Projet:	Documentation sur Grammer (en Français)
Programme:	Grammer (1-page App)
Auteur:	Zeda Elnara (ThunderBolt)
Adaptateur:	Louis Becquey (persalteas)
E-mail:	xedaelnara@gmail.com (en anglais, svp...)
Version:	2.27.07.12
Dernière mise à jour:	10 Septembre 2012

I. Introduction et Installation

Le Grammer est un langage de programmation créé par Zeda Elnara. Il existait déjà d'autres langages tels que le TI Basic, le Celtic, l'Axe, ou l'assembleur. Pourtant ce langage est particulier car il est interprété de manière extrèmement rapide, parfois plus rapide que l'Axe !

Le Grammer a en fait été optimisé pour la vitesse et hormis l'application nécessaire pour interpréter les programmes, c'est langage merveilleusement pratique. Les programmes ne sont pas lourds, et ils peuvent être lancés en étant archivés.

Le Grammer est plus facile à apprendre que l'assembleur, le niveau doit être un peu plus difficile que le Celtic.

Un des points communs de Grammer avec le TI-Basic: c'est un langage interprété (et non compilé comme l'Axe ou

l'assembleur). Si l'interprétation ralenti le programme, les routines utilisés s'exécutent beaucoup plus rapidement.

Ce tuto est prévu pour montrer aux utilisateurs tout ce que Grammer peut faire de cool, et les meilleurs façons de l'exploiter, alors amusez-vous bien avec !

Vous aurez surement besoin de retourner en arrière, n'oubliez pas la combinaison de touches **Ctrl+F** sur un ordinateur, qui permet de rechercher un mot (une commande) dans le document !!

Envoyer et installer Grammer sur sa calculatrice :

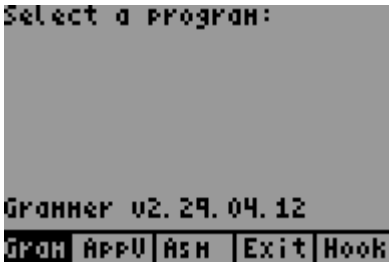
La première chose à faire est de se procurer l'application *Grammer.8xk*.
A cette date, Grammer est en développement, la dernière version évoluant souvent, je préfère vous donner un lien vers la page internet ou sont postées les mises à jour de Grammer plutôt qu'un lien de téléchargement direct.

Regardez donc sur cette page : http://www.omnimaga.org/index.php?board=199.0 et cherchez le lien de téléchargement de la plus récente version.

Il vous faut ensuite envoyer cette application à votre calculatrice, via un logiciel comme TI Connect. Vous pouvez trouver plusieurs tutoriels d'utilisation de TI Connect sur internet, j'en citerai deux:

Le mien: http://espace-ti.forumactif.com/t535-ti-connect-le-logiciel-officiel-ti-ordi
Celui d'un autre célèbre programmeur, kindermoumoute: http://www.siteduzero.com/tutoriel-3-425663-utilisation-de-ti-connect.html

Avec tout ça, vous devriez arriver à envoyer l'application à votre calculatrice.
Une fois qu'elle y est, lancez là.



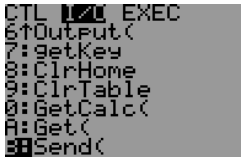
Ceci est l'écran d'accueil de l'application, à l'heure ou j'écris ce tutoriel.
(v2.29.04.12)

Vous pouvez choisir d'afficher la liste de programmes Grammer, la liste de programmes en assembleur-Grammer, ou la liste d'Appvars Grammer.

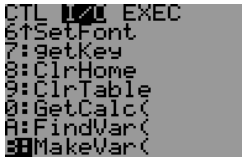
Le fait de démarrer l'application une seule fois installe Grammer sur la TI. Grammer sera désinstallé au premier "**RAM Cleared**".

Appuyez sur 

pour installer les Hooks de Grammer. (Cela remplacera le nom de certaines fonctions pour une meilleure compréhension des algorithmes Grammer, mais les programmes fonctionnent même si les Hooks ne sont pas activés.)



A gauche, le menu sans les Hooks, à droite, avec les Hooks.



Dans ce tutoriel, j'explique les fonctions en considérant que les Hooks sont activés.

Créer un programme Grammer :

Il y a deux choses à faire pour qu'un programme dans l'éditeur TI-Basic soit interprété comme un programme Grammer. Vous remarquerez que dans le cas contraire, les Hooks ne s'affichent pas, même si vous les avez activés.

Vous devez donc commencer votre programme par **.0:** et le terminer par un **Stop**.

(note: vous n'êtes pas obligés de terminer le programme avec, mais vous devez caser au moins un Stop dans le programme. Je pense particulièrement à ceux qui voudraient mettre des subroutines en fin de code. La commande **Stop** dit à Grammer que le programme est fini, donc elle arrêtera son éxécution, et la TI retournera à l'écran principal.)

A partir de là, vous pouvez tester toutes les instructions expliquées dans la première partie.

Dès que c'est fait, vous pouvez constater dans les menus que les Hooks sont apparus (si vous les avez activés).

Sommaire:

L'annotation (*) que je laisse de temps en temps signifie que je souhaiterais donner plus de précisions ou d'explications sur le fonctionnement de la commande dont on parle, mais que je ne le peux pas encore puisque j'ignore comment elle fonctionne réellement, ou qu'elle est en perpétuelle évolution (Grammer est toujours en dévellopement...)
Nous attendrons donc le futur.

_____ II. Fonctionnement d'une TI 83+/84+_____

C'est la partie énervante, où on va parler de trucs compliqués et pas intéressants, mais que vous êtes obligés de comprendre un minimum. Je suis tellement méchant avec vous que je commence par ça, et c'est 8 euros le paragraphe, messieurs-dames. (joke :P)
Si vous ne comprenez pas tout ? C'est pas grave. Pas du tout. Simplement, vous ferez un peu moins de choses avec

Grammer.

Les mémoires:

Une calculatrice TI 83 Plus ou TI 84 Plus, ainsi que les modèles dérivés, possède deux mémoires différentes.
La mémoire RAM, la plus utilisée dans les programmes puisqu'elle est modifiable rapidement, et la mémoire Archive, qui sert surtout à stocker des choses puisqu'elle est plus stable que la RAM.

La puce RAM de la calculatrice contient la mémoire RAM.
La puce ROM de la calculatrice contient la mémoire Archive, le Boot-Code, et l'OS de la calculatrice.

Les octets:

-Vous connaissez le principe d'un octet: Il s'agit de 8 chiffres binaires ou 2 chiffres hexadécimaux qui permettent d'enregistrer une valeur.
Dans le chapitre XVI de ce tutoriel se trouve un petit cours pour convertir les nombres entre leurs valeurs décimales ("normales"), hexadécimales (ou un octet tient en deux chiffres) et binaires (ou il n'y a que des 0 et des 1, et ou un octet fait donc 8 chiffres).

-La présence d'un **b** après un nombre composé de 0 et de 1 indique que c'est une valeur en binaire, la présence d'un **h** indique que c'est une valeur en hexadécimal.
petit exemple: 01011101**b** = 93 = 5D**h**

-La mémoire contient donc des octets. Comment s'y retrouve-t-elle dans ces octets pour savoir où elle a rangé telle ou telle information ? Eh bien chaque octet a une adresse selon sa position dans la mémoire. Par exemple, le 654e octet contenu dans la mémoire a l'adresse 654.
Cette fonctionnalité nous permet de demander à la calculatrice de nous dire la valeur présente à l'adresse que nous voulons (on appelle celà "lire la mémoire"), et nous pouvons aussi lui demander de ranger un nombre précis qui nous plait à une adresse précise (on appelle celà "écrire dans la mémoire").

-Les adresses des octets vont de 0 à 65535, après, plus de mémoire. (ça fait donc 65536 adresses en comptant le 0, on peut stocker 65536 octets dans une RAM de TI 83+/84+).

8 bits, 16 bits, 32 bits:

-On remarque que 1 octet est capable de stocker un nombre de 0 à 255. Après, il faut un deuxième octet.
Ainsi, on appelle les nombres de 0 à 255 des nombres 8 bits , ceux de 256 à 65535 des nombres 16 bits, et ceux d'après des nombres 32 bits (qui nécéssitent plus de 2 octets !).

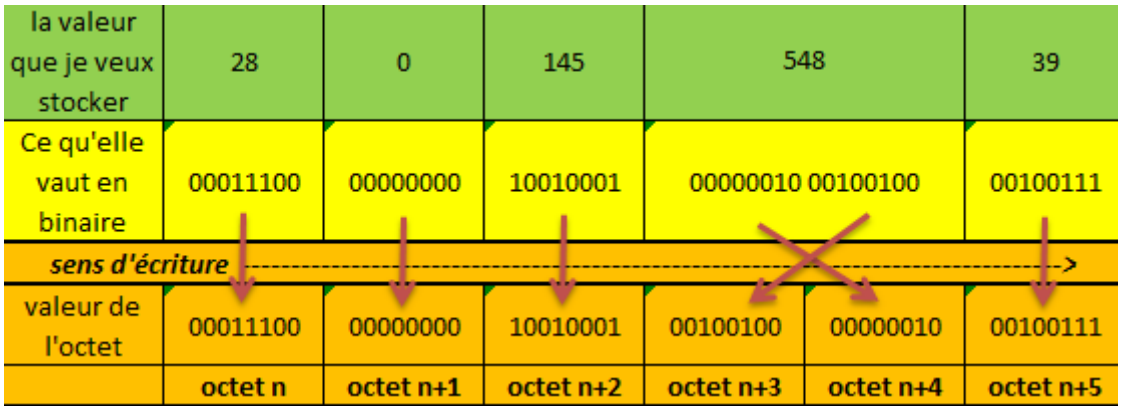
11111111b = 255 , le dernier nombre 8 bits
11111111 11111111b = 65535 , le dernier nombre 16 bits
11111111 11111111 11111111 11111111b = 2 147 483 647 , le dernier nombre 32 bits.
Les nombres 64 bits ne sont pas compris sur une TI, désolé pour vous !

-16 bits d'information constituent ce qu'on appelle un "**mot**", 32 bits un "**double mot**".

Attention ! Il est parfaitement possible de transformer un nombre 8 bits en 16 bits !
Tout simplement, le deuxième octet sera "vide" , c'est à dire plein de zéros.

00110101 (8 bits) = 0000000000110101 (16 bits)

-Là où ça se complique, c'est que les deux octets 00000000 et 00110101 seront enregistrés dans la RAM en commençant par **les valeurs les plus petites d'abord**. C'est toujours comme ça pour les nombres 16 bits et 32 bits, l'octet qui contient les valeurs les plus petites du nombre est enregistré d'abord. *Prenons un exemple. Je vais demander à la calculatrice d'enregistrer des nombres dans la mémoire, la suite de nombres 28,0,145,548,39, à l'adresse n.*
548 est un nombre 16 bits, qui sera stocké en deux octets, le premier valant 512 et le second valant 36 (512+36=548).
Eh bien, l'octet valant 36 sera enregistré avant celui valant 512.



Vous êtes donc d'accord que quand on utilise les octets de la RAM, vaut mieux savoir ce qu'on est censé trouver.
Si vous demandez à la TI: "**lis moi la valeur de l'octet à l'adresse n+3**" , elle vous dira **36**.
Si vous lui demandez: "**lis moi le nombre 16 bits à l'adresse n+3**" , elle vous dira **548**.
Si vous lui demandez: "**lis moi le caractère 16 bits enregistré à l'adresse n+3**", elle vous dira le caractère ayant pour numéro 548. (nous reverrons ça plus loin.)

Tout ça pour vous dire que si vous n'êtes pas précis avec votre TI, on s'emmêle vite les octets et ça devient un joyeux bordel.

Retour aux choses plus simples :)

Ensuite, les valeurs de ces octets stockés à différentes adresses peuvent être utilisées soit pour faire des calculs (ben oui, c'est des nombres, on peut leur faire faire des maths...) , soit à afficher du texte, c'est à dire une suite de caractères à l'écran.
Comment un nombre est il transformé en caractère ?
La calculatrice a en fait dans son système un tableau qui lui dit: tel nombre = tel caractère.

Certains de ces caractères sont désignés par un nombre 8 bits, d'autres par un 16 bits.
Je vais essayer de mettre ce tableau en annexe si jamais je le trouve en version copi-collable (ce qui n'est pas encore sur...)

_____ III. Vocabulaire - Définitions _____

un Token :

C'est le nom qu'on donne à tout symbole/caractère/fonction, mais pour désigner son "aspect graphique", et non pas sa fonction. Par exemple, on dira que pour avoir mis des Hooks dans l'application Grammer, Zeda a du modifier le token **Get(** en **FindVar(**.
Exemples de Tokens: **3** , **b** , **ClrDraw** , **log(** , **?** , etc...

Ans et Θ' :

Ans est une chose assez essentielle a comprendre. Ans est une sorte de variable, qui est modifiée automatiquement par la calculatrice au cours des opérations et des commandes.

Il faut comprendre que chaque expression, chaque commande, la plupart des lignes du programme modifient Ans.
Dans ce tutoriel, quand je dis que quelque chose "retourne" une valeur ou "donne" une valeur, c'est que cette chose fait prendre cette valeur à Ans.

Un calcul ou un test logique feront prendre à Ans la valeur du résultat.
La plupart des commandes (nous les verrons tout au long du tutoriel) "retourneront" quelque chose dans Ans, une adresse vers un octet, une valeur signifiant si il y a eu erreur ou si ça a réussi, etc...

Θ' est un complément de Ans. Souvent, il apporte une information en plus qui a besoin d'être précisée.
Il est utile dans les maths par exemple, pour dire si un nombre est en 16 bits ou en 32 bits. (nous le verront plus loin).

une Chaine :

Une Chaine est une suite de caractères (lettres, nombres, tokens, symboles...) qui peut être de n'importe quel type. Du texte, un nom de label (voir ci-après), le code d'un sprite, le code d'un programme...

Une chaine est introduite par un guillemet.

Exemples de chaines: **"BONJOUR** , **"01234567890ABCDEF** , **":If A=4:5-B**

une Variable de l'OS :

Ce sont les variables qu'on utilise en TI Basic, les variables de la calculatrice, utilisables même en dehors des programmes. On peut les utiliser avec Grammer, mais il faudra lui préciser que ce sont les variables de l'OS.

Les variables lettres-de-l'alphabet devront être précédées d'un "_" . Les programmes, Chaines, Images elles peuvent être utilisées avec certaines commandes d'accès à la mémoire, en entrant leur nom dans une chaine précédée d'une lettre marquant leur type.

Les barrés ne sont pas encore disponibles, ou susceptibles de faire tout planter. Alors on n'essaie rien avec les barrés.

Liste des lettres de type:

_ = Réel (alphabet)
~~A = Liste~~
~~B = Matrice~~
C = Equation
D = Appvar
E = Programme
F = Programme protégé
G = Image (Pic)
H = BDG
~~L = Complexe~~
~~M = liste complexe~~
N = indéterminé
~~O = Fenêtre~~
~~P = Zsto~~
~~Q = Table~~
~~R = LCD~~
~~S = Backup~~
~~T = Application~~
U = AppVar
V = Programme temporaire (XTEMP000 etc...)
W = Groupe

Pointeur et Variable de Pointeur :

Comme nous l'avons vu, toutes les données que vous utilisez dans un programme sont stockées dans la mémoire RAM à un endroit précis, sous forme d'octets. Ces octets, nous l'avons vu, peuvent se retrouver grâce à leur adresse.

Les adresses peuvent s'enregistrer dans ce qu'on appelle les "variables-pointeur". Ce sont les lettres de A à Z et Θ, ainsi que leur primes (A' à Z' et Θ') dans lesquelles on peut enregistrer des adresses de mémoire.
Si ça vous arrange, vous pouvez remplacer une variable avec une prime par sa minuscule. (A'=a ; B'=b ...)

Ces variables pointeurs sont prévues pour contenir **des nombres 16 bits**. Ainsi, vous allez stocker dedans des adresses 16 bits, les adresses entre 0 et 65535. C'est plutôt bien fait, puisqu'il n'y a pas d'autres adresses qu'entre 0 et 65535.

Pointeur, c'est le nom donnée à cette adresse lorsqu'elle est utilisée dans une variable pointeur.

Un **pointeur**, donc, est un nombre défini par sa **valeur** qui est entre 0 et 65535 (comme je vous l'avais dit), qui est enregistré dans une **variable-pointeur**, et qui "pointe" vers la RAM à l'**adresse** en question.

Alors comment enregistrer une adresse dans une variable pointeur ?
En utilisant la fonction "sto" (qui veut dire "stocker"). C'est une flèche.

→ sto stocke la dernière valeur obtenue (Ans) dans une variable.

La dernière valeur obtenue avant la flèche sera enregistrée dans la variable placée après la flêche.
La variable peut être une variable-pointeur, ou une variable de l'OS.

Les variables de l'OS lettres doivent être précédées d'un "_" pour ne pas les confondre avec les variables-pointeurs.
:**5→A** stocke l'adresse 5 dans la variable-pointeur A
:**5→_ A** stocke le nombre 5 dans la variable A de l'OS.

Techniquement, la flèche enregistre toujours Ans dans la variable: Le 5 modifie Ans (qui devient égal à 5), puis Ans est stocké dans la variable. Il serait donc aussi possible de faire: **:5**
:→A
On obtiendrait exactement la même chose.

-Nous avions dit que les variables-pointeurs étaient prévues pour être en 16 bits.
Lors des calculs, si vous devez stocker des résultats nécéssitant 32 bits (une grosse multiplication par exemple), il vous faudra préciser deux variables.
Ainsi, vous écrirez ***calcul*→AB** . Ans sera égal à B et Θ' sera égal à A.

-Il est également possible d'enregistrer en pointeur l'adresse d'une chaine:

Par exemple, pour la chaine de caractères "BONJOUR", vous ferez **"BONJOUR"→A** , l'adresse de la chaine "BONJOUR" sera enregistrée dans **A**. ça permet de retrouver l'endroit de la mémoire ou est enregistrée la chaine bonjour. A est une variable-pointeur qui contient le pointeur X (un nombre entre 0 et 65535), qui va dire à la calculatrice: "lis ce qui est écrit dans la mémoire à l'adresse X ".
Plus précisément, X est l'adresse ou commence la chaine "BONJOUR". Si vous demandez à la calculatrice "lis ce qui est écrit dans la mémoire à l'adresse **A+3**", elle vous répondra "JOUR". **A** étant une variable contenant un nombre, on peut faire des maths avec.

-Utiliser sto avec les chaines de l'OS:

"BONJOUR→Str1	enregistre la chaine "BONJOUR dans la variable Str1 de l'OS. (Str1 à Str256 sont disponibles, sachant que Str0 = Str10 et que Str00 = Str100)

un Sprite :

Un sprite est une petite image, souvent utilisée pour la déplacer à l'écran. En Grammer, les sprites peuvent mesurer jusqu'à 64 pixels de haut, ont une largeur qui doit être un multiple de 8 inférieur ou égal à 96. (en gros il peuvent faire 8 de large, ou 16, ou 24, 32, 40, 48, 56, 64, 72, 80, 88 ou encore 96 de large.)

un Argument :

On les appelle déjà comme ça dans les autres langages de progra, ce sont les informations qu'on donne à une commande pour qu'elle s'éxécute. Le plus souvent, ils sont séparés par des virgules, parfois collés.

Dans la suite du tutoriel, je vais présenter les commandes sous forme :

Commande(argument1 , argument2 , ... , argument x

Si certains de ces arguments sont entre [crochets], c'est qu'ils sont facultatifs et que vous n'êtes pas obligés de les préciser.

IV. Nombres et Maths

Le Grammer est un langage qui a été pensé pour la vitesse et pour les graphismes, pas pour les Maths. Il a donc son système de Maths un peu à part.

Particularités des Maths-Grammer:

-En Grammer, les nombres sont tous enregistrés par octets, donc les valeurs sont des entiers compris entre 0 et 65535.
C'est une des différences majeures avec le TI-Basic (et un point commun avec l'Axe).
Pas de négatifs, pas de nombres à virgule, pas de nombres supérieurs à 65535 car... il reviennent à zéro !

En effet, 65535+1 vous donnera le résultat 0 en Grammer, 65535+32 donnera 31...
Et inversement, ne vous étonnez pas si 1-4 ne vous donne pas -3 mais 65533.
C'est assez difficile de s'y habituer, mais il faut bien le retenir.

- En Grammer, la priorité des opérations n'est pas respectée. Les opérations se font donc de droite à gauche.
(et non pas gauche à droite !)
Par exemple, calculons 3*4+6/4-2 :

= 3*4+6/4-2
= 3*4+ 6 / **2**
= 3* 4 + **3**
= 3 * **7**
= **21**

- Ensuite, il y a l'astuce du Θ' pendant les opérations. Ans prendra la valeur du résultat, et Θ' est modifié pour apporter des informations supplémentaires.

+	lors d'une **addition**, Θ' vaudra 1 si le résultat dépasse 65535, sinon il vaudra 0.
-	lors d'une **soustraction**, Θ' vaudra 65535 si le résultat est inférieur à zéro (négatif), sinon il vaudra 0.
*	lors d'une **multiplication**, Θ' contiendra les 16 bits supérieurs si le résultat est de 32 bits, sinon 0.
/	lors d'une **division**, Θ' contiendra le reste de la division.
²	comme pour la multiplication, Θ' permet de re-composer le résultat 32 bits.

Optimisation:

- Après, je tiens à préciser qu'il n'est pas possible d'optimiser ses opérations en supprimant les symboles de multiplication:

11X+3Y+4 deviendra 11*X+3*Y+4.

- Enfin, quelque chose qui modifie pas mal la structure du code: Les parenthèses sont interdites, elles ont un autre rôle. Par exemple, si vous vouliez faire **If 24=((A*8)+(B*12)),** eh bien vous ferez:

```
:A*8
:+B*12
:If =24
```

ou même: **:If A*8:+B*12:=24** car les deux points, s'ils ne sont pas une "vraie" nouvelle ligne,

ne sont pas reconnus par le If.

voire même: :**If A*8 +B*12 =24** car vous pouvez remplacer le symbole deux points par un **espace**. (Seulement pour des calculs, pas sur une ligne avec une fonction utilisant des arguments.)

Vous gagnez en octets (moins de parenthèses) et en lisibilité !
Ces optimisations de maths dans les conditions peuvent se faire avec If, While et Repeat, qui ne voient pas tout les trois le token deux-points comme une nouvelle ligne.

La logique boléenne:

- Les opérateurs logiques sont utilisés pour comparer des valeurs entre elles. (> , = , < , and , or , etc...)

Par exemple: **A=B-5 or C>D+B**

= , ≠ , > , < , ≤ et ≥ Si l'expression est vraie, elle vaut 1, si elle est fausse, elle vaut 0. C'est souvent utilisé dans les conditions des If, While, Repeat etc.

or , and , xor Attention: **contrairement au TI-Basic**, les nombres seront comparés en tant que valeur **BINAIRES**.

Je rapelle le fonctionnement: En logique boléenne, toute valeur qui existe (différente de zéro) vaut 1. Les tests de logique de ces 3 commandes permettent donc de comparer entre elles deux valeurs toutes les deux comprises entre 0 et 1.
or: L'expression vaudra 1 si une des deux valeurs au moins vaut 1, sinon elle vaudra 0.
and: L'expression vaudra 1 si les deux valeurs vallent 1, sinon elle vaudra 0.
xor: L'expression vaudra 1 si les deux valeurs sont différentes, sinon elle vaudra 0.

exemple:	**5 xor 3**	**1)** *traduisons en binaire (b après le nombre)*
=	**101b xor 011b**	**2)** *On effectue le test pour chaque bit 1 par 1*
=	**110b**	**3)** *On re-traduit en décimal*
=	**6**	**4)** *Ainsi,* ***5 xor 3=6***

Bien entendu, les 1 et les 0 sont le plus souvent le résultat de tests avec = , > etc...
MAIS souvenez-vous: les maths se font **sans priorité, et de droite à gauche**.
Par exemple, si nous avons A=3, B=4 et C=1, et le test suivant:

A=3 and B=4 or C=1
A=3 and B=4 or 1
A=3 and B=5
A=3 and 0
A=0
0

Le résultat n'est pas 1 mais bien 0. Attention.

Inv(Inverse les 16 bits du nombre.

Inv(6 par exemple.
Inv(6 = Inv(00000000 00000110b =11111111 11111001b = 65529. (soit 65535-6)

De manière générale, on remarque que **Inv(** ***nombre*** = **65535-*nombre***.
Ne pas confondre avec le moins de la négativité (-) qui donne **6553*6*-*nombre.***

Les opérateurs mathématiques:

² , min(, max(, lcm(, gcd(, nCr fonctionnent comme en TI-Basic.

/ division

Le reste est stocké dans Θ' . Ajouter un espace après un / indiquera que vous utilisez des nombres négatifs:
par exemple: **65533/65535=0** **65533/ 65535 = -3/-1 = 3**

* multiplication

Si le résultat dépasse 16 bits, Θ' contiendra les 16 bits supérieurs et Ans les 16 suivants.

- soustraction

Si le résultat est négatif, la calculatrice lui ajoute 65536, et Θ' prend la valeur 65535.
par exemple, **3-6 = -3** s'affichera **65536-3=65533**.

+ addition

Si le nombre dépasse 65535, la calculatrice soustrait 65536 automatiquement, et Θ' prend la valeur 1.

√(racine carrée

Ajouter une apostrophe après le symbole racine carrée renverra un arrondi de la racine.

abs(valeur absolue

Donne la valeur absolue du nombre. Si le nombre est supérieur à 32768 (la moitié de 65536) Grammer considère que c'est un négatif, la fonction affichera donc 65536 moins le nombre en question.

sin(sinus

Donne le sinus d'un nombre. Il a une période de 256 (au lieu de 360), et renvoie un nombre entre -127 et 127.
L'angle droit correspond donc à 64 et non pas à 90.

cos(cosinus

Donne le cosinus d'un nombre. Il a une période de 256 (au lieu de 360), et renvoie un nombre entre -127 et 127.
L'angle droit correspond donc à 64 et non pas à 90.

rand valeur aléatoire

Génère un nombre aléatoire entre 0 et 65535.

randInt(entier aléatoire

Pour avoir un entier aléatoire entre X et Y, entrez **randInt(X,Y+1**

>lFactor facteur

Stockera le plus petit facteur de Ans dans Θ' puis le résultat de Ans/Θ' dans Ans.
Ainsi:

```
If Ans=Ans>Frac
Text(0,0,"Nombre Premier !
```

V. Parlons des Chaines

Les chaines sont des suites de caractères. Elles servent donc à enregistrer du texte, et à l'afficher. Les Chaines sont toujours introduites par un guillemet, vient ensuite le contenu, et parfois un guillemet de fin qui est souvent facultatif.

" guillemet introduit une chaine.

Indique que ce qui suit est une chaine. Tout ce qui suit sur la ligne fait partie de la chaine, sauf si un deuxième guillemet ou une flèche sto vient la refermer.
Il renvoie un pointeur vers la chaine demandée, qu'on peut stocker dans une variable-pointeur pour la réutiliser.

```
:"HELLO WORLD→A
:Text(0,0,A
```

ou plus simplement:

```
:Text(0,0,"HELLO WORLD
```

Les pointeurs étant supportés, il est donc aussi possible d'afficher des Chaines de l'OS (Str1, Str2...) en entrant leur adresse.

La commande Text a plusieurs options, un paragraphe lui est consacré pour son fonctionnement.

VI. Conditions If

If

Les If sont utilisés souvent dans les programmes, puisqu'il permettent au programme de ne faire quelque chose qu'à une (ou plusieurs) conditions.

Contrairement au reste du programme, le **If** ne lit pas le symbole deux-points "**:**" comme introduisant une nouvelle ligne. Il vous faut donc mettre l' (les) instruction(s) qui suit(vent) sur une VRAIE nouvelle ligne, en pressant [enter].

```
:If condition
: commande-à-exécuter
```

et si vous voulez mettre plusieurs choses à éxécuter:

```
:If condition
: commande-1 : commande-2 : etc...
```

Les commandes multiples sont alors séparées par deux-points mais PAS par une nouvelle ligne.
Par exemple:

```
:If A=3 ou C=2
:C+1→C:Horizontal C:"BONJOUR"→A
```

Pratique, non ? Si vous avez du mal à vous y habituer, le système avec les Then et End du TI-Basic fonctionne toujours, je vous rassure :)

VII. Les fonctions de boucle

Ce sont les même qu'en TI-Basic également, mais je les rapelle.

Les fonctions de boucle servent à répéter la même portion de code plusieurs fois de suite. Elles se composent donc d'une première ligne contenant leur type (Repeat, While, ou For) ainsi que les informations permettant de savoir quand

elle commence et quand elle s'arrête.

Viennent ensuite les lignes de code à répéter, il peut y en avoir autant qu'on veut.

Pour clore la portion de code à répéter, on finit par une ligne contenant la fonction **End**. A ne pas oublier.

While et Repeat, tout comme If, ne voient pas le token deux-points comme une nouvelle ligne, ce qui vous permet d'optimiser en remplaçant les parenthèses de votre condition (voir paragraphe III). Entrez la première ligne de la portion de code sur une VRAIE nouvelle ligne, en pressant [enter].

Repeat

Elle permet de répéter le code jusqu'à ce que la condition indiquée soit vraie. Si la condition est déjà vraie au départ, le code n'est éxécuté qu'une seule fois.

:Repeat *condition*
: *lignes de code*
: End

While

Elle permet de répéter le code tant que la condition indiquée est vraie. Si la condition est fausse dès le début, le code est sauté et n'est pas éxécuté du tout.

Le schéma est le même que pour Repeat, sauf qu'on remplace Repeat par While.

*On note donc la différence entre **While** et **Repeat: While** teste la condition au début de la boucle, **Repeat** à la fin..*

For

Elle permet de répéter le code avec une variable dont la valeur monte d'1 à chaque tour. On choisit la variable, on définit la valeur de début, la valeur de fin. Un exemple vaut mieux que de la théorie, alors voilà:

:For(A , 3 , 15
:lignes de code
: ...
: etc...
: End

Les lignes de code seront répétées plusieurs fois. La première fois, A vaudra 3, la 2e, A vaudra 4, à chaque tour, A va monter de 1 et le code sera répété, jusqu'à ce que A ait atteint 15, ou la boucle va s'arrêter.

For(*variable* **,** *valeur-de-début* **,** *valeur-de-fin*
: *lignes de code*
:End

Deuxième utilisation: Tout simplement, il est possible d'utiliser la fonction For(pour effectuer une portion de code un nombre exact de fois:

For(*nombre*
: *lignes de code*
:End

Ainsi, **For(8** éxécutera la portion de code 8 fois de suite.

VIII. Labels et code

Les Labels:

Un label est un repère dans le code, qui sert à dire "on est ici". Ils sont définis par un point suivi d'un nom.
Par exemple : **.DEBUT , .MENUPRINCIPAL , .CREDITS**
Il n'y a pas de limite au nom du label, mais souvenez vous qu'un label de 50 caractères pèse super-lourd et ralentit assez le programme. Un label de 5 caractères est rapide et précis.

Lbl

Lbl renvoie l'adresse d'un label. Par exemple, **Lbl "AB** va chercher le **.AB** dans le programme et donner l'adresse de ce label, qu'on peut ensuite stocker dans une variable pointeur.
On peut aussi choisir dans quel programme chercher le label (facultatif) : **Lbl "AB","TUTO** cherchera le .AB dans le programme prgmTUTO.

:Lbl "*nom-du-pointeur***", ["***nom-du-prog***"]**

On a pas besoin de mettre le point, mais il faut mettre des guillemets. (au moins avant le nom.)

A quoi ça sert, d'avoir l'adresse du label ? Soit à copier la portion de code suivant le Label, soit à demander au programme de retourner à ce label avec un Goto.

Goto

Goto interromp la lecture du programme par la calculette et la fait reprendre à l'endroit de la mémoire qu'on lui indique. Il faut donc le faire suivre d'un pointeur contenant l'adresse d'un label, par exemple.
Pour retourner au menu principal à la fin d'un programme:

:.MENUPRINCIPAL
: *code du menu principal*
:.PROGRAMME
: *code du programme*
:.FIN
:Lbl "MENUPRINCIPAL"→*A*
:Goto A

Ceci dit, les deux dernières lignes peuvent s'optimiser en ***:Goto Lbl "MENUPRINCIPAL"*** *si on ne veut pas passer par un pointeur.*
Ainsi, le programme qui lisait les lignes de code de haut en bas, voyant cette commande, va recommencer à lire depuis l'endroit où le **.MENUPRINCIPAL** a été posé.

Ensuite, il est possible de créer un "label temporaire" avec la fonction Return.

Return

Return permet de stocker dans une variable-pointeur l'adresse mémoire de la ligne de code suivante. Ainsi, vous pouvez retourner ensuite à cette ligne de code avec un Goto. Attention, il ne faut pas que le pointeur change de valeur entre temps !

:Return→*A*
:If getkey =/= 15 : Goto A

un moyen simple et efficace d'attendre que la toucher [CLEAR] soit pressée.

Pour sauter des lignes de code, vous pouvez aussi utiliser In(:

In(

Permet de sauter des lignes de code. Avec **In(4** , les 3 lignes de code suivantes seront ignorées.
(le In(4)+3 lignes suivantes font bien 4 lignes)
Avec **In(-9** , le programme revient 9 lignes en arrière.

Les subroutines:

Ce sont des portions de code que vous voulez utiliser souvent dans le programme, et que vous mettez à part au lieu de les recopier plein de fois bêtement.
Une subroutine s'écrit par exemple a la fin de votre code (ou au début), considérez la comme si c'était un deuxième programme, mais écrit dans le même que le premier. Elle commence par un Label et finit par un End.

call

Pour dire au programme principal d'exécuter la sub-routine, on utilise la commande **call**.

voilà un exemple. Le programme principal est en gris, la subroutine en rouge, à la fin du programme. On note que j'ai laissé un label **FIN** après la sub-routine, que l'on puisse sortir du programme en sautant par dessus la sub-routine.

```
:.PROGRAMMEPRINCIPAL
: commandes                    nous sommes donc dans un programme normal, avec des commandes.
: commandes
:Lbl "SUBROUTINE→A             // on stocke l'adresse mémoire de la subroutine "SUBROUTINE" dans le
pointeur A
:callA
:callA
: commandes                    // la subroutine est éxécutée à chaque fois que callA est exécuté
: commandes
:callA
: commandes
:callA
: commandes
:Goto Lbl "FIN                 //lorsque le programme en arrive là, il va sauter au Label "FIN" directement le code de la
:                              subroutine n'est pas obligé de se ré-exécuter.
:.SUBROUTINE
: commandes
: commandes
:End
:
:.FIN
:Stop                          // On aurait pu aussi mettre le Stop directement à la place de Goto Lbl "FIN, ce qui
                               aurait arrêté le programme avant qu'il atteigne la sub-routine.
```

Les interrupteurs

Si vous avez bien compris comment fonctionne une subroutine, lisez ceci. Je vais parler des interrupteurs en Grammer. Il s'agit de courtes sub-routines (n'essayez pas avec un trop long code !) , que le programme éxécutera de multiples fois tant que l'interrupteur est lancé.

Func Pour activer un interrupteur, utilisez **Func** , pour l'éteindre **Func** x **0**

```
:Lbl "SUBROUTINE→A
:commandes
:FuncA
:commandes                     pendant que ces commandes s'éxécutent, la subroutine s'éxécute aussi en même temps.
:FuncA0                        il est recommandé de ne pas prendre de code trop complexe.
:commandes
:Stop
:
:.SUBROUTINE
: commandes
:End
```

Les interrupteurs les plus utiles ne font qu'une ligne (sur qu'il fonctionnent), par exemple **DispGraph**, (pour mettre a jour souvent l'écran), **RunPart** ,(pour mettre à jour les particules automatiquement), ou encore l'incrémentation d'une variable pour faire un compteur de temps en même temps que le programme principal tourne.

Func *pointeur*[**0**] , [*vitesse*]

La vitesse d'actualisation est réglée par un nombre, plus il est petit, plus c'est rapide. La valeur par défaut (si rien n'est précisé est de 128, ça dure un peu moins d'une seconde.)

IX. Les Sprites

C'est avec ça qu'on fait de beau programmes graphiques. Ce sont de mini-images qu'on fait déplacer à l'écran, et qui permettent d'avoir de beaux décors de programmes rapidement.

Pour comprendre comment les sprites sont codés et enregistrés, on va devoir faire un peu de conversion binaire→hexadécimal. Vous savez que le binaire est la base 2, l'hexadécimal la base 16.

Imaginons un sprite représentant un cercle de 8 pixels sur 8 pixels.

les pixels allumés sont codés par un 1, ceux éteints par un zéro. Il se trouve que 00111100 est un nombre en binaire. On convertit ce nombre de sa valeur binaire en sa valeur hexadécimale, et ceci pour chaque ligne. Si vous ne savez pas faire de tête, de nombreux programmes le faisant existent déjà, je vous invite a les utiliser.

Ainsi, le code hexadécimal de notre sprite est 3C4281818181423C.

0 0 1 1 1 1 0 0 =3C
0 1 0 0 0 0 1 0 =42
1 0 0 0 0 0 0 1 =81
1 0 0 0 0 0 0 1 =81
1 0 0 0 0 0 0 1 =81
1 0 0 0 0 0 0 1 =81
0 1 0 0 0 0 1 0 =42
0 0 1 1 1 1 0 0 =3C

Pour utiliser cette valeur de 3C4281818181423C, qui est un trop grand nombre, on va l'enregistrer comme chaine de caractères le plus souvent. Ainsi, la chaine "3C4281818181423C" est le code de votre image de cercle.

Il est ensuite possible de compiler cette chaine hexa en assembleur pur (via la commande AsmComp).
Si vous utilisez des données de sprites en hexadécimal non-compilé, il vous faudra ajouter 8 à la méthode d'affichage du sprite lorsque vous la préciserez avec Sprite ou Tile, comme je vais le dire ci dessous.

L'affichage des sprites:

Les sprites, tout comme le texte ou n'importe quel graphisme, va s'afficher sur l'écran selon les coordonées qu'on lui donne. Les coordonées du sprite sont en fait ceux qu'on donne au pixel haut-gauche du sprite.

(Un sprite faisant plusieurs pixels de large et de haut, il ne peut avoir de coordonées comme un point. On utilise donc les coordonnées de son pixel en haut à gauche.)

Le repère sur l'écran, lui, est tout fait tout simplement par les numéros des pixels. L'origine se trouve sur le pixel en haut à gauche de l'écran (qui a les coordonnées 0,0). Ensuite, les abscisses augmentent de 1 par pixel vers la droite et les ordonnées de 1 par pixel vers le bas.

Il y a 5 mainères différentes d'afficher notre sprite sur l'écran. On peut l'afficher **par dessus l'écran** existant, ou avec des tests de logique.

Pour chaque pixel du sprite la méthode va déterminer comment le pixel s'affiche, en prenant en compte l'état de l'écran avant l'affichage de ce sprite.

Overwrite (par dessus l'écran) Le pixel est allumé ou éteint selon ce que dicte le code du sprite.

ET: Le pixel ne sera allumé après l'affichage du sprite que si le code du sprite le veut allumé et qu'il était déjà allumé avant. Dans tous les autres cas, le pixel sera éteint.

OU: Le pixel sera allumé soit si le code du sprite le veut allumé, soit si il était déjà allumé avant. Celà donne un effet de transparence: vous revoyez les 0 et 1 de votre cercle: avec la méthode OU, zéro ne veut plus dire "éteint" mais "transparent". Utile par exemple pour faire un curseur: le curseur ne fait pas 8x8 pixels, les zones du carré 8x8 inutilisées sont alors transparentes pour un meilleur confort visuel qu'un carré blanc.

OUExcl (XOR) : Le pixel ne sera allumé après l'affichage du sprite que si le code du sprite et l'état actuel de l'écran se

contredisent. Si ils sont d'accord, le pixel sera éteint. Utile pour faire de jolies animations (mais inutiles).

Effacer: Sur le code du sprite, imaginez que les 0 veulent dire "transparent" et que les 1 veulent dire "effacer en force". Tout les pixels ON sur le sprite seront OFF sur l'écran, mais ceux OFF sur le sprite ne changeront pas l'état de l'écran.

DataSwap: Elle permet de entre l'état actuel de l'écran et le code du sprite. Ainsi, elle permet d'afficher un sprite sans effacer ce qu'il y a derrière, qui se remet automatiquement lorsque le sprite est "swapé" une deuxième fois.

Masqué: Affichera un sprite masqué, c'est une technique de logique complexe qui fait que le sprite n'efface pas le décor quand il avance. Pour l'explication détaillée de la technique, voyez cet article (parlant du Xlib mais qu'on peut adapter): http://tibasicdev.wikidot.com/xlib:masked-sprites
Pour cela, vous devrez avoir le code de votre masque et le code de votre sprite.
exemple: "1234567890ABCDEF" le code du masque (au hasard)
"ABCDEF0123456789" le code du sprite (au hasard également)
Vous les mélangez en alternant les octets: 1 du masque, 1 du sprite, en commençant par le masque.
"12AB34CD56EF7801902 3AB45CD67EF89"
Vous obtenez ainsi le code du sprite-masqué. Le pointeur devra pointer sur ce code.

Gris: Pour dessiner un sprite en niveau de gris. Le principe est le même, sauf qu'au lieu d'alterner les octets d'un masque et d'un sprite, on va alterner les octets de 2 sprites. Le premier dont les pixels seront vus en gris et le deuxième aura donc ses pixels noirs.
Pour que ça marche bien, il faudra souvent mettre à jour l'écran, sinon on ne voit pas de gris.

Sprite(

Elle est plus lente que Tile, elle a moins de possiblités, mais elle permet d'afficher le sprite aux coordonées de son choix sur l'écran.

:Sprite(*méthode* **,** *pointeur* **,** *Y* **,** *X* **, 1 ,** *hauteur* **,** *[buffer]*

méthode: mettez 0 (overwrite), 1 (ET), 2 (XOR) , 3 (OU) , ou 5 (effacer). (rappel: ajoutez 8 si votre code est en hexa...)
pointeur: Une variable-pointeur qui pointe vers l'endroit ou est stockée le code du sprite dans la mémoire.
Y: l'ordonée du pixel du coin haut-gauche du sprite.
X: l'abscisse de ce même pixel.
1: Pour l'instant c'est 1. Il y aura peut être une autre option dans le futur. Discutez pas =P
hauteur: la hauteur du sprite en pixels.
buffer (facultatif) : le buffer sur lequel afficher le sprite. (voir paragraphe XI)

Tile(

La commande Tile est rapide, elle permet d'afficher un sprite sur l'écran à une abscisse multiple de 8. Il y a donc 12 "colonnes" de sprites qu'on peut afficher sur le même écran. L'autre avantage à part la vitesse, est d'afficher des sprites de largeur diverses. Il a aussi les autres options de méthode que Sprite n'autorise pas.

:Tile(*méthode* **,** *pointeur* **,** *Y* **,** *X* **,** *[largeur]* **,** *[hauteur]* **,** *[buffer]*

méthode: mettez 0 (overwrite), 1 (ET), 2 (XOR) , 3 (OU) , 4 (DataSwap), 5 (effacer), 6 (masqué), 7 (Gris). (même rappel...)
pointeur: Une variable-pointeur qui pointe vers l'endroit ou est stockée le code du sprite dans la mémoire.
Y: l'ordonée du pixel du coin haut-gauche du sprite.
X: l'abscisse de ce même pixel, qui correspond a son numéro de colonne (0 à 11).
largeur: La largeur du sprite en pixels, divisée par 8 (1=8 de large, 2=16 de large etc...), non précisé=8.
hauteur: la hauteur du sprite en pixels. Non précisé=8.
buffer (facultatif) : le buffer sur lequel afficher le sprite. (voir paragraphe XI)

X. La commande Text

Text(

Text permet d'afficher des chaines sur l'écran graphique de la TI. Une lettre faisant 6x4 pixels, on peut faire 24 colonnes sur l'écran d'une TI. *(plus d'infos: voir commande Fix)*
L'abscisse X du texte à préciser sera en fait le numéro de la colonne où elle doit apparaitre.
L'ordonnée Y en pixels, pas en lignes, est l'ordonnée du pixel haut-gauche de la 1ère lettre du texte.

Vous remarquerez que Grammer utilise sa propre police de caractères, et tous les lettres/symboles même l'espace font 4 pixels de large.

Pour afficher:

- du texte: **:Text(** *Y* **,** *X* **,"***Chaine* ou **:Text(** *Y* **,** *X* **,** *pointeur-vers-une-chaine*

- un nombre: **:Text('** *Y* **,** *X* **,** *nombre* **,** *[base-de-sortie]*
(notez l'apostrophe avant le Y)
La base de sortie permet d'afficher le nombre dans une autre base.
(vous allez savoir faire un convertisseur bin-hex !)

- un nombre 32 bits: il faut que la variable contenant les 16 derniers bits soit une prime. Example B et C':
:Text(' *Y* **,** *X* **,** *BC'*

- un caractère spécial: **:Text(** *Y* **,** *X* **,'** *numéro-du-caractère*
Il faut mettre une apostrophe suivie du numéro du caractère.

- une chaine en ASCII: **:Text(** *Y* **,** *X* **,° "** *chaine-ASCII*
Il faut mettre un degré avant la chaine.
par exemple, **Text(Y,X,°"HIrandM(WORLD** affichera **HI WORLD** car randM(correspond à l'espace dans le set ASCII.

options de la commande:

- écrire à la suite du précédent texte affiché: remplacer "**Y,X,**" par le symbole degré °
:Text(°"OKAY **:Text('°A**

- écrire à quelques pixels de différence : utiliser un + suivi du nombre de pixels.
:Text(+2,+5,"TEXTE DECALE **:Text(+10,,"COOL!**

- écrire lettre par lettre (avec un délai): utiliser **:/Text(** ou alors **:Text(ʳ**
:Text(ʳ°"OKAY **:Text(ʳ'°A,2**

le délai est modifiable avec la commande **Fix Text(** : plus le nombre précisé après FixText(est grand, plus le délai sera long. La valeur par défaut semble être 10. (Pour trouver **Fix**: [2nd][0][cos][v][v][enter])
Si vous utilisez une commande /Text puis des commandes Text(°, l'affichage lettre-par-lettre persiste. Il vous faudra reprécisez des coordonnées à la main pour arrêter l'affichage.

- Notez que la commande Text(supporte les retours à la ligne et quand elle arrive en bas de l'écran, elle revient en haut.

- Text(utilisée sans arguments renverra dans Ans la valeur de l'ordonée actuelle et dans Θ' l'abscisse actuelle.

XI. Les Graphismes

Il est bien entendu possible de dessiner autrement qu'avec des sprites. Il faut se souvenir que l'affichage de choses sur l'écran se fait en deux temps: l'enregistrement de ce qu'on va afficher sur l'écran quelque part dans la mémoire, la somme de plusieurs modifications de l'écran, tout ça, les "calculs" de pixels allumés ou éteints se font sur un "écran virtuel" qui est enregistré dans la mémoire et qu'on appelle un *buffer*.
La commande DispGraph affiche ensuite le buffer sur l'écran, appliquant ainsi les modifications. Les graphismes utilisant des niveaux de gris utilisent eux non pas un mais plusieurs buffers, qui s'affichent tour a tour.

Les buffers:
Un buffer est en fait une zone de la RAM ou on stocke un écran virtuel, sur lequel on peut ensuite dessiner, et qu'on peut effacer ou afficher sur le vrai écran. Un buffer fait 768 octets. (Comme une Pic, oui oui)

Comme un buffer est une zone de la RAM, il est donc défini par l'adresse de l'endroit ou il commence. On peut donc créer des pointeurs vers ces buffers. Ces pointeurs peuvent être utilisés par exemple pour dire à une commande de dessin de dessiner sur ce buffer là, et pas celui d'à coté. (vous verrez ensuite.)

Les adresses de buffers sont souvent exprimées en hexa (on pourrait les mettre en décimal, mais bon, c'est une tradition), et pour qu'on sache que le numéro est en hexa, on met un "pi" devant.

Nous utilisons 3 buffers courants, en Grammer. Mais vous pouvez très bien vous créer des buffers perso.

adresse= **π9872** : C'est le buffer que le moteur de particules utilise (vous verrez à la fin du paragraphe des graphismes). N'utilisez pas cette zone de la RAM si vous utilisez les particules aussi.

adresse= **π86EC** : Grammer utilise ce buffer pour certaines fonctions (pas forcément graphiques), comme convertir l'hexa en sprite ou éxécuter du code avec AsmPrgm (voir plus loin AsmPrgm).

adresse= **π9340** : C'est le buffer de l'écran. Celui par défaut, que l'OS utilise aussi, je pense.

Le buffer par défaut est celui qui sera utilisé avec les commandes graphiques qui vont suivre si aucun pointeur vers un buffer particulier n'est précisé. Un buffer est défini comme par défaut par la commande SetBuf.

SetBuf

Définit le buffer par défaut, en précisant l'adresse de ce buffer.

Pour les niveaux de gris, vous pourrez avoir besoin de **SetBuf °** qui définit le gray-buffer
SetBuf ' qui définit le black-buffer

:SetBuf [°]['] *adresse-ou-pointeur*

DispGraph

Affiche un buffer à l'écran graphique. Seule, elle affiche le buffer par défaut. Suivie d'une adresse ou d'un pointeur, elle affiche le buffer précisé par cette adresse.

:DispGraph *[adresse-ou-pointeur]*

ClrDraw

Efface le buffer. Réinitialise en même temps les coordonées du texte qui avaient été enregistrées

(avec Text). On peut effacer un buffer précis en entrant son pointeur après:

ClrDrawπ9872 par exemple.

Les commandes de dessin:

Circle(

Dessine un cercle à partir des coordonées X et Y du centre, ainsi que du rayon de ce cercle. Plusieurs options sont disponibles.

:Circle(*Y* **,** *X* **,** *rayon* **,** *[type]* **,** *[pattern]* **,** *[buffer]*

type : entrez ici 1 (cercle noir) , 2 (cercle blanc) , ou 3 (inversion des pixels du cercle). Si non précisé, ce sera noir.

pattern: mettez 0 pour le désactiver. Il permet de sauter des pixels dans le dessin du cercle. Par exemple 10101010 va le dessiner 1 pixel sur 2 (1 veut dire effacer). Vous pouvez convertir le nombre binaire en décimal (moins long), ou le laisser en binaire, mais précédé d'un E [2nd][,].
Les valeurs doivent être d'un octet seulement.

buffer: le pointeur du buffer sur lequel dessiner. Non précisé = buffer par défaut.

Exemple de code:

```
:.0:
:Repeat getKey(15
:A+1 ↓ A
:If A>20
:Then
:0 ↓ A
:ClrDraw
:End
:randInt(0,96 ↓ X
:randInt(0,64 ↓ Y
:randInt(0,64 ↓ R
:Circle(Y,X,R,3
:DispGraph
:End
:Stop
```

Rect('

Dessine une ligne sur l'écran entre deux points.

:Rect(' *X1* **,** *Y1* **,** *X2* **,** *Y2* **,** *[option][+option]* **,** *[buffer]*

X1 et Y1: Les coordonées de l'extrémité 1 du segment à tracer.

X2 et Y2: Les coordonées de l'extrémité 2 du segment à tracer.

option: 0 (ligne blanche) , 1 (ligne noire) , 2 (inversion des pixels de la ligne). Noir si non précisé.

buffer: le pointeur (ou l'adresse directe) du buffer sur lequel dessiner. Non précisé = buffer par défaut.

+option: Deux possibilités pour l'instant: +8 ou +4.
Bien entendu, il est possible d'optimiser 1+4 par 5, vous l'aurez compris.

+4: Vous pouvez alors rajouter des arguments après le buffer qui serviront de *pattern*.

Pour expliquer ce qu'est le pattern: il permet de dessiner la ligne en ignorant des pixels. exemple, si je veux que ma ligne se trace noire, en dessinant 4 pixels, puis en en ignorant deux, puis en en traçant 3, en en ignorant 1 (et après ça recommence), je vais faire:
:Rect('X1,Y1,X2,Y2,1+4,Z,4,2,3,1
Si on ne veut pas préciser Z (le buffer), on peut l'ignorer **(...Y2,1+4,,4,2...)**
J'obtiens ainsi un ligne en pointillés personalisés.

Attention, si vous ne mettez pas un nombre pair d'arguments, le processus s'inverse.

+8: (*)

astuces:

- Pour faire des pointillés réguliés, par exemple 2 px/2px, précisez juste 2 en argument:
:Rect('X1,Y1,X2,Y2,5,,2

- Si vous ne voulez votre pattern qu'une seule fois (que ça ne reparte pas au début), indiquez -2 comme dernier argument après vos arguments.
:Rect('X1,Y1,X2,Y2,5,,4,2,3,1,-2

Rect(

Dessine un rectangle sur l'écran. Il y a également des options de scrolling et de détection de pixels allumés dans la zone du rectangle ou sur sa bordure, qui elles ne tracent rien a l'écran.

:Rect(*X* **,** *Y* **,** *hauteur* **,** *largeur* **,** *option*

X: Abscisse du coin haut-gauche du rectangle (en pixels: 0-95).
Y: Ordonnée de ce même coin (en pixels: 0-63)
hauteur: La hauteur en pixels du rectangle (1-64).
largeur: La largeur en pixels du rectangle (1-96).
option:

0	rectangle plein blanc	7	rectangle inversé à bordure noire
1	rectangle plein noir	8	rectangle noir à bordure blanche
2	rectangle plein inversé	9	rectangle inversé à bordure blanche
3	rectangle vide noir	10	fait monter le contenu du rectangle d'1 pixel
4	rectangle vide blanc	11	fait descendre le contenu d'1 pixel.
5	rectangle vide inversé	14	Renvoie le nombre de pixels On de la zone
6	rectangle blanc a bordure noire	15	Idem, mais seulement pour la bordure de la zone.

Pxl-Off(- Pxl-Change(- Pxl-On(

Eteint - inverse - allume un pixel précis selon ses coordonées. Possibilité de le faire sur un buffer précis en rentrant son pointeur en dernier argument.

:Pxl-On(*Y* **,** *X* **,** *[buffer]*
:Pxl-Off(*Y* **,** *X* **,** *[buffer]*
:Pxl-Change(*Y* **,** *X* **,** *[buffer]*

ClrHome

Efface le buffer de l'écran home. Réinitialise en même temps les coordonées du curseur.

Théoriquement, elle ne sert pas beaucoup, Grammer n'ayant pas de commande pour écrire sur l'écran home. C'est un buffer de 128 octets qu'on peut quand même modifier, alors la commande existe.

Horizontal

Dessine une ligne horizontale sur l'écran.

:Horizontal *Y* **,** *[option]* **,** *[buffer]*

Y: L'ordonée de la ligne horizontale (0-63).
option: 0 (ligne blanche) , 1 (ligne noire) , 2 (inversion des pixels de la ligne). Noir si non précisé.
buffer: le pointeur du buffer sur lequel dessiner. Non précisé = buffer par défaut.

Vertical

Dessine une ligne verticale sur l'écran.

:Vertical *X* **,** *[option]* **,** *[buffer]*

X: L'abscisse de la ligne verticale (0-95)
option: 0 (ligne blanche) , 1 (ligne noire) , 2 (inversion des pixels de la ligne). Noir si non précisé.
buffer: le pointeur du buffer sur lequel dessiner. Non précisé = buffer par défaut.

ShiftBuf(

Cette commande sert a faire un scrolling de l'écran graphique dans une direction précise.

:ShiftBuf(*nombre-de-pixels* **,** *direction , [buffer]*

nombre-de-pixels: Il faut préciser à la commande le nombre de pixels dont vous souhaitez décaler l'écran dans la direction demandée.
direction: 1 (bas), 2 (droite) , 4 (gauche), 8 (haut). On peut faire plusieurs directions en ajoutant les valeurs, par exemple pour décaler l'écran vers le haut et la droite: 8+2=10
buffer: le pointeur du buffer sur lequel dessiner. Non précisé = buffer par défaut.

TileMap(

Utilisée pour le tilemapping, dans les jeux principalement.

:TileMap(*0* **,** *carte* **,** *tile* **,** *largeur-de-map* **,** *map-X-offset* **,** *map-Y-offset* **,** *méthode*

carte: pointeur vers les données de map
tile: pointeur vers le set de tiles
largeur-de-map: la largeur de la carte (au moins 12)
map-X-offset: la position X dans les données de map
map-Y-offset: la position Y dans les données de map
méthode: les 7 méthodes d'affichage du sprite (voir Tile)

Fill(

Utilisée pour faire des commandes de dessin en gros.

0 écran noir
1 inverse l'écran
2 remplit l'écran avec un damier
3 remplit l'écran avec un autre damier
4 Permet de copier un octet sur tous les octets du buffer avec une logique OU. L'octet est à préciser en deuxième argument, le plus facile est de l'écrire en binaire, mais on peut aussi bien le mettre en hexa (entre 0 et F) ou en décimal (entre 0 et 15).
Par exemple, **Fill(4,%10101010** va créer un buffer virtuel dont tous les octets seront 10101010 (1 pixel sur 2), puis va mélanger ce buffer avec le buffer par défaut, en appliquant une logique OU, créant ainsi un fond d'écran.
Si vous ne voulez pas de la logique OU, les Fill(5,6 et 7 peuvent vous servir.
5 idem, logique XOR
6 idem, logique ET
7 idem, logique EFFACER
8 copie un le buffer sur un autre buffer, préciser le pointeur du deuxième buffer(**Fill(8, pointeur**)
9 copie un autre buffer sur le buffer de l'écran, avec une logique OU
10 idem, logique ET
11 idem, logique XOR
12 idem, logique EFFACER
13 idem, logique SWAP (échange le buffer avec celui précisé.)
14 le buffer est copié sur lui même avec un décalage de X pixels vers le bas et une logique OU (X à préciser)
15 idem, logique ET
16 idem, logique XOR
17 idem, logique EFFACER

18 comme la 14, mais le décalage se fait vers le haut.
19 comme la 15, mais le décalage se fait vers le haut.
20 comme la 16, mais le décalage se fait vers le haut.
21 comme la 17, mais le décalage se fait vers le haut.
22 fait bruler l'écran pour 1 cycle. Préciser le type: 0 (feu blanc) ou 1 (feu noir).
23 **Fill(23,***type*,*Y*,*X*,*largeur*,*hauteur* permet de faire bruler une zone précise de l'écran. Y , X , largeur et hauteur définissent un rectangle (qui n'est pas tracé). Voir Rect, et le type est le même que ci dessus (feu noir ou blanc).
Attention, X (la largeur) compte en colonnes, pas en pixels ! (1 col= 4px)

RecallPic

Affiche sur le buffer une image de l'OS. (comme en TI Basic). Il est possible de choisir la méthode (voir "sprites") et le buffer sur laquelle ont veut l'afficher.

:RecallPic *numéro* , [*méthode*] , [*buffer*]

Attention, Grammer lui peut créer jusqu'à 256 Pic, archivées ou non. (plus exactement, le "numéro" peut être de 0 à 255.) Vous verrez simplement apparaitre dans le menu mémoire des Pics aux noms étranges comme **length(** ou **^** .

Si vous souhaitez utiliser les Pics normales du TI Basic, enlevez 1 à leur numéro: Pic1=0 , Pic2=1, etc...

StorePic

Stocke un buffer dans une image de l'OS. Si non précisé, c'est le buffer par défaut qui est enregistré.
Mêmes remarques que pour RecallPic: 256 Pics de dispo, archivées ou non, et celles du Basic ont un numéro de moins.
Une Pic stockée ainsi pèsera 779 octets. *(taille du buffer (768) + taille du nom (2) + 9)*
Cette fonction est utile pour les images en niveaux de gris.

:StorePic *numéro* , [*buffer*]

Le moteur de particules:

Grammer possède son propre moteur de particules intégré. Vous devez indiquer la position de la particule au départ, puis sa trajectoire va être gérée automatiquement par le programme. Les particules mises à jour en boucle vont à peu près à 5.66 FPS.

AddPart(

Ajoute une particule aux coordonnées du pixel précisé (**:AddPart(Y,X**).

RunPart

Recalcule la position des particules, et les affiche à l'écran.

ClrPart

Efface le buffer spécifique aux particules.

PartType(

Change le type de particule, et les règles de gravité.

PartType(*option* **,** *[" chaine-d'ordre]*

option: 0 = sable, 1 = eau bouillante, 2 permet de changer l'ordre de gravité.
chaine d'ordre: utilisez, séparées par des virgules, la priorité des opérations.
D (Down) = Bas
L (Left) = Gauche
R (Right) = Droite
U (Up) = Haut

exemple: **:PartType(2,"D,LR,U**

essaye d'aller en bas, si ce n'est pas possible (obstacle) va sur les cotés, et sinon, en haut....

Il est possible d'utiliser un chiffre si vous préférez, si la chaine ne vous arrange pas:

considérez 16 bits comme composés des directions:
0000 1000 0110 0001 *Les priorités sont de droite à gauche.*

essaye d'aller en bas, si ce n'est pas possible va sur à gauche ou droite, et sinon, en haut, et sinon rien....
Inscrivez donc: **:PartType(2,2145** (0000100001100001 en décimal)

AddPart('

Sert à convertir tout les pixels allumés d'une zone de l'écran en particules !
On lui donne de quoi définir un rectangle, et tous les pixels allumés du rectangle deviendront des particules. les coordonées sont celles du coin haut-gauche.

AddPart(' *Y , X , hauteur , largeur*

Pour vous apprendre leur fonctionnement, je préfère un exemple qu'un long discours:

:ClrPart	*Efface le buffer spécifique aux particules.*
:AddPart(2,2	*Crée une particule sur le pixel de coordonées 2,2*
:Repeat getKey=15	*la portion de code sera répétée jusqu'à ce que [clear] soit pressée*
:RunPart	*Met à jour la position de la particule dans le buffer*
:DispGraph	*Affiche l'écran*
:End	

Si vous avez compris comment fonctionne un interrupteur, la bonne idée consiste à en allumer un qui contienne l'instruction **:RunPart** , comme ça vos particules sont automatiquement mises à jour tant que l'interrupteur est allumé.

Les niveaux de gris:

On peut faire du dessin en niveau de gris si on utilise deux buffers.

Pour celà, une **mise à jour constante ou très rapide** de l'écran est nécéssaire. **DispGraph** partout, tout le temps, par exemple dans un interrupteur.

(Merci Xeda pour ce magnifique dessin ! ;))

On va définir pour celà les deux buffers au préalable avec **SetBuf(°** *(graybuffer)*et **SetBuf('** *(blackbuffer).*
(remarque: Si on ne définit pas SetBuf(' , c'est le buffer par défaut qui sert de Blackbuffer.)
Les noms de "Black" et "Gray" buffers ne sont là que pour se repérer.

Ce qu'il faut savoir, c'est que si un pixel n'est allumé que sur un des deux, il sera **gris**, s'il est allumé sur les deux, il sera **noir**.

Par conséquent, il y a des pixels qui doivent être allumés sur les deux, et d'autres sur qu'un seul.

Conseils pratiques:

1) **Définir les deux buffers a une adresse différente chacun** (le blackbuffer par défaut est celui de l'écran normal, il n'y a pas de graybuffer par défaut)
2) **Dessiner les zones noires sur le blackbuffer**
3) **Dessiner les zones grises et les noires sur le graybuffer**
4) mettre a jour l'écran intensément à coup de DispGraph.
et le tour est joué !
(encore une fois, je rapelle qu'on peut très bien mettre les zones noires sur le gray et les zones grises+noires sur le black, ces noms ne sont là que pour nous aider à nous repérer...)

```
:.0:
:SetBuf(°π9872
:SetBuf('π86EC
:ClrDrawπ9872
:ClrDrawπ86EC
:For A,1,9
:Circle(30,30,A,1,0,π9872
:End
:For A,1,9
:Circle(30,30,A,1,0,π86EC
:A+1 ↓ A
:End
:Repeat getkey(15
:DispGraph
:End
:Stop
```

Nous obtenons ainsi un cible gris/noir/gris/noir/gris etc....

Dans le cas où dessiner ligne par ligne n'est pas très optimisé, (comme par exemple pour l'image ci dessus), il est bien entendu possible de rappeler une image (RecallPic) sur un buffer précis. ça va bien plus vite, et ça doit se valoir question poids.

Dans le cas ou le rendu de gris ne vous plait pas, il y a moyen de changer le pourcentage: Au lieu que chaque buffer compte pour 50% de la couleur noire (par défaut), on peut rendre un des deux buffers plus clair et l'autre plus foncé, à différents niveaux. Et tout ça varie encore selon le niveau de contraste que vous utilisez.

Celà se règle avec la commande *nombre* **↓ SetBuf(**
et selon la valeur du nombre:

Nombre	**° buffer**	**' buffer**	
0	100%	0%	
1	50%	50%	(par défaut)
2	33%	67%	
3	25%	75%	
4	17%	83%	
5	8%	92%	
6	0%	100%	(comme si y'avait pas de niveaux de gris)
7	50%	50%	(comme le 1)
8	67%	33%	
9	75%	25%	

10	83%	17%
11	92%	8%

Contrast(

Modifie le contraste de l'écran. Entrez une valeur entre 0 et 39, la normale étant 24 environ.

Les pourcentages s'appliquent à ce constraste, ce qui fait beaucoup de possibilités différentes. Rappelez vous seulement qu'une TI a contraste élevé consomme plus.

XII. Enregistrer des données

Ces fonctions servent à accéder à ce qui est enregistré dans la mémoire, et a modifier tout ça.

FindVar(

Cette fonction permet d'accéder aux variables de l'OS. Précisez le nom de la variable dans une chaine, et la fonction donnera un pointeur vers cette variable. Si la variable n'existe pas, il donnera 0.
Si elle est archivée, Θ' contiendra le numéro de la page flash où elle est enregistrée, (et Ans aura une valeur inférieure à 32768).

Exemple: **:FindVar("ESPRITES→A** *stockera dans A un pointeur vers le contenu du programme SPRITES.*

Pourquoi le E ? Je le rappelle, c'est la lettre de type qui indique qu'on parle d'un programme.

:FindVar(" *lettre-de-type nom-de-la-variable*

(

A utiliser pour lire un octet précis de la RAM. (8-bits)

{

A utiliser pour lire deux octets précis de la RAM. (16-bits = 1 "mot")

WriteB(

A utiliser pour écrire un octet précis dans la RAM. Préciser le pointeur vers l'endroit où écrire, puis l'octet à écrire.

:WriteB(*pointeur* **,** *contenu-8-bits*

WriteW(

A utiliser pour écrire un "mot" (soit 2 octets=16 bits) dans la RAM.

Précisez le pointeur vers l'endroit ou écrire, puis le contenu à écrire.
Par exemple, écrire 0 (un nombre 16 bits) dans le programme prgmABC:

```
:FindVar("EABC→A      // toujours la lettre de type...
:WriteW(A,0           // va placer deux octets de zéros (soit le nombre 16-bits 0) à
                         l'adresse du pointeur A.
```

:WriteW(*pointeur* **,** *contenu-16-bits*

MakeVar(

Cette commande sert à créer une variable de l'OS quand elle n'existe pas encore. Il suffit de préciser tout d'abord la taille en octets allouée à la variable qu'on veut créer, puis son nom dans une chaine, en n'oubliant pas la lettre de type. Une chose sympa ? Vous avez le droit d'utiliser les lettres minuscules.

:MakeVar(*taille* **,"** *lettre-de-type nom*

[[[(

Le crochet sert à enregistrer des octets dans la RAM, des octets de masse.

On l'utilise de plusieurs façons, selon qu'on veut écrire un octet à la fois (nombres de 0 à 255) ou des "mots" complets (nombres de 0 à 65535, de deux octets).

: *adresse* **[** *octet1 , octet2 , ... , octetX* écrit les octets précisés à l'adresse donnée.
: *adresse* **[[** *2-octets1 , 2-octets2 , ... , 2-octetsX* écrit par groupe de 2 octets, à l'adresse donnée.

Les adresses peuvent être précisées directement, ou dans un pointeur. Les octets sont précisés en Décimal, mais stockés dans la RAM en hexa. Pour stocker un "mot" au milieu de valeur à 1 octet, ajouter le degré ° après la valeur.

Exemples **:A[0,3,65,23** écrira "00034117" dans la RAM à l'adresse que contient le pointeur A.
(la suite 00,03,41,17 est la traduction en hexadécimal de 0,3,65,2.)

:63542[21,16,32°,231,765° écrira "15102000E7FD02" dans la RAM à l'adresse 63542.
(La suite 15,10,2000,E7,FD02 est la traduction en hexadécimal de 21,16,32,231,765, sachant qu'on à demandé à ce que le 32 et le 765 prennent deux octets.)
(obligatoire dans le cas du 765: il faut obligatoirement 2 octets pour stocker un aussi grand nombre, mais inutile dans le cas du 32: d'où un octet vide (deux 0) qui ne sert à rien.)

:A[[123,456,7645,34567,0,0,34 écrira "7B00C801DD1D0787000000002200", les deux crochets indiquant que tous sont priés d'utiliser 2 octets pour s'enregistrer.
(7B00,C801,DD1D,0787,0000,0000,2200 est la traduction en hexa de 123,456,7645,34567,0,0,34, chaque nombre étant prié de s'enregistrer sur deux octets.)

Plus simplement, il est possible d'entrer directement les données en hexa dans la mémoire en utilisant la parenthèse:
On en se soucie plus alors de savoir si c'est en 1 ou 2 octets à la fois.

:A[(1BAC534FF625A3D20A0F5 écrira "1BAC534FF625A3D20A0F5" dans la mémoire, soit l'exacte même chose.

GetInc(

La commande **GetInc(** est faite pour lire la mémoire.
Elle renvoie l'octet que pointe le pointeur qu'on lui précise, puis augmente le pointeur de 1. Ainsi, la prochaine fois que le pointeur est utilisé, il pointra vers l'octet qui est juste après. Utilisée plusieurs fois de suite, la commande lit donc plusieurs octets consécutifs.

exemple: lire les premiers octets du programme prgmABC:

```
:FindVar("EABC→A
:Text('0,0,GetInc(A,16          //ne pas refermer la parenthèse.
:Text('°GetInc(A,16
:Text('°GetInc(A,16             //le 16 sert à afficher l'octet en hexadécimal.
:Text('°GetInc(A,16               On l'affiche en hexa, parce qu'un octet en
:Text('°GetInc(A,16               hexa fait 2 caractères et prend peu de place.
:Text('°GetInc(A,16
```

Archive
UnArchive

Sert à archiver/désarchiver une variable, même un programme. Préciser le nom de la variable dans une chaine, précédée par sa lettre de type.

:(Un)Archive " *lettre-de-type nom*

Delvar

Fonctionne exactement comme les deux précédentes fonctions, sauf qu'elle supprimme la variable au lieu de la changer de mémoire.

:Delvar " *lettre-de-type nom*

sub(

Utilisée pour supprimmer du contenu d'une variable.

:sub(*durée* **,** *saut* **,"** *lettre-de-type nom*

durée: Le nombre d'octets à supprimer dans la variable. 4 supprimera 4 octets.
saut: Le nombre d'octets à sauter avant de commencer à supprimer, si vous voulez effacer non pas le début de la variable, mais plus vers le milieu.
lettre-de-type: Lettre qui indique le type de variable.
nom: Le nom de la variable.

Insert(

Exactement l'inverse de **sub(**, elle sert à insérer du contenu dans une variable..

:Insert(*durée* **,** *saut* **,"** *lettre-de-type nom*

durée: Le nombre d'octets à insérer dans la variable. 4 insérera 4 octets. Ces octets seront vides.
saut: Le nombre d'octets à sauter avant de commencer à insérer, si vous voulez insérer non pas au début de la variable, mais plus vers le milieu.
lettre-de-type: Lettre qui indique le type de variable.
nom: Le nom de la variable.

XIII. La commande Misc(

Misc(

Elle sait en faire, des choses, la commande Misc ! Selon les arguments qu'on lui donne, elle est capable de trouver des données et de les copier ou les déplacer. Et aussi plein d'autres trucs qui n'ont rien a voir.

Misc(0," *lettre nom1* **","** *lettre nom2* **",** *[taille]* **,** *[début]*

Misc(0 sert à faire des copier-coller. par exemple, copier la chaine Str1 dans la chaine Str2, ou le programme ABC dans le programme BCD:

:Misc(0,"DStr1","DStr2 **:Misc(0,"EABC","EBCD"**

Si la nouvelle variable existait déjà le contenu est remplacé. Sinon elle est créee directement avec le contenu copié. Ans contiendra un pointeur vers la nouvelle variable, et Θ' la taille de cette nouvelle variable (en octets).

taille : Si au lieu de copier toute la variable, vous ne voulez copier que 123 octets par exemple.
début : Si au lieu de commencer à copier vos 123 octets au début de la variable, vous souhaitez sauter quelques octets et commencer à copier plus loin.

Misc(1, *adresse1* **,** *adresse2***,** *taille*

Misc(1 sert à copier les données se situant après l'adresse1 (incluse) et les coller après l'adresse2 (elle aussi incluse). "taille" est le nombre d'octets à copier/coller. Les adresses peuvent être des variables-pointeurs ou des valeurs pointeurs directement écrites.

exemple: sauvegarder l'image de l'écran dans Pic1:
:Misc(1, adresse-du-buffer-utilisé , adresse-de-Pic1 , 768
768 est la taille d'un buffer et d'une Pic. Ceci est l'équivalent de **StorePic 1**.

Misc(2, *adresse1* **,** *adresse2***,** *taille*

Misc(2 sert à copier les données se situant avant l'adresse1 (incluse) et les coller avant l'adresse2 (elle aussi incluse). "taille" est le nombre d'octets à copier/coller. Les adresses peuvent être des variables-pointeurs ou des valeurs pointeurs directement écrites.

La différence entre **Misc(1** et **Misc(2**: Choisissons de copier 4 octets de données à l'adresse 87 par exemple.

Etat de la mémoire:

adresse		*82*	*83*	*84*	*85*	*86*	*87*	*88*	*89*	*90*	*91*	*92*	
	...	|	|	|	|	|	|	|	|	|	|	|	...
contenu de l'octet (hex) (choisi au hasard)		*5F*	*7A*	*E0*	*03*	*B1*	*49*	*F8*	*1C*	*7A*	*30*	*DB*	

Misc(1,87, *adresse2* **,4** va copier de 87 à 90, les données copiées seront "49F81C7A".
Misc(2,87, *adresse2* **,4** va copier de 84 à 87, les données copiées seront "E003B149".

Misc(3, *pointeur*

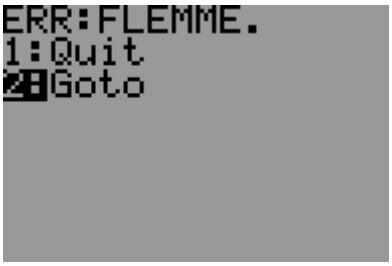
Celui ci permet de customiser ce que le programme fait lors d'une erreur. Par exemple enregistrer des données, ou quitter le programme dans les règles de l'art, ou encore réinitialiser des variables et recommencer. Il faut donc préciser la commande **Misc(3** au début du programme, suivie par le pointeur du label suivi par le code à éxécuter lors de l'erreur.

Ans contiendra le code de l'erreur: 0 = ERR ARRET (vous avez appuyé sur [On]), 1 = ERR MEMOIRE.

Misc(3 n'affecte pas la valeur de Θ' ou d'Ans. Si l'erreur change leur valeur, il seront remis ensuite à la valeur qu'ils avaient avant, le programme s'en souvient.

Misc(4, *option , ["chaine]*

Cette commande là est plutôt géniale. Elle permet de créer une erreur sur demande.
option: 0 (ERR ARRET) , 1 (ERR MEMOIRE) ou 2 (personnalisé).
Si vous utilisez 2, vous avez alors besoin de préciser le nom de votre erreur dans une chaine, ensuite.
exemple: erreur FLEMME, faites **Misc(4,2,"FLEMME** .



Tripoter le Hardware de la TI (Attention dangereux et peu utile pour les débutants !)

La TI utilise comme tout matériel informatique une carte mère sur laquelle est branchée divers ***ports***, qui servent à ceci ou celà.

Exemples: le port Jack, l'écran LCD, le clavier, le timer (sur 84+ uniquement)...
Avec Grammer, il est possible d'envoyer des octets à ces ports, ou de lire leur état, pour communiquer avec eux.
La plupart du temps, celà se fait automatiquement *(exemple, la commande **conj** utilise le port jack, la commande **getkey** utilise le clavier, la commande **DispGraph** utilise l'écran LCD)*, mais il est également possible de le faire manuellement, via les commandes Misc(5 et 6.

/!\ VOUS UTILISEZ CECI A VOS RISQUES ET PERILS /!

liste des ports:

00 = port Jack (cable TI-TI)
01 = clavier (pour lire les touches pressées)
16 = commande de l'écran LCD
69 (low) à 72 (up) = Timer, 8 bits par 8 bits

Misc(5, *valeur* **,** [*port*]

Pour envoyer une valeur dans un port.
Si le port n'est pas précisé, la valeur par défaut est 0 (port Jack).

- Envoyez une valeur de 0 à 3 au port jack et elle sera envoyée via le cable. (*)

- Pour désactiver l'écran LCD, écrivez 2 dans le port 16. Pour le réactiver, écrivez 3.

(Attention, l'écran parait éteint, mais la TI est allumée et fonctionne normalement, sauf qu'elle n'affiche rien. Vous devez arriver à créer un programme pour la rallumer à l'aveuglette, faute de quoi vous n'avez plus qu'a retourner en acheter une autre.)

Misc(6, *port*

Lit la valeur d'un port. Ne présente pas de risques graves à utiliser contrairement à **Misc(5**.

Exemple: un programme qui compte le temps:

```
:0:Return
:ClrDraw
:Return→T
:Misc(6,69
:+256*Misc(6,70
:→B
:Repeat getkey(15
:Text('0,0,-B-expr(T
:DispGraph
:End
:Stop
```

(toutes ces commandes n'ont pas été vues, si vous ne les comprenez pas lisez leur description dans le chapitre d'après...)

Backup des pointeurs:

Il est possible de sauvegarder toutes les variables pointeurs d'un coup dans son endroit de la RAM préféré, pour pouvoir ensuite les utiliser et les remettre a leur valeur de départ. C'compliqué tout ça.

En Bref, les variables pointeurs toutes mises ensemble pèsent 108 octets.

Misc(7 , *adresse*

Enregistrera les 108 octets des variables pointeurs a l'adresse de la RAM indiquée.

Misc(8 , *adresse*

Redistribuera a chacun la valeur qu'il avait lors du dernier backup.

Autres Commandes

Voilà expliquées les fonctions que je n'ai pas déjà citées.

Les opérateurs:

// double slash — permet d'entrer des annotations, des commentaires...

Tout ce qui est entré sur une ligne après un double slash est ignoré. Cela permet au programmeur de mettre des notes dans son programme.

π symbole pi — indique que le nombre qui suit est en hexadécimal.

Par exemple, π3F sera compris comme 63.

! plusieurs utilisations.

- fonctionne comme **not(** en Basic, c'est à dire donne l'inverse du résultat d'un test logique boléen.
- Placé devant une condition **If**, il permet que l'instruction du If ne soit exécutée que si la condition est FAUSSE.
- Placé devant un **While**, un **Repeat** ou un **Pause If** , il permet d'inverser la condition de ces fonctions.

_ [2nd][.] — signale que la variable suivante est une variable de l'OS.

E E des puissances de 10 [2nd][,] — signale que la chaine qui suit est en binaire.

Autres Fonctions:

Pause Indiquer après **Pause** la durée de la pause en centièmes de seconde. (approximatif).

:Pause *nombre-de-100e-de-seconde*

Pause If Suivie d'une condition, cette fonction arrête le programme tant que la condition est vraie.
Il est souvent d'utiliser !Pause If , par exemple pour attendre qu'une touche soit pressée:

:!Pause If getkey=9

:Pause If *condition*

AsmPrgm Permet l'éxécution de code assembleur non-compilé(hexadécimal). C9 est requis.

:AsmPrgm*code assembleur*

Asm(Permet l'éxécution d'un programme en assembleur, tout comme en TI-Basic.

getkey je vous en ai parlé déjà plusieurs fois dans des exemples comme **:!Pause If getkey=9 .**
Nous allons voir son fonctionnement en détail. Elle permet de détecter quelles touches du clavier sont pressées. Chaque touche possède en fait un numéro, dont je donne la liste ci dessous.
getkey aura en fait la valeur de la dernière touche pressée. On peut ainsi faire des conditions selon les touches pressées:

If getkey=9 *Si la dernière touche pressée est [enter]*

*Cette notation peut s'optimiser en **getkey(9** .*

Dans les boucles, il est souvent conseillé de la stocker dans une variable pour s'en souvenir et faire des tests ou des calculs dessus.

getkey→K puis K est utilisé dans les calculs. (Traditionellement, on utilise K ou G pour stocker le getkey, mais ce n'est absolument pas une obligation).

Input

Permet de demander le contenu d'une chaine àl'utilisateur. Input donne en fait le pointeur vers la chaine que l'utilisateur rentre, qui est conservé jusqu'à la prochaine utilisation d'**Input**.

Contrairement au TI-basic, l'Input se passe ici sur l'écran graphique, et il n'y a pas de point d'interrogation affiché automatiquement. Le texte que vous rentrez sera inversé (surligné).
La touche [On] fera planter le programme (bug qui sera surement réglé dans le futur), vous ne pouvez pas
quitter un programme en appuyant sur [On] au cours d'un Input.
Il est possible d'effacer ses caractères avec [del] ou [clear].

Appuyer sur [enter] vous fait forcément sortir du Input, même si aucune valeur n'a été rentrée.
Le pointeur pointe alors vers des données qu'il y avait là avant.
(pratique pour entrer une valeur "par défaut".)

Input ["*texte*"]

il est en effet possible d'ajouter une chaine de caractères après Input: ce texte sera affiché après ce que l'utilisateur tape.

Le plus souvent, Input est utilisée suivie d'un sto→ pour enregistrer le pointeur dans une variable.

Si c'est un nombre que vous voulez demander à l'utilisateur, utilisez (comme en TI Basic) la commande **expr(** , qui permet de transformer le contenu d'une chaine en nombre.

:expr(Input→A *stocker la valeur numérique que l'utilisateur entrera dans la chaine dans la variable A.Pour un nombre 32 bits, utilisez deux variables.*

Exemple de code:

```
:.0:
:ClrDraw
:Text(°"(x,y)=(
:expr(Input ",)→X
:Text(,+1
:expr(Input ")→Y
:Pxl-On(Y,X
:DispGraph
:Stop
```



expr(

Permet même plus que ça: Si vous avez une chaine qui contient une ligne de code, elle permet d'éxécuter cette ligne de code.
Ainsi, elle peut traduire une chaine en nombre, mais également en une instruction, un calcul...

inString(

Permet de trouver une sous-chaine à l'intérieur du programme. Une sous-chaine est en fait une suite de caractères qui fait partie d'une chaine plus large (comme le code du programme par exemple, qui est en fait une grosse chaine...).
La fonction donne le pointeur de la sous-chaine, la taille de la sous-chaine est aussi donnée automatiquement dans Θ'.

:inString(*début* **,"** *chaine-à-chercher*

début: Un pointeur vers l'endroit du programme ou doit commencer la recherche. Souvent, c'est un pointeur vers un Label. (voir exemple ci dessous).
chaine à localiser: La sous-chaine que vous recherchez.

Un exemple: rechercher la sous chaine "DEF"

```
:Lbl "ALPHABET→A
:inString(A,"DEF→B
: ...
:Stop
:
:.ALPHABET
:ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

B sera un pointeur vers la chaine "DEF" et Θ' vaudra 3.

length('

Utilisée pour trouver des lignes de code dans un programme. (renvoie le pointeur de la ligne.)
La longueur de la ligne en octets sera stocké dans Θ'.
Notez que cette commande interprète chaque "deux-points" comme une nouvelle ligne, même si ce n'est pas une vraie nouvelle ligne.

length(' *début* **,** *durée* **,** *numéro-de-ligne* **,** *[numéro-du-caractère]*

début: Pointeur vers l'endroit o⋅commencer la recherche (un programme, un label, un **Return**...)
durée: La durée de la recherche, en octets. Si vous mettez 0, celà cherchera dans toute la RAM.
numéro-de-ligne: Le numéro de la ligne qu'on cherche. Il est forcément inférieur ou égal à la durée, sinon, le programme s'arrête de chercher avant qu'il soit arrivé à la bonne ligne.
numéro-du-caractère: Dans le cas ou les données dans lesquelles vous recherchez ne sont pas des lignes de code mais autre chose, c'est à dire qu'il n'y a pas les "deux-points" qui veulent dire nouvelle ligne.
Vous pouvez donc entrer ici le numéro du caractère qui sera considéré comme marquant le début d'une nouvelle ligne, si le deux-points ne vous plait pas. ça permet par exemple de chercher le 4e mot dans une phrase, en séparant les mots par des espaces.
La liste des caractères est donnée en annexe de ce tutoriel.

l'intérêt de cette commande est principalement de stocker la ligne de code extraite dans une chaine de l'OS (Str1 par exemple), puis de l'éxécuter avec **expr(.**

length(

Donne la taille d'une variable (en RAM ou en Archive), ainsi que le pointeur sur cette variable dans Θ'.
Si la variable n'est pas trouvée, Θ' vaura -1 (soit 65535)

Il faut préciser le nom de la variable dans une chaine, en n'oubliant pas la lettre de type.

Vous remarquerez que la taille donnée par cette commande est inférieure à celle donnée par le menu mémoire de la TI. C'est parce que cette commande donne juste la taille des données. Le menu de la TI, lui, donne *taille-des-données + taille-du-nom + 9* , systématiquement.

Fix

Fix Text(: voir commande **Text(**.
Cette commande sert à initialiser un ou plusieurs modes à la fois.

Fix 1 : Ecrire en texte inversé (blanc sur fond noir)
Fix 2 : Inverse les pixels. Un pixel On sera considéré comme éteint, et un Off comme allumé
Fix 4 : Empêcher l'interruption du programme par [On]. [On] pourra alors être utilisé avec **getkey** (key-code pour la touche [On] = 41)
Fix 8 : Mode Hexadécimal: tous les nombres sont lus en hexa, comme s'il y avait π devant.
Fix 16: Ce mode vous permet d'envoyer une erreur quand vous utilisez Pxl-test(en dehors des limites de l'écran.

Notez qu'il est possible de les additionner: **Fix 3** = **Fix 1+2,** donc il inversera le texte ET les pixels.

Fix utilisé seul est égal à l'état du mode. On peut ainsi l'utiliser dans des tests.
Par exemple **Fix ou 6** pour interdire On et inverser l'écran.

Full

Ne fonctionne pas sur les TI 83+ basiques: cette commande sert à activer le mode 15 MHz. Grammer est déjà bien rapide, mais si vous avez une 84+ ou une Silver Edition, vous pouvez l'accélérer encore !

Full 0 : Passe en mode 6 MHz (mode de base)
Full 1 : Passe en mode 15 MHz (rapide !)
Full 2 : Change le mode entre l'un et l'autre.

SetFont

Commande utilisé pour changer de police de caractères.

:SetFont *0,1-ou-3* **,** *[pointeur]*

SetFont0 : Passe en mode police Grammer (par défaut), tous les lettres et symboles font 4 pixels de large et 6 de haut. Même l'espace. Les coordonées X sont à donner en "colonnes" et non pas en pixels donc. *(comme si le graphique était un homeScreen en TI-Basic.)*
SetFont1 : Passe en mode police TI-OS, avec des largeurs de caractères variables.
SetFont2: Mode Police Grammer, mais où l'on place le texte non pas en colonnes mais aux coordonnées de pixel.
SetFont3 : Pour instaurer une police Omnicalc ou BatLib. Précisez le pointeur vers le programme-police Omnicalc.
pointeur : Si vous avez un set de caractères personnalisé d'installé sur votre TI, entrez un pointeur vers le début de ce set, et la police de caractères personnalisée sera instaurée. Si ce set est un set Omnicalc, ajoutez 11 à ce pointeur (les sets omnicalc ont 11 octets d'infos inutiles pour Grammer au début).

Je veux utiliser la police Omnicalc CUSTFONT.

conj(

Une commande super importante !! En effet, c'est celle qui vous permet de jouer une note.

:conj(*note* **,** *octave , durée*

note:	Do=0	Fa#=6	octave:	7 octaves de 0 à 6 sont dispo.
	Do#=1	Sol=7		mais restez en medium, je vous en prie.
	R•2	Sol#=8		
	R•=3	La=9	durée:	une valeur de 1 (court) à 64 (très long).
	Mi=4	La#=10		
	Fa=5	Si=11		

conj('

Permet de lire des données de son brutes, sans avoir à convertir les nombres...
La taille est en "mots" et pas en octets. Donc si vous connaissez la taille en octets, divisez là par 2. (*)

:conj(' *durée* **,** *pointeur , taille*

Menu(

Permet d'afficher une popup ressemblant à un clic droit de souris, avec un choix multiple d'options.
Le mieux, c'est un exemple pour vous le présenter:

:Menu(10,10,40,"TITRE...","OPTION1","OPTION2","OPTION3"

Lorsque l'utilisateur appuiera sur [enter], Ans prendra la valeur du numéro de l'option moins 1
(option1=0 , option2=1 , etc...)

:Menu(*Y , X ,* *largeur* **,** **"** *titre* **" , "** *option1* **" , "** *option2* **" , ...**

Attention, vous devez penser à régler la largeur pour que les textes ne dépassent pas.
Attention aussi: le menu ne s'efface pas de l'écran après votre choix... à vous de l'effacer.

XV. Catalogue du Grammer

Commande avec le Hook	Commande "Basic"	Emplacement
>lFactor	>Frac	[math][1]
Misc(	solve(	[math][0]
WriteW(	iPart(	[math][>][3]
WriteB(	int(	[math][>][5]
Inv(	not(	[2nd][math][>][4]
Clrpart	R>Pr(	[2nd][apps][5]
RunPart	R>PΘ(	[2nd][apps][6]
AddPart(	P>Rx(	[2nd][apps][7]
PartType(	P>Ry(	[2nd][apps][8]
GetInc(	Is>(	[prgm][alpha][math]
call	prgm	[prgm][alpha][x-1]
SetBuf	Disp	[prgm][>][3]
SetFont	Output(	[prgm][>][6]
FindVar(	Get(	[prgm][>][alpha][math]
MakeVar(	Send(	[prgm][>][alpha][apps]
Rect	Line(	[2nd][prgm][2]
ShiftBuf(	Tangent(	[2nd][prgm][5]
Contrast(	Shade(	[2nd][prgm][7]
Tile	Pt-On(	[2nd][prgm][>][1]
Sprite	Pt-Off(	[2nd][prgm][>][2]
TileMap	Pt-Change(	[2nd][prgm][>][3]
_	¿	[2nd][.]
Insert(	augment(	[2nd][x-1][>][7]
2^(	e^(	[2nd][ln]

Les autres fonctions se trouvent au même endroit qu'en TI-Basic. *(voir le catalogue du TI-Basic: http://tiemulation.kegtux.org/TIBasic.htm)*

XVI. Leçon de Binaire

Quelques explications sur comment traduire un nombre entre ses formes décimale, hexadécimale et binaire.
Si vous connaissez déjà tout ça, vous pouvez sauter ce chapitre.

Decimal = Base 10 = notre système numérique courant (0 à 9)
Hexadécimal = Base 16 = langage machine (0 à F)
Binaire = Base 2 = octets, pixels et sprites (0 à 1)

Dec>Hex:

1) Diviser le nombre par 16, et prendre le reste de cette division. Si c'est entre 0 et 9 c'est bon, si c'est entre 10 et 15 utilisez les lettres A à F. (A=10, B=11, C=12, D=13, E=14, F=15)
2) Si le nombre est 16 ou plus, re-divisez le.
3) recommencez jusqu'à ce qu'il ne reste plus rien.

exemple: convertir 32173 en hex.

32173/16= 2010 ,	reste 13 :	“D”
2010/16= 125 ,	reste 10 :	“A”
125/16= 7 13,	reste 13 :	“D”
7/16= 0 ,	reste 7 :	“7”

Le nombre est donc **7DAD**.

Hex>Dec

1) Prendre le chiffre des unités, multipliez le par 16^0.
2) Prendre le chiffre des dizaines, multipliez le par 16^1.
3) Prendre le chiffre des centaines, multipliez le par 16^2.
4) ainsi de suite selon la taille du nombre, multipliez le par 16^n.
5) Additionnez tout les résultats obtenus.

exemple: convertir 7A02 en décimal.

2x16^0	=2
0x16^1	=0
10x16^2	=2560
7x16^3	=28762

2+0+2560+28762=31234
Donc le nombre vaut **31234**.

Binaire:

C'est pareil, remplacez juste les 16 par des 2, et faites les mêmes calculs.

Bin>Hex: les calculs marcheraient, mais il est quand même bien plus rapide et pratique quand vous concevez des sprites tout le temps de connaitre cette liste par coeur...

(Rappel: le petit b veut dire que le nombre est en binaire, le petit h qu'il est en hexa)

 = 0000 b = 0 h
 = 0001 b = 1 h
 = 0010 b = 2 h
 = 0011 b = 3 h
 = 0100 b = 4 h
 = 0101 b = 5 h
 = 0110 b = 6 h
 = 0111 b = 7 h
 = 1000 b = 8 h
 = 1001 b = 9 h
 = 1010 b = A h
 = 1011 b = B h
 = 1100 b = C h
 = 1101 b = D h
 = 1110 b = E h
 = 1111 b = F h

Autres bases:
Octal, ou autres encore, remplacez les 16 par 8 ou le nombre de votre choix.

XVII. Liste de Tokens

(*) Peut-être plus tard si j'en trouve une bien...

XVIII. Scripts intéressants

Il existe certains petits, qui relèvent parfois plus de l'assembleur que du Grammer, mais que je peux laisser a votre attention.

Les auteurs respectifs de ces codes sont actuellement Xeda et Matrefeytontias. Merci à eux.

- Autoriser les minuscules:
:AsmPrgmFDCB24DEC9
(Désactiver les minuscules: :AsmPrgmFDCB249EC9)

- Lancer une Application:

Les Applications ont un nom de 8 tokens, même si on dirait pas (les tokens suivants sont des espaces).
Il faut bien insérer les 8 tokens a l'endroit du nom.

:Get("ENomDel'App
:AsmPrgm2178843614EF4E4CD8D306C38040C9

- Lancer un programme en demandant son nom:

(*) En dévellopement

Remerciements

Je voudrais remercier ceux qui m'ont aidé a écrire ce tutoriel, donc bien entendu Xeda, merci à elle, c'est précieux une fille aussi intelligente il en faudrait plus des comme ça.
Merci à Kindermoumoute pour avoir pris plusieurs fois le temps de m'expliquer des choses tordues dans les entrailles de la RAM d'une TI.
Merci au Professeur Refeyton pour son support pour les scripts (même si je les ai pas postés).
Merci à l'hébergeur kegtux pour son serveur gratuit. Merci aux communautés Omnimaga et Cemetech pour leurs forums de support et de dévellopement.
Merci à Adobe Acrobat X pour écrire des pdf.
Merci à Samsung pour le netbook qui m'a permis de taper ce tuto, à la Bibliothèque Universitaire Sciences de Grenoble et tout particulièrement à l'Université Joseph Fourier pour la connexion internet qu'elle me prête, merci à ma mère qui me laisse écrire de tutos alors que je devrais bosser,
merci à ceux qui ont lu ce tutoriel, grâce à vous j'ai l'impression d'avoir été utile.

Merci et a bientôt dans la prochaine mise a jour.

updates at http://tiemulation.kegtux.org/Grammertutorial.htm