, il possède néanmoins un statut spécial, sinon je n'en aurais pas parlé. :lol:

Avantages

Au niveau des graphismes, on peut y utiliser n'importe quelle romcall de dessin, qu'elle soit pour l'écran standard ou graphique. Il suffit pour cela d'activer un flag mais j'en parlerai plus tard.
En plus de cela, le fait qu'il soit dans la RAM permet sa manipulation directe simplement et le tracé de motifs plus complexes. En effet vous aurez simplement à charger un nombre dans le bon emplacement pour dessiner sur le graph buffer sa représentation binaire.

Par exemple si vous faites

```
ld a,54
ld (plotsscreen),a
```

le premier octet du graph buffer (correspondant aux 8 premiers pixels de la première ligne de l'écran) ressemblera à ceci :

car 54 = %0110110.

> On travaille sur des **octets**, alors attention aux premiers bits !

De plus, le dessin de l'écran standard peut être remplacé en un clin d'oeil par celui du graph buffer. Ce dernier s'efface alors.

À la base, cette propriété s'utilise pour passer d'un dessin à l'autre le plus rapidement possible, ce qui évite que l'utilisateur soit gêné par le rafraîchissement de tous les pixels de l'écran.
Bien sûr notre résolution n'est pas énorme mais il faut en tenir compte !

Vous pouvez aussi utiliser le fait que les dessins faits sur le graph buffer sont invisibles pour y mettre des données que vous ne souhaitez pas forcément voir apparaître. Il vous suffira alors de copier le graph buffer à l'écran ou non en fonction de votre choix.

Romcalls

Le strict nécessaire !

_grbufclr

Efface le graph buffer ; cela revient à mettre tous les octets de plotsscreen à plotsscreen+767 à zéro.

_grbufcpy

Copie le graph buffer à l'écran.
L'écran initial n'influe pas puisque celui-ci est remplacé.
Le graph buffer**ne change pas** après l'appel de cette romcall.

Elles détruisent tous les registres.

Flags

Pour dessiner sur le graph buffer via des romcalls, c'est très simple : il suffit d'activer le flag :

- **textwrite**, de sgrflags pour le texte en petits caractères ;
- **bufferonly**, de plotflag3 pour les fonctions graphiques.

Pour le texte en grands caractères, il faudra repasser... ! :p

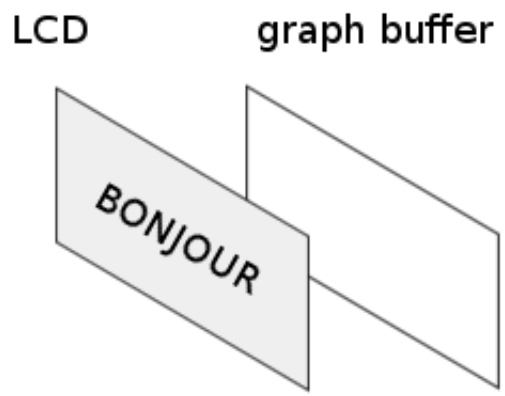
L'activation d'un de ces flags provoque le dessin des romcalls concernées uniquement sur le graph buffer.
Ces deux flags peuvent être activés simultanément.

> Si la désactivation de ces flags entraîne le tracé uniquement sur le LCD, vous pouvez choisir de dessiner sur le LCD **et** sur le graph buffer en même temps en activant le flag**plotloc** du groupe plotflags.

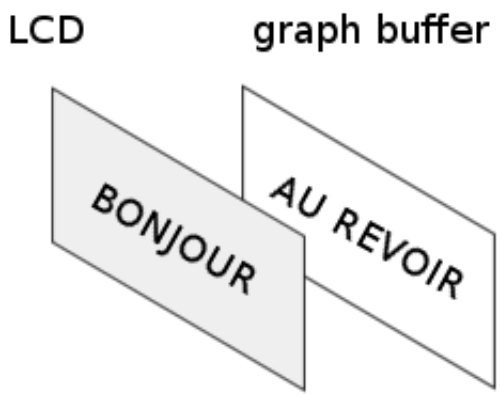
Exemple d'utilisation

Voici un schéma de l'utilisation générale du graph buffer couplé à un visuel et au code correspondant :

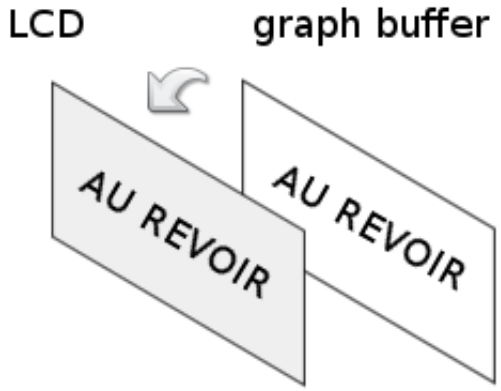
<tableau>
<ligne>
<cellule>



</cellule>
<cellule>



</cellule>
<cellule>



</cellule>
</ligne>
<ligne>
<cellule>On remplit le LCD...</cellule>
<cellule>puis le graph buffer...</cellule>
<cellule>et on copie !</cellule>
</ligne>
<ligne>
<cellule>

</cellule>
<cellule>

</cellule>
<cellule>

</cellule>
</ligne>
</tableau>
Voici le code exécutable correspondant :

```
#include "ultimate.inc"
start:
        bcall(_clrlcdfull)
        bcall(_grbufclr)
        ld de,$1010
        ld (pencol),de
        ld hl,texte
        bcall(_vputs)
        bcall(_getkey)
        set textwrite,(iy+sgrflags)
        ld hl,texte_buffer
        ld de,0
        ld (pencol),de
        bcall(_vputs)
        bcall(_getkey)
        bcall(_grbufcpy)
        bcall(_getkey)
        res textwrite,(iy+sgrflags)
quit:
        ret
texte:
.db "BONJOUR",0
texte_buffer:
.db "AU REVOIR",0
.end
```

Les deux textes sont placés à des coordonnées différentes pour bien visualiser que **la copie du graph buffer efface le LCD** ; le « BONJOUR » ne se trouve au final pas « sous » le « AU REVOIR » !

> Remarquez le **res textwrite,(iy+sgrflags)** à la fin du programme : la calculatrice garde les valeurs des flags que vous modifiez jusqu'à l'extinction.

Et alors... ?

Si vous enlevez cette ligne, textwrite sera toujours activé, même si vous relancez le programme. Vous voyez le problème ? Dans ce cas, le bonjour ne s'afficherait pas sur le LCD mais directement sur le graph buffer, et ne serait donc visible qu'à la troisième étape.

La calculatrice prévoit par défaut l'affichage sur le LCD uniquement, veillez donc :

- à **rétablir les valeurs par défaut** aux flags modifiés à la fin du programme ;
- à **activer / désactiver les flags voulus au début du programme** par mesure de précaution. Je ne le fais pas ici car n'ayant lancé que ce programme, je savais qu'il rétablissait le flag à sa valeur par défaut. Après c'est à vous de voir. ;)

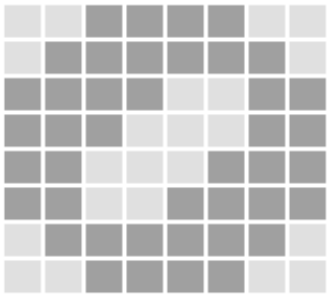
Bien, maintenant que vous connaissez les grandes lignes, passons à une application très intéressante du graph buffer, j'ai nommé... ^^

Les sprites

Nan, rien à voir avec une quelconque boisson gazeuse. :D

Les sprites sont dans notre cas de petites images mouvantes ou non placées sur le <itlaique>graph buffer</italique>.
Par exemple, prenons le jeu Pong ; la balle et les « raquettes » sont des sprites, pareil pour les pièces de Tetris, les croix ou les ronds d'un morpion...

Exemple en image :



Voici un sprite de 1 x 8 octets sur l'écran de la calculatrice.
Sur le graph buffer, l'état d'un pixel est représenté par l'état d'un bit.

On a donc ceci sur le graph buffer :

```
0 0 1 1 1 1 0 0
0 1 1 1 1 1 1 0
1 1 1 1 0 0 1 1
1 1 1 0 0 0 1 1
1 1 0 0 0 1 1 1
1 1 0 0 1 1 1 1
0 1 1 1 1 1 1 0
0 0 1 1 1 1 0 0
```

Ceci n'est pas très représentatif, mais si on enlève les zéros... on obtient ceci :

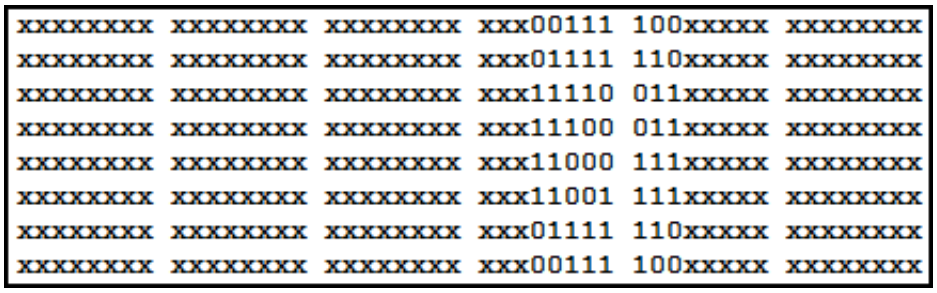
```
    1 1 1 1
  1 1 1 1 1 1
1 1 1 1     1 1
1 1 1       1 1
1 1       1 1 1
1 1     1 1 1 1
  1 1 1 1 1 1
    1 1 1 1
```

Comme quoi le binaire est parfois plus simple que le décimal. :-°

Vous me direz : eh ben c'est facile, il suffit de copier les bits du sprite sur le graph buffer !
En fait, ce n'est pas aussi simple que ça... >_

Le raisonnement est bon, mais n'oubliez pas qu'on compte en **octets**.
Cela implique que si on veut placer son sprite à un endroit qui ne correspond pas à un octet plein du graph buffer, on aura quelques problèmes !

Une 'tite image ne fera pas de mal :



Ici le sprite débute au 27e bit ; nous divisons par 8 avec **srl** pour connaître l'octet...
Et nous obtenons 3.

Le sprite commence en effet dans le troisième octet (n'oubliez pas que la numérotation commence à zéro).
Si on lance la routine sprite avec ces seules informations, notre sprite commencera donc à l'octet 3, c'est-à-dire au bit 24.
Et nous, on voulait le bit 27. :'(

Il faudra donc ajouter du code pour spécifier le bon bit. En l'occurrence, il faudra se servir des rotations et décalages.
C'est compliqué à comprendre les premières fois mais ne partez pas, ça vaut vraiment le coup ! ^^

Voici donc la routine sprite expliquée ; les « paramètres » sont :

- **a** : abscisse ;
- **l** : ordonnée ;
- **ix** : adresse du sprite ;
- **b** : nombre de lignes du sprite.

À noter que le sprite dans ce cas fera obligatoirement un octet de large.
Cette routine place dans le graph buffer un sprite de type « XORed », c'est-à-dire que si on a noir sur noir ou blanc sur noir, la couleur des pixels du sprite s'inversera.

```
sprite:
       ld e,l
       ld h,0
       ld d,h
       add hl,de
       add hl,de
       add hl,hl
       add hl,hl
       ld e,a
       and 7
       ld c,a
       srl e
       srl e
       srl e
       add hl,de
       ld de,plotsscreen
       add hl,de
```

Ce code permet d'obtenir l'adresse exacte du début du sprite.
Le principe est simple :

- on prend l'ordonnée du sprite ;
- on la multiplie par 12 (car une ligne du graph buffer fait 12 octets) pour avoir l'octet correspondant ;
- on réalise un modulo 8 sur l'abscisse pour trouver le nombre de bits à décaler. Ce nombre est stocké dans **c** ;
- on divise l'abscisse par 8 pour trouver la colonne de l'octet ;
- on additionne l'adresse de l'octet « ligne » avec celle de l'octet « colonne » ;
- et on additionne l'adresse précédente à celle du graph buffer.

On a maintenant l'adresse de l'octet où placer le sprite dans **hl**, et le nombre de bits de décalage dans **c**.

```
putSpriteLoop1:
       ld d,(ix)
       ld e,0
       ld a,c
       or a
       jr z,putSpriteSkip1
```

On stocke la première ligne du sprite dans **d** et on teste si le nombre de bits à décaler est nul ou non.

> Eh ! Ce n'est pas **cp** pour comparer ?

Dans le cas général, oui.
C'est d'ailleurs possible ici, mais or agit sur **a** et le flag**z**.
Comme **a**or**a** renvoie zéro seulement si **a**=0 (allez voir la table de vérité si vous ne vous souvenez plus :p) et que les opérateurs logiques sont les instructions les plus rapides, ce choix est entièrement justifié !

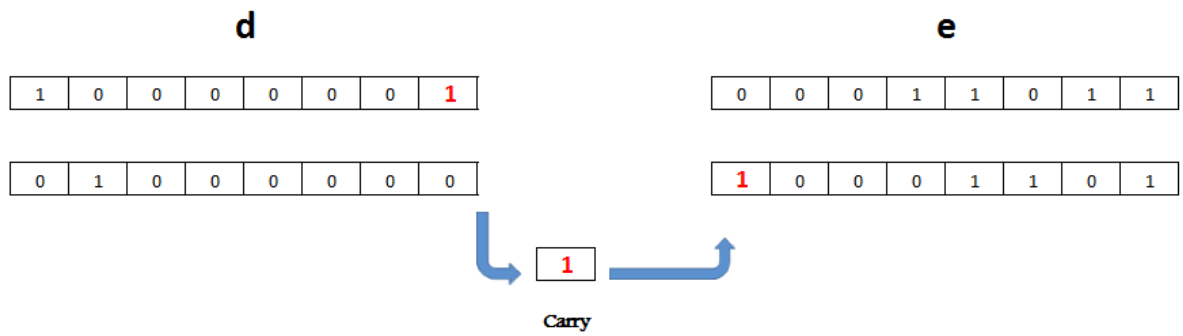
S'il est nul (non, pas le choix, l'accumulateur :D), c'est que l'adresse du premier bit (donc du MSB) de la ligne du sprite est aussi le MSB d'un octet du graph buffer. Il n'y a donc pas de décalage nécessaire.

```
putSpriteLoop2:
        srl d
        rr e
        dec a
        jr nz,putSpriteLoop2
```

Ce code est appelé en cas de décalage.

L'astuce est de faire passer la ligne du sprite sur deux octets qui seront alignés avec ceux du graph buffer. Ici on utilise pour cela le carry flag comme intermédiaire.

En effet, srl décale vers la droite et place le bit 0 dans le carry. Et que fait rr ?!
Il effectue une rotation et **utilise le carry en tant que valeur du bit 7 !**
En gros ça donne ça :



En utilisant **dec**, qui agit sur le flag**z**, on répète l'opération **a** fois.
On a donc au final la partie gauche du sprite dans **d**, et la droite dans **e** (et **a** à zéro, mais ça ne sert à rien de le savoir... :-°).

```
putSpriteSkip1:
        ld a,(hl)
        xor d
        ld (hl),a
        inc hl
        ld a,(hl)
        xor e
        ld (hl),a
        ld de,11
        add hl,de
        inc ix
        djnz putSpriteLoop1
        ret
```

Voici la boucle appelée directement s'il n'y a aucun décalage ; elle sert juste à placer la ligne du sprite dans le graph buffer.
Vous devriez tout comprendre si vous vous rappelez que les opérateurs logiques stockent leur résultat dans **a**. :D

Donc :

- on récupère l'octet courant du graph buffer dans **a** ;
- on place la partie gauche du sprite à cet endroit en utilisant xor ;
- on répète l'opération pour la partie droite à l'octet suivant ;

- on passe à l'octet suivant du graph buffer (on n'ajoute pas 12 car **hl** a déjà été incrémenté une fois) ;
- on passe à la ligne suivante du sprite...
- et c'est reparti !

Notez que djnz prend en compte le registre **b** qui contient le nombre de lignes du sprite.

OK, et maintenant... le code complet !!

```
sprite:
       ld e,l
       ld h,0
       ld d,h
       add hl,de
       add hl,de
       add hl,hl
       add hl,hl
       ld e,a
       and 7
       ld c,a
       srl e
       srl e
       srl e
       add hl,de
       ld de,plotSScreen
       add hl,de
putSpriteLoop1:
sl1:   ld d,(ix)
       ld e,0
       ld a,c
       or a
       jr z,putSpriteSkip1
putSpriteLoop2:
       srl d
       rr e
       dec a
       jr nz,putSpriteLoop2
putSpriteSkip1:
       ld a,(hl)
       xor d
       ld (hl),a
       inc hl
       ld a,(hl)
       xor e
       ld (hl),a
       ld de,11
       add hl,de
       inc ix
       djnz putSpriteLoop1
       ret
```

Pour la petite histoire, il existe d'autres routines sprite mais celle-ci est la plus optimisée (pas la plus simple car elle se valent toutes à ce niveau !).
D'ailleurs si vous connaissez, c'est cette routine qui est utilisée à l'appel de **isprite** sous **Ion**.

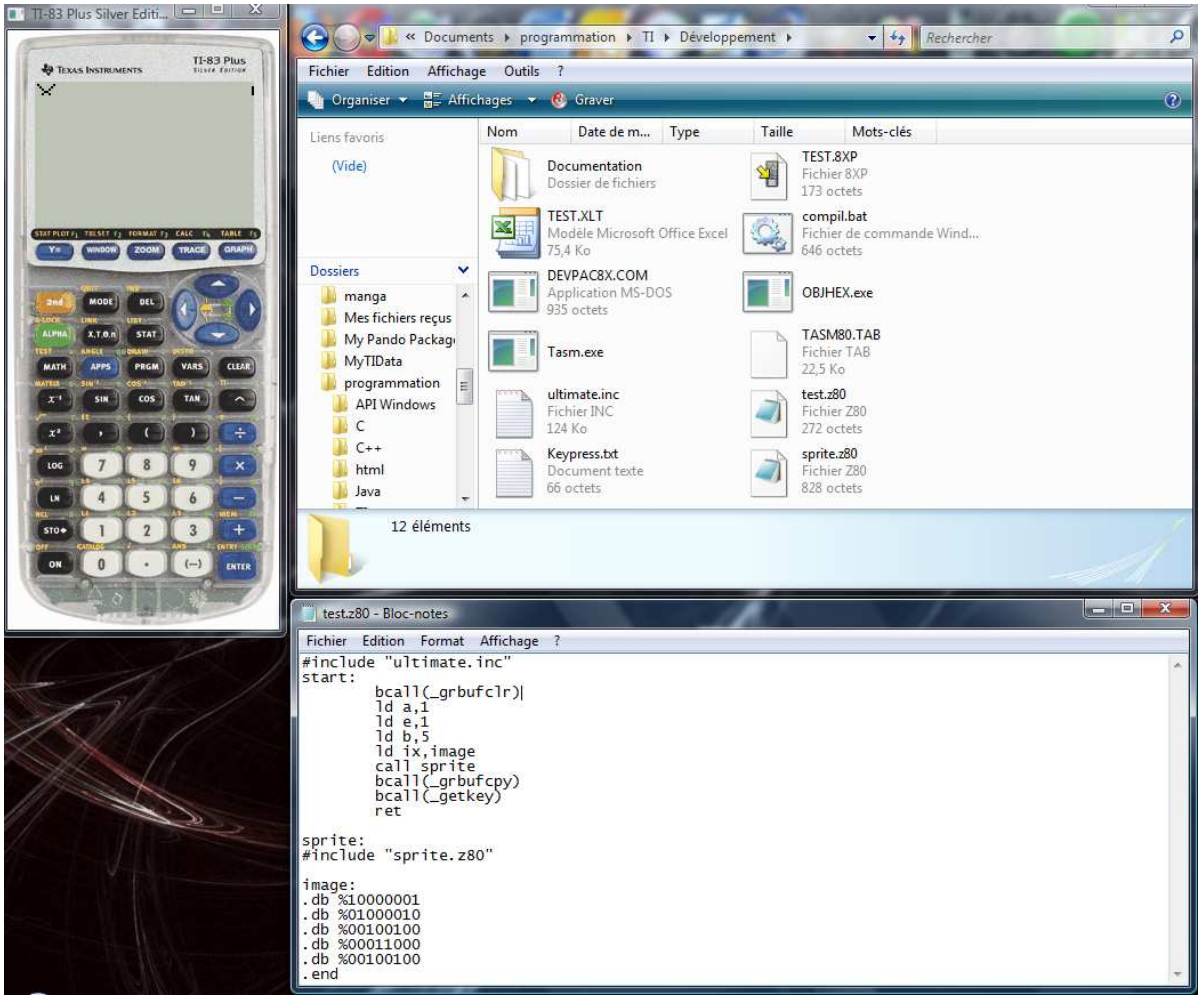
Pour clarifier votre code, placez ce code dans un fichier Z80 « sprite » (pour l'exemple) séparé et dans le même dossier que votre projet.
Quand vous aurez besoin de cette routine, utilisez

```
call putsprite
```

avec

```
putsprite :
        #include "sprite.z80"
```

Petit récapitulatif en image :



Pour mouvoir un sprite, le schéma est généralement le même : placement, affichage, effacement, puis on recommence !
Donc il reste à voir... l'effacement !!

Il existe deux solutions pour cela ; je n'en traiterai qu'une, la plus simple.
Le problème est d'effacer le sprite du graph buffer.

Notre sprite est de type XORed, donc...? Si on place à nouveau le même <itlaique>sprite</italique> au même endroit, les bits reprendront l'état qu'ils avaient avant l'affichage !
Donc pour effacer un sprite, il suffit de le **placer de nouveau au même endroit et de copier le graph buffer à l'écran.**

Bon, je crois que pour coder des programmes vraiment dynamiques, il n'y a plus qu'à gérer les actions de l'utilisateur. Parce que je crois qu'il s'impatiente, là ! :lol:
Eh bien c'est parti !

Pression des touches

> Allez, plus qu'un chapitre et vous posséderez toutes les bases nécessaires à la création de « vrais » programmes !

Les romcalls

C'est l'interaction la plus simple à gérer ; le but est de détecter l'appui sur une touche et tant qu'à faire de détecter quelle touche a été pressée !
Nous verrons ici des solutions passant par des romcalls. Elles sont donc à utiliser dans des programmes ne demandant pas une rapidité très importante, mais sont très pratiques pour la saisie de texte, pour déplacer le curseur d'un menu...

_getkey ($4972)

Si vous êtes des programmeurs Basic, vous ne pouvez pas être passés à côté de cette fonction qui renvoie un code correspondant à la touche pressée.
Eh bien en Assembleur, c'est quasiment la même chose !
Quand vous utilisez `bcall(_getkey)` le programme stoppe à cette instruction en attendant l'appui sur une touche, puis renvoie le code correspondant dans l'accumulateur.

Les codes se trouvent dans **ultimate.inc** à la section **keyboard key names**, mais ils sont faciles à comprendre ; il suffit d'ajouter « k » devant le nom de la touche. On a par exemple k0, kyequ (Y=), kleft... Mais aussi kquit, klinkio, kmatrixed... Eh oui, 2ND ou ALPHA peuvent être activés !

Lorsque le programme attend, vous pouvez donc :

- activer 2ND ou ALPHA ;
- changer le contraste ;
- éteindre la calculatrice.

Le problème si vous éteignez la calculatrice à ce moment, c'est que la RAM utilisée par le programme ne sera libérée qu'après un reset. Ce n'est pas forcément grave, mais ça implique une perte de mémoire. :colere2:
Pour contrer ceci vous pouvez utiliser **_getkeyretoff** ($500B) qui n'est pas dans le fichier include... Si on tente d'éteindre la calculatrice pendant l'exécution de cette romcall, la valeur **koff** est stockée dans l'accumulateur, et c'est tout !
L'inconvénient est que cette romcall agit sur l'OS, il faudra donc aller dans un menu avant de pouvoir éteindre la calculatrice !

D'ailleurs, lancez ce programme et essayez d'éteindre la calculatrice quand celle-ci attend la pression d'une touche :

```
#include "ultimate.inc"
start:
        bcall(_clrlcdfull)
        bcall($500B)
        ld l,a
        ld h,0
        ld de,0
        ld (currow),de
        bcall(_disphl)
quit:
        ret
.end
```

> Ne vous inquiétez pas si votre **Done** est un peu bizarre !

Comme vous avez dû le comprendre, la calculatrice ne s'éteindra pas, et 63 sera affiché. Pourquoi 63 ?!

Eh oui... $3F = 63 !

Et voilà, vous pouvez toujours faire « 2ND+ON », il faudra systématiquement faire un tour dans un menu pour pouvoir éteindre (ou « 2ND QUIT ») !
Encore un exemple sur les dangers de l'Assembleur... :-°

Il existe tout de même une solution pour éviter cela. Apparemment ce comportement serait dû à un flag sans nom et donc sans documentation. Toujours est-il que _getkeyretoff met à 1 le flag**7,(iy+40)**. Il suffira donc de le « resetter » avant la fin du programme pour pouvoir éteindre la calculatrice normalement...

Tant qu'on y est, parlons des flags !

- **indiconly, (iy+indicflags)** doit **impérativement** être à zéro ; sinon aucune pression ne sera détectée !
- **apdable, (iy+apdflags)** représente l'activation de l'APD. S'il est à un et que vous attendez à un getkey, la calculatrice finira par s'éteindre, mais l'exécution du programme reprendra à l'allumage.
- **lwrcaseactive, (iy+applwrcaseflag)** permet d'activer ou non l'écriture minuscule suite à deux pressions sur « ALPHA ». S'il est à un, les minuscules sont activées.
- **onrunning, (iy+onflags)** détermine le comportement du programme en cas d'appui sur « ON », si celui-ci ne génère pas un ERR:BREAK. S'il est à un, le programme continuera à tourner, et vice-versa.
- **oninterrupt, (iy+onflags)** est à 1 uniquement si « ON » a été pressé pendant le programme.

> Je n'ai pas élucidé le fait que l'appui sur « ON » génère ou non un ERR:BREAK puisque mon émulateur et ma calculatrice ne réagissaient pas de la même façon. Il vous faudra tester avant...

> Une fois que **oninterrupt** est actif, la calculatrice est bloquée, que **onrunning** soit actif ou non. Cela peut paraître bizarre mais dans notre cas, le programme attendrait indéfiniment au getkey. La solution consiste à désactiver **oninterrupt** lors de la détection d'une pression sur « ON » !

Pour exemple, voici la célèbre boucle getkey, qui affiche le code de la touche renvoyé par getkey jusqu'à appui sur « 2ND QUIT ». Ce programme est protégé (sous réserve de ne pas obtenir un ERR:BREAK...) contre l'appui sur « ON » et l'extinction, sans affecter le fonctionnement par défaut de la calculatrice !

```
#include "ultimate.inc"
_getkeyretoff .equ $500B
start:
        set onrunning,(iy+onflags)        ; assure l'exécution du programme
        bcall(_clrlcdfull)
loop:
        bcall(_getkeyretoff)
        cp 0                                                    ; teste si ON a ét
é pressé
        jr z,resetinterrupt                         ; si oui, on va désactiver l'interrup
tion
        cp kquit                                                ; sinon, on teste si 2N
D QUIT a été pressé...
        jr z,quit                                                ; ... et on quitte si o
ui
        ld l,a
        ld h,0                                                ; on stocke le code de l
a touche dans hl...
        ld de,0
        ld (currow),de
        bcall(_disphl)                                  ; ... et on l'affiche
        jr loop                                                  ; et on boucle !
quit:
        res 7,(iy+40)                                        ; on active l'extinction no
rmale
        ret
resetinterrupt:
        res oninterrupt, (iy+onflags)        ; on désactive le blocage suite à l'appui s
ur ON
        jr loop
.end
```

> L'utilisation de ce type de boucle nécessite obligatoirement une condition de sortie puisque ON n'interrompt pas le programme. Ou alors enlever les piles... !

Ici, si vous appuyez sur « ON » (code de touche : 0), rien ne changera à l'écran et le programme continuera de tourner normalement.

Une dernière chose : ces romcalls détruisent **bc**, **de** et **hl** !

_getcsc ($4018)

Cette fois, voilà le vrai « équivalent » de la fonction Basic ; cette romcall renvoie un code correspondant à la touche pressée, mais sans attente. Si aucune touche n'est détectée lors de l'exécution de la romcall ou si « ON » est pressé, celle-ci renvoie la valeur 0.

De plus, _getcsc est plus rapide que _getkey et détruit seulement **hl**.

En contrepartie l'APD n'est pas pris en charge, pas plus que l'activation de 2ND ou ALPHA, même si ces derniers peuvent être détectés.

Les codes renvoyés se trouvent dans ultimate.inc à la section **scan code equates**. Ils se reconnaissent à leur préfixe « sk ».
Remarquez qu'il y a beaucoup moins de codes que pour _getkey puisque l'appui sur « 2ND » ou « ALPHA » ne le modifie pas ; en revanche, il renverra **sk2nd** ou **skalpha** (cela implique qu'on ne pourra pas éteindre la calculatrice :lol:) !

Voici l'équivalent du code précédent, mais avec _getcsc ; le programme s'arrête à l'appui sur « ENTER ».

```
#include "ultimate.inc"
start:
        set onrunning, (iy+onflags)
        bcall(_clrlcdfull)
loop:
        bcall(_getcsc)
        bit oninterrupt, (iy+onflags)        ; on teste si ON a été pressé
        jr nz,onpressed
        cp 0                                                    ; on vérifie si aucu
ne touche n'a été pressée
        jr z,nokeypressed
        cp skenter                                      ; si appui sur ENTER...
        jr z,quit                                               ; ... on quitte
        ld l,a
        ld h,0
        ld de,0
        ld (currow),de
        bcall(_disphl)                                  ; on affiche le code de la touche
        jr loop
quit:
        ret
onpressed:
        res oninterrupt, (iy+onflags)        ; on désactive le blocage
nokeypressed:
        halt                                                    ; on passe en mod
e "low power" (pour ne pas afficher les 0)
        jr loop
.end
```

> Je suppose que vous avez remarqué le **halt**. Ne vous en préoccupez pas, il sert juste à visualiser le code de la touche pressée. Si vous l'enlevez, vous ne verrez que des zéros puisqu'il n'y a pas d'attente et que la boucle est très rapide !

Voilà, vous connaissez toutes les romcalls de saisie !
_getkey**_getkey s'utilise lorsque le programme ne s'intéresse qu'aux entrées de l'utilisateur tandis que _getcsc permet d'exécuter du code en attendant la pression des touches.**

Utilisation des ports : le direct input

C'est la méthode la plus rapide puisqu'elle n'utilise aucune romcall !...

Introduction aux ports

On peut définir un port comme un point de transit de données entre le processeur et l'un de ses « périphériques ».

Si vous avez du mal à comprendre, voici une belle métaphore :

Citation : Sean McLaughlin

> Imaginez que vous voulez aller de Yokohama à San Francisco en voiture ; c'est impossible car ces villes sont séparées par des kilomètres d'océan. Vous allez donc aller au port de Yokohama en voiture, et embarquer avec elle sur un bateau jusqu'au port de San Francisco. Une fois là-bas, vous pourrez de nouveau vous déplacer en voiture.

Vous et la voiture représentez les données envoyées, Yokohama le processeur et San Francisco le périphérique. Ils n'ont pas de réelle connexion et utilisent donc des ports pour interagir.

Il y a plusieurs périphériques sur notre TI comme l'écran, la connexion USB... et surtout (du moins ici !) le clavier !
Tous ces périphériques possèdent donc leur propre port grâce auquel on pourra envoyer et recevoir des voitures informations. o_O

Il est nécessaire de connaître les instructions suivantes :

Écriture

- **out ([entier]), a** envoie au port [entier] l'octet de l'accumulateur,
- **out (c), [registre 8 bits]** envoie au port **c** l'octet de [registre 8 bits] ;

Lecture

- **in a, ([entier])** stocke l'octet envoyé par le port [entier] dans l'accumulateur,
- **in [registre 8 bits], (c)** stocke l'octet envoyé par le port **c** dans [registre 8 bits].

Le port n°1 !

Exercice : une souris

Ce chapitre lié à celui des graphismes offre pas mal de possibilités, dont... (je suis sûr que vous y avez pensé :lol:) des jeux !!
C'est donc la création d'un jeu qui clôturera la deuxième partie de ce tuto !

Le cours n'est pas fini ; d'ailleurs je me demande s'il sera fini un jour tant il y a de possibilités ! ;)
