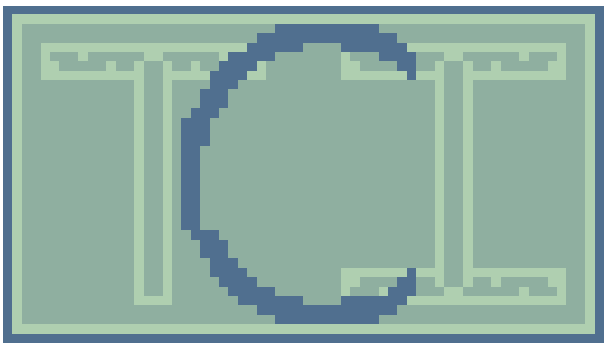
# TI-92plus, V-200 & TI-89



# Tutorial ASM

Rédigé par Pascal MARTIN
http://www.squalenet.net/

# Sommaire

# Introduction

Historiquement, l'Assembleur est le premier langage véritablement puissant (au sens de rapidité, mais aussi de fonctionnalités offertes) que nos calculatrices TI aient proposées. Déjà du temps de la TI-92, il était possible de programmer en Assembleur sur ces machines (ou plutôt, pour ces machines !).
Depuis, il est devenu possible de programmer en C pour les TI-89, 92+ et V-200. Ce langage est traité dans un autre des tutoriaux de la TCI.

L'Assembleur est aussi, pour chaque machine, le langage le plus basique qui soit, ou presque : chaque processeur dispose d'une liste d'instructions, qui correspondent en mémoire à une simple de suite de bits. En Assembleur, nous travaillons directement avec ces instructions, sauf que, au lieu de les écrire sous forme de suite de bits, nous leur avons fait correspondre des noms plus faciles à retenir, généralement de trois, ou quatre, lettres.
Une fois que nous aurons écrit notre programme sous forme d'une suite de "noms d'instructions", nous demanderons à un logiciel, nommé Assembleur, de la traduire en en suite de bits, qui soit compréhensible pour le microprocesseur.

Pour nos calculatrices, il existe deux logiciels Assembleurs :

- GNU-AS : C'est l'Assembleur utilisé par le compilateur C inclus dans le pack TIGCC.
- A68k : C'est celui qui est généralement utilisé par les programmeurs en Assembleur.

La syntaxe demandée par le premier est assez lourde, et, de plus, les programmeurs Assembleurs utilisent généralement le second. Nous étudierons donc dans ce tutorial la syntaxe attendue par A68k. (Les différences ne sont pas énormes ; peut-être les étudierons-nous plus tard...)

Tout au long de ce tutorial, nous supposerons, sauf quand le contraire sera explicitement spécifié, que nous travaillerons sous l'interface TIGCC-IDE, en utilisant l'Assembleur A68k.
Vous pouvez télécharger le pack TIGCC sur http://tigcc.ticalc.org ; il vous permettra de programmer en Assembleur A68k, GNU-AS, mais aussi en GNU-C !
Ce tutorial a été conçu pour la version 0.95 de TIGCC. Il est donc possible que certains exemples que nous vous proposerons ne fonctionnent pas avec des versions antérieures. De même, bien que la TIGCC-Team essaye au maximum de conserver une compatibilité maximale avec les anciennes versions, il est possible que certains exemples ne fonctionnent pas avec des versions plus récentes de TIGCC (cela est fortement improbable, mais, on ne sait jamais, ce qui explique pourquoi nous précisons que cela peut se produire).

Écrire un tutorial est chose difficile : il faut parvenir à être progressif dans le niveau de difficulté, des sujets abordés ; il faut aussi réussir à être clair, de façon à ce que nos explications soient comprises par le plus grand nombre, et, surtout, il faut rédiger des exemples suffisamment brefs pour être aisément compréhensibles, tout en étant suffisamment complets afin de bien mettre en évidence leur sujet.
Nous vous demandons donc d'être indulgent par rapport aux erreurs que nous pourrions être amené à commettre, et nous en excusons d'avance.
Pour toute suggestion et/ou remarque, n'hésitez pas à nous contacter via l'adresse E-mail que vous trouverez en bas de chaque page de notre tutorial.

Bon courage !

# Licence

# Chapitre 1

## I:\ Soyez courageux !

Pour ne rien vous cacher, le langage d'Assembleur (souvent désigné sous le terme "Assembleur", bien que ce soit un abus de langage, puisque "Assembleur" désigne le logiciel qui assemble) est difficile, voire même très difficile. De plus, sa syntaxe est extrêmement rebutante, notamment lorsqu'on débute son apprentissage.

Je vais peut-être vous paraître dur, mais si votre but est de diffuser la semaine prochaine, ou même le mois prochain, le jeu de vos rêves alors que vous ne connaissez actuellement encore rien de l'Assembleur, vous avez trois solutions :

- Cesser de rêver.
- Revoir vos rêves à la Baisse.
- Ne pas continuer la lecture de ce tutorial.

Pourquoi cela ? Tout simplement parce que, en une semaine, ou même en un mois, vous ne parviendrez pas à atteindre un niveau suffisant : il vous faudra non seulement connaître l'Assembleur, mais aussi savoir écrire des algorithmes adaptés à vos besoins, savoir implémenter ces algorithmes en Assembleur, puis il vous faudra le temps de programmer le jeu en question, ce qui ne se fait pas en un jour !
Ce que je veux éviter, c'est que vous commenciez à apprendre l'Assembleur, que vous réalisiez dans un mois que c'est plus complexe que vous ne le pensiez, et que, découragé, vous laissiez tout tomber. Je veux que, maintenant, vous admettiez que c'est difficile, et que, sachant cela, vous décidiez de la conduite à adopter : ne pas continuer à lire ce tutorial, et dire "au revoir" au langage Assembleur, ou alors, poursuivre votre lecture, fermement résolu à aller jusqu'au bout de ce tutorial, et, ensuite, à continuer d'apprendre par vous-même.
C'est à vous, et à vous seul, de choisir.

# II:\ Quelques notions de Bases, vraiment indispensables

Après ces propos pas vraiment encourageants, je le reconnais, nous allons pouvoir commencer à nous intéresser au sujet de ce chapitre.
Principalement, ce qui fait la puissance des programmes en Assembleur, c'est qu'ils "disent" directement au processeur que faire ; ceci est aussi, force est de le reconnaître, un danger : il n'est pas difficile de planter la calculatrice lorsqu'on programme en Assembleur : il suffit de bien peu de choses.
Puisque nous donnons des ordres directement au processeur, et que celui-ci n'est capable de rien d'autre que de manipuler des bits, il nous sera absolument nécessaire de comprendre, et de savoir utiliser, le binaire.

En général, on désigne sous le terme de "Base n" la base qui utilise les chiffres de 0 à n-1. Par exemple, nous utilisons quotidiennement la base 10, aussi appelée décimale, qui utilise les chiffres de 0 à 9.
Le logiciel Assembleur est capable de "comprendre" les bases 2 (binaire), 10 (décimal), et 16 (hexadécimal). La base 10 est reconnue car c'est elle que nous avons l'habitude de manipuler, la base 2 est admise car c'est la plus proche de la machine (chaque bit peut prendre deux valeurs : 0 ou 1, le courant passe ou ne passe pas, la charge magnétique est positive ou négative, ...), et la base 16 est employée pour minimiser le nombre d'erreurs à la lecture et à l'écriture par rapport à la base 2 (car 4 chiffres binaires correspondent à un chiffre hexadécimal).

En langage d'Assembleur pour le logiciel Assembleur A68k (nous dirons à présent en ASM-A68k, ou alors ASM, ou A68k tout court, bien que les deux dernières écritures soient des abus de langage), les nombres exprimés en base 2 doivent être préfixés d'un signe pourcent "%", les nombres exprimés en base 16 doivent être préfixés d'un caractère dollar "$", et les nombres en base 10 ne doivent pas être préfixés.

En base 16, puisque notre alphabet ne comporte pas 16 chiffres, nous utilisons les dix chiffres usuels, soit de 0 à 9, puis les six premières lettres, soit de A à F.
A présent, nous désignerons sous le terme de "digit" chaque chiffre d'une écriture (il s'agit du terme anglais pour "chiffre") ; cela parce que le terme de "chiffre" n'est pas adapté aux écritures hexadécimale, qui peuvent comporter des lettres. En binaire, un digit est également appelé "bit", en abréviation de "BInary digiT".
Lorsque nous parlerons de, ou des, digit(s) de poids faible, il s'agira de ceux correspondant à la partie droite d'une écriture, en comptant de droite à gauche, et quand nous parlerons de digits de poids fort, il s'agira de ceux de gauche du nombre. Par exemple, dans le nombre %1001010, le bit de poids fort vaut 1, et les quatre bits de poids faible valent %1010.

# III:\ Passage d'une base à une autre

Savoir différencier une base d'une autre est certes utile, mais il peut être quasiment indispensable de passer de l'une à l'autre.
Certains d'entre-vous ont peut-être déjà étudié ceci (Voila quelques années, c'était au programme de Spécialité Maths, en Terminale S, je ne sais pas si c'est toujours le cas : les programmes de Lycée ont changé depuis). Si vous êtes dans ce cas, vous pouvez passer à la partie suivante (une fois n'est pas coutume), bien qu'un petit rappel ne fasse jamais de mal...

Tout d'abord, voici un petit tableau de correspondance :

| Décimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Binaire | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
| Hexadécimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

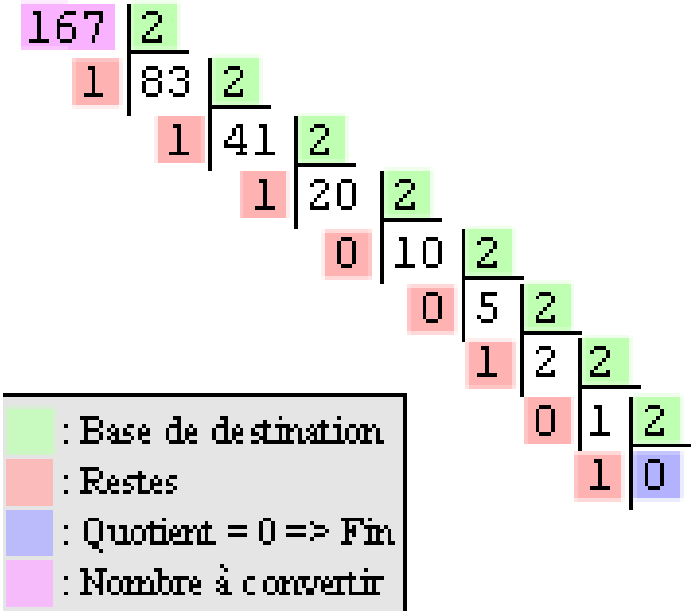
## A: De décimal vers binaire et hexadécimal

Que l'on veuille passer de décimal vers binaire, ou de décimal vers hexadécimal, le raisonnement est fondamentalement le même. Nous noterons ici b la base de destination, que ce soit binaire, ou hexadécimal.

La méthode que je considère comme la plus simple est celle-ci :

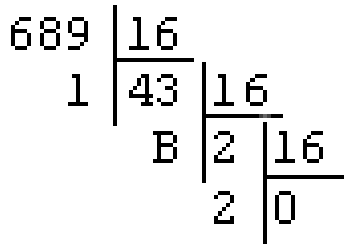
- On divise le nombre en base 10 par b.
  => On obtient un quotient q1.
- On divise q1 par b.
  => On obtient un quotient q2.
- On divise q2 par b.
  => On obtient un quotient q3.
- Et ainsi de suite...
- Quand le quotient obtenu vaut 0, on cesse les divisions, et on "remonte" dans la liste des restes obtenus, en les écrivant de gauche à droite.

Par exemple, on veut convertir 167 de la base 10 vers la base 2 :




Donc, on a : 167 = %10100111.

Convertissons à présent 689 de la base 10 vers la base 16 (L'organisation de la succession de divisions est la même que pour le premier exemple) :




Donc, on a : 689 = $2B1.

## B: De binaire et hexadécimal vers décimal

Peu importe la base source, si la base de destination est le décimal, on procède toujours de la même façon. Nous noterons ici b la base d'origine, que ce soit le binaire, ou l'hexadécimal.
La méthode, bien que extrêmement simple est assez difficile à expliquer. En supposant que l'on ai un nombre xyzt en base b, voici comment procéder :
xyzt = x*b^3 + y*b^2 + z*b^1 + t*b^0
Et ensuite, on calcule le membre de droite de cette expression.
Je pense que deux exemples suffiront à clarifier mes dires :

Par exemple, convertissons %11010010 en décimal :
```
%11010010 = 1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 0*2^0
          = 1*128 + 1*64 + 0*32 + 1*16 + 0*8 + 1*4 + 1*2 + 0*1
          = 422
```
Donc, %11010010 = 422.

A présent, convertissons $5DA1 en décimal :
```
$5DA1 = 5*16^3 + D*16^2 + A*16^1 + 1*16^0
      = 5*4096 + 13*256 + 10*16 + 1*1
      = 23969
```
Donc, $5DA1 = 23969.

## C: De binaire à hexadécimal, et inversement

Il n'y a rien de plus facile : il suffit de se rappeler qu'un digit hexadécimal correspond à quatre digits binaires, et que la conversion se fait du quartet (groupe de quatre bits, parfois aussi appelé "Nibble") de poids faible vers celui de poids fort, en remplissant éventuellement de 0 à gauche si nécessaire. Il suffit ensuite d'utiliser le tableau de correspondance donné plus haut.

Par exemple, convertissons %1011001001101010 en hexadécimal :
```
%10.1100.1001.1010.1010 = %0010.1100.1001.1010.1010
                        = $  2    C    9    A    A
```
Donc, %1011001001101010 = $2C9AA.

Et convertissons $1B2D en binaire :
```
$1B2D = %0001.1011.0010.1101
      = %1101100101101
```
Donc, $1B2D = %1101100101101.

Voila la fin de ce premier chapitre atteinte.
Notez que votre TI-89/92/V-200 est parfaitement capable de réaliser les conversions de base entre binaire, décimal, et hexadécimal... mais reconnaissez qu'il peut être intéressant de parvenir à se passer de la calculatrice !

# Chapitre 2

## I:\ Ce qui est "visible"

Ce tutorial est prévu pour les calculatrices Texas Instrument modèles TI-89/92/V-200. A peu de choses près, ces trois machines sont identiques : leurs différences majeures sont au niveau de la taille de l'écran, et du clavier.

Il existe deux versions matérielles différentes : les HW1, et HW2.
Les HW1 sont les plus anciennes, les HW2 les plus récentes. Les HW2 comportent quelques fonctionnalités supplémentaires par rapport aux HW1 (par exemple, les HW1 n'ont pas de support d'horloge). La V-200 est considérée comme une HW2 (Il n'existe pas de V-200 HW1).
Au cours de notre apprentissage de la programmation en Assembleur, il sera rare que nous ayons à nous soucier des différences entre les différentes versions matérielles, mais, quand il le faudra, nous préciserons que c'est le cas.

Il existe aussi plusieurs versions de "cerveaux". Ce "cerveau" est le logiciel qui vous permet de faire des mathématiques, de programmer en TI-BASIC, de dessiner, de lancer des programmes en Assembleur, ... ; bref, ce "cerveau" est ce grâce à quoi votre machine est plus qu'un simple tas de composants électroniques.
Différentes versions allant de 1.00 sur TI-92+, 1.01 sur TI-89, et 2.07 sur V-200, à, pour le moment, 2.09 (sortie durant le second trimestre 2003) ont été diffusées par Texas Instrument. En règle générale, plus la version est récente, plus elle comporte de fonctions intégrées, directement utilisables en Assembleur.
Ce "cerveau" est généralement appelé "AMS" (pour Advanced Mathematic Software), ou, par abus de langage, "ROM" (Pour Read Only Memory), puisque l'AMS est stocké dans un mémoire flashable de ce type.
Le plus souvent possible, nous essayerons de rédiger des programmes compatibles entre les différentes versions de ROM, mais, dans les rares cas où ce ne sera pas possible (par exemple, parce que nous aurons absolument besoin de fonctions qui ne sont pas présentes sur certaines anciennes ROM), nous le signalerons.

# II:\ Le Microprocesseur

Nos TIs sont dotées d'un processeur Motorola 68000, cadencé à 10MHz sur HW1, et à 12MHz sur HW2.
(Ce processeur était utilisé sur les premiers Macintosh, sur certaines consoles ATARI, et est toujours utilisé sur les rames de TGV atlantiques !)

Ce processeur comporte 8 registres de données, nommés de d0 à d7, et 8 registres d'adresse, de a0 à a7. (Un registre est un petit emplacement mémoire situé à l'intérieur même du processeur, et qui est donc très rapide ; c'est le type de mémoire le plus rapide, mais aussi le plus cher, ce qui explique pourquoi il n'y en a que si peu). Ces 16 registres ont une taille de 32 bits.
Les registres de données, de même que ceux d'adresse (de a0 à a6 seulement : a7 est un peu particulier), sont généraux, ce qui signifie que l'on peut mettre un peu ce qu'on veut dedans (Il n'y a pas, comme sur certains autres processeurs, un registre réservé pour les boucles, ou autre). Cependant, étant donné le faible nombre de registres, il vaut mieux les utiliser de façon judicieuse !
Le registre a7 est particulier dans le sens où il joue le rôle de "pointeur de pile" : c'est lui qui contient l'adresse du sommet de la pile. La pile ("Stack", en anglais), est une zone mémoire un peu particulière, sur laquelle on "empile" ou "dépile" des données... Nous verrons plus tard ce que cela signifie exactement.

Le M68k (Abréviation de Motorola 68000) possède également un registre nommé "PC" (Program Counter) qui sert à indiquer en permanence au processeur quelle est la prochaine instruction qu'il devra utiliser. Rassurez-vous, lors de l'exécution normale d'un programme, c'est le processeur lui-même qui se charge de modifier de manière adéquate la valeur du PC. Ce registre est de 24 bits.

Ce processeur dispose aussi d'un registre de statut, nommé "SR" (Status Register), sur 16 bits. Nous étudierons tout d'abord les 8 bits de poids faible, puis les 8 bits de poids fort.
Voici le schéma représentatif du SR :

| Nom | T | - | S | - | - | I2 | I1 | I0 | - | - | - | X | N | Z | V | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N° bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Registre de Flags :
Il s'agit de l'octet de poids faible du SR.
Voici, globalement, la signification des différents bits utilisés le comportant :

- Bit C (Carry) : Utilisé comme retenue.
  Ce bit servira par exemple de neuvième bit si vous additionnez deux nombres de huit bits.
- Bit V (oVerflow) : Ce bit sera armé (vaudra 1) en cas de valeur trop grande pour être représentable.
- Bit Z (Zero) : Ce bit sera armé si le résultat d'un calcul (en général) vaut 0.
- Bit N (Negative) : Ce bit sera armé si le bit de poids fort d'un résultat l'est (Nous verrons bientôt pourquoi, quand nous étudierons le complément à deux).
- Bit X (eXtended) : Ce bit est une copie du bit C, mais ne sera pas modifié de la même façon.

Octet Système :
Il s'agit de l'octet de poids fort du SR.
Voici à présent, de faççon rapde, la signification des différents bits utiles le composant :
Attention, ceci est assez technique. Si vous n'avez pas déjà de bonnes connaissances en architecture et fonctionnement des processeurs, il est probable que vous ne saisirez pas ce qui suit. Ce n'est pas pour l'instant grave : ceci n'est utilisé qu'à un niveau assez avancé de programmation. Vous comprendrez en temps voulu.

- Bits I0, I1, I2 : Ces trois bits permettent de définir le masque d'interruptions du processeur. Plus la valeur codée par ces trois bits est élevée, plus le niveau de priorité est élevé.
- Bit S (Superviseur) : Quand ce bit est armé, c'est que le processeur n'est pas en mode utilisateur (le mode par défaut, le plus utilisé), mais en mode Superviseur. Le mode Superviseur vous donne accès à toutes les instructions du processeur, notamment à celles de contrôle du système, qui ne sont pas accessibles en mode utilisateur. Vous ne pouvez pas modifier l'octet système du SR lorsque vous n'êtes pas en mode Superviseur ; donc, pour passer du mode utilisateur au mode Superviseur, il convient d'appeler une routine système spéciale.
  Travailler en mode Superviseur n'est utile que pour écrire des programmes contrôlant le système, et nécessitant un niveau avancé de programmation. Nous ne travaillerons pas en mode Superviseur avant fort longtemps.
- Bit T (Trace) : Lorsque ce bit est armé, une interruption est générée à la fin de chaque instruction. Ce bit est utilisé, ainsi que cette fonctionnalité, par les débogueurs. Je ne voie pas d'autre type de programme qui puisse lui fournir un emploi.

# III:\ Façon que la TI a de calculer

Ce dont nous allons ici parler tient encore une fois plus au processeur qu'à la TI, puisque nous nous pencherons sur la représentation des nombres en interne (au niveau du processeur et de la mémoire).

## A: Représentation des nombres

Les processeurs Motorola stockent les nombres en Big Endian ; c'est-à-dire que l'octet de poids fort d'un nombre est rangé "à gauche".
Un petit exemple rendra les choses plus clair : le nombre $12345678 sera représenté en mémoire par la suite d'octets suivant : $12, $34, $56, $78.
A titre d'information, les processeurs Intel de la famille ix86 (386, 486, et Pentiums) auraient représenté ce nombre par la suite d'octets suivant : $78, $56, $34, $12.
Force est de reconnaître que la technique utilisée par Motorola permet au programmeur de parfois se "simplifier la vie" !

## B: Codage des entiers en Binaire

Les entiers, que ce soit négatifs, ou positifs, sont codés selon la technique dite du complément à deux.

Pour les entiers positifs :
Nous avons déjà vu comment passer du décimal au binaire ; c'est exactement ce qu'il faut faire ici.
Par exemple, en travaillant avec des mots de deux octets, on aura :

16849 = %0100000111010001 (=$41D1)

Pour les entiers, le bit de poids fort est considéré comme un bit de signe. S'il vaut 0, l'entier est positif, et s'il vaut 1, l'entier est négatif.
Par exemple, %1101111000010001 ne sera pas considéré comme 56849, mais comme nombre négatif !

Pour les entiers négatifs :
Cette fois-ci, c'est un peu plus difficile ; mais il suffira de retenir la technique utilisée, qui sera toujours la même. (Il y a d'autres méthodes, mais c'est celle-ci que j'utilise, car c'est celle que je considère comme la plus simple).
Voici, toujours avec des mots de deux octets, comment encoder la valeur -16849 :
On commence par "traduire" la valeur absolue du nombre ; ici, 16849 :

16849 = %0100000111010001

Puis, on inverse la valeur de tous les bits (les bits à 0 passent à 1, en inversement ; cela revient à faire le complément à un du nombre), on obtient :

%1011111000101110

Enfin, on ajoute 1 au résultat :

```
  %1011111000101110
+ %0000000000000001
= %1011111000101111
```

Remarque : Une addition en binaire se fait de la même façon qu'en décimal. La seule chose dont il faut se rappeler, c'est que 1+1 = 10.

(Précisons que la TI est dotée d'une fonction de conversion qui parvient à effectuer ceci ! Il suffit de saisir -16849>Bin).

Utiliser la technique du complément à deux permet de mémoriser :

- Sur un octet, des valeurs allant de -128 (%1000.0000) à 127 (%0111.1111).
- Sur deux octets, des valeurs allant de -32768 (%1000.0000.0000.0000) à 32767 (% 0111.1111.1111.1111).
- Sur quatre octets, des valeurs allant de -2147483 à 2147483647.

Comme vous avez pu le remarquer, la valeur minimale représentable est égale à la valeur maximale représentable, plus un. Ceci n'est pas sans inconvénients, et peut parfois poser problème, si l'on ne fait pas assez attention.
Il arrive que l'on travaille avec des valeurs dites "non-signées" ("unsigned" en anglais). Dans ce cas, les 8, 16, ou 32 bits sont tous utilisés pour coder une valeur positive, comprise entre 0 et, respectivement, 2^8-1, 2^16-1 et 2^32-1.

# IV:\ Un peu de vocabulaire

Le processeur Motorola 68000 que nous utilisons, pour lequel nous allons programmer, est capable de reconnaître trois types de données entières. A chacun de ces types correspond une lettre, qui sera souvent utilisée comme suffixe pour certaines instructions, afin que le processeur puisse déterminer sur quel type de données elles doivent travailler ; le suffixe sera séparé de l'instruction par un point.

- Octet : Le type le plus petit (mis à part le bit, bien entendu !), mais qui constitue la base de toutes nos mémoires et systèmes informatiques. Octet se dit "Byte" en anglais (ne pas confondre avec "bit" : un "byte" est composé de huit "bits"), ce qui explique que le suffixe qui lui corresponde soit ".b".
- Mot : Le type le plus courant sur processeur M68k. Le mot est codé sur 16 bits (le 68000 est un processeur 16 bits en externe). "Mot" se dit "Word" en anglais. Le suffixe correspondant est donc ".w".
- Mot Long : Un mot long ("Longword" en anglais) est codé sur 32 bits (le M68k est un processeur 32 bits en interne, comme en témoigne la taille des registres). Le suffixe associé aux mot longs est ".l".

Je n'ai pas l'intention de vous forcer à ingérer plus de théorie tout de suite, même si je suis d'avis que celle-ci pouvait s'avérer nécessaire, sans vous permettre enfin de programmer ; dès le prochain chapitre, nous réaliserons notre premier programme en Assembleur.
J'aurai pu, comme j'en avais l'intention au départ, partir dès le premier chapitre sur un exemple de programme, et vous fournir les explications que je vous ai donné dans les chapitres 1 et 2 en même temps que celles portant sur cet exemple, mais j'ai vite réalisé que cela ferait beaucoup trop de nouveautés en même temps. J'ai donc préféré cette solution qui même si elle est moins "ludique" car ne commence pas par du code, est probablement plus adaptée pour une bonne raison : lorsque vous parviendrez au premier exemple de code, vous disposerez déjà de bases qui, bien que assez peu approfondies, devraient vous faciliter la compréhension.

Bonne continuation !

# Chapitre 3

Dans ce chapitre, nous allons enfin commencer à programmer !
Nous commencerons par une petite discussion sur le sujet "Kernel vs Nostub", puis nous réaliserons un premier programme, ne faisant rien, mais qui nous permettra de nous familiariser avec TIGCC, afin de pouvoir continuer sur des projets plus importants.

## I:\ "Kernel vs Nostub" : Que choisir ?

Sur nos TIs, il existe deux modes de programmation. Le premier (historiquement parlant, puisqu'il a vu le jour au temps de la TI-92) est le mode Kernel. Le second, qui s'est répandu à peu près en même temps que la possibilité de coder en langage C pour nos TIs est le mode Nostub.
Chaque mode, quoi qu'en disent certains, présente des avantages, et des inconvénients. Je vais ici tenter de vous décrire les plus importants, afin que vous puissiez, par la suite, faire votre choix entre ces deux modes, selon vos propres goûts, mais aussi (et surtout !) selon ce dont vous aurez besoin pour votre programme.

Mode Kernel :

| Avantages | Inconvénients |
|---|---|
| • Permet une utilisation simple de librairies dynamiques (équivalent des Dll sous Windows) déjà existantes (telles Ziplib, Graphlib, Genlib, ...), ou que vous pourriez créer par vous-même.<br>• Le Kernel propose de nombreuses fonctionnalités destinées à vous faciliter la vie, ainsi qu'un système d'anticrash parfois fort utile. (Une fois le kernel installé, l'anticrash l'est aussi, pour tous les programmes que vous exécutez sur la machine ; pas uniquement le votre !) | • Nécessite un programme (le Kernel) installé avant que vous ne puissiez lancer le votre.<br>• L'utilisation de librairies dynamiques fait perdre de la RAM lors de l'exécution du programme (parfois en quantité non négligeable) si toutes les fonctions de celle-ci ne sont pas utilisées. Cependant, notez qu'il est tout à fait possible de programmer en mode Kernel sans utiliser de librairies dynamiques ! Naturellement, la mémoire RAM est récupérée une fois l'exécution du programme terminée. |

Mode Nostub :

| Avantages | Inconvénients |
| --- | --- |
| <ul><li>Ne nécessite pas de Kernel installé (Fonctionne même, normalement, sur une TI "vierge" de tout autre programme).</li><li>En théorie, si le programme n'a pas besoin des fonctionnalités proposées par les Kernels (qu'il lui faudrait ré-implémenter !), il pourra être plus petit que s'il avait été développé en mode Kernel (car les programmes en mode Kernel sont dotés d'un en-tête de taille variable, qui peut monter à une bonne cinquantaine d'octets, et jamais descendre en dessous de environ 20-30 octets)<br>Cela dit, en pratique, c'est loin de toujours être le cas, en particulier pour les programmes de taille assez importante.</li></ul> | <ul><li>Ne permet pas, en ASM, la création et l'utilisation de librairies dynamiques (du moins, pas de façon aussi simple qu'en mode Kernel !) ; cela est actuellement permis en C, mais pas encore en ASM.</li><li>En cas de modifications majeures (par Texas Instrument) dans les futures versions d'AMS, certaines fonctions d'un programme Nostub peuvent se révéler inaccessibles, et alors entraîner des incompatibilités entre la calculatrice et le programme. Il faudra alors que l'auteur du programme corrige son code et redistribue la nouvelle version du programme (Sachant que la plupart des programmeurs sur TI sont des étudiants, qui stoppent le développement sur ces machines une fois leurs études finies, ce n'est que rarement effectué !). Ce n'est pas le cas en mode Kernel, pour les fonctions des librairies dynamiques : l'utilisateur du programme n'aura qu'à utiliser un Kernel à jour pour que le programme fonctionne de nouveau.</li></ul> |

Dans ce tutorial, nous travaillerons en mode Nostub. Non que je n'apprécie pas le mode Kernel (bien au contraire), mais le mode Nostub est actuellement le plus "à la mode". Je me dois donc presque dans l'obligation de vous former à ce qui est le plus utilisé...
Cela dit, il est fort probable que, dans quelques temps,nous étudions pendant quelques chapitres le mode Kernel, ceci non seulement à cause de son importance historique, mais pour certaines des fonctionnalités qu'il propose. A ce moment là, nous le signalerons explicitement.
Bien que n'étudiant pas tout de suite le mode Kernel, je tiens à préciser, pour ceux qui liraient ce tutorial sans trop savoir quel Kernel installer sur leur machine (s'ils souhaitent en installer un, bien entendu), que le meilleur Kernel est actuellement PreOS, disponible sur www.timetoteam.fr.st. C'est le seul qui soit à jour : DoorsOS est totalement dépassé, TeOS l'est encore plus, de même que PlusShell, et UniversalOS n'est plus à jour. De plus, PreOS propose nettement plus de fonctionnalité que DoorsOS ou UniversalOS ! (Notons que PreOS permet naturellement d'exécuter les programmes conçus à l'origine pour DoorsOS ou UniversalOS).

# II:\ Notre tout premier programme

Non, Non, vous ne rêvez pas, nous allons enfin voir notre premier programme ! Il ne fera absolument rien, si ce n'est, une fois lancé, retourner le contrôle au système d'exploitation de la TI (AMS, aussi appelé TIOS (TI Operating System)), mais nous permettra de découvrir, en douceur, les bases indispensables à tout programme que nous écrirons par la suite en Assembleur.

## A: Créer les fichiers nécessaires à notre programme

Nous supposerons que vous avez déjà installé le pack TIGCC, et ce de façon correcte.
Créez, quelque part sur votre disque dur, un dossier vide dans lequel nous placerons tous les fichiers de notre projet. Utiliser un dossier par projet est une habitude que je vous recommande vivement de prendre : même si, pour l'instant, nos projets ne se composeront pas d'une grande quantité de fichiers, d'ici quelques temps, lorsque vous travaillerez sur de gros projets, vous risquez d'être amené à utiliser beaucoup de fichier différents. Prenez donc l'habitude de bien les classer ! (Croyez-moi, quand on travaille sur un projet de plus de 500, voire même de plus de 1000, fichiers tout compris (codes sources, aide, images, archives externes, librairies et autres, on est bien content de les avoir bien ordonné dès le début (je ne plaisante pas !))
Nous désignerons, dans la suite de cette partie, ce dossier sous le nom "dossier1" ; libre à vous de choisir un nom plus évocateur !
Ouvrez le programme TIGCC IDE (Selon l'installation que vous avez effectué, un raccourci a normalement du être créé dans le menu démarrer de Windows).
Créez un nouveau projet ; pour cela, cliquez sur "File", puis "New", et enfin sur "Project". Nommez le du nom que vous souhaitez donner à votre programme. Dans la suite de nos explications, nous supposerons que vous avez choisi le nom "prog1".
A présent, il va falloir créer un fichier comportant l'extension ".asm", qui contiendra la texte de notre programme (son "code source"). Pour cela, cliquez sur "File", puis "New", puis sur "A68k Assembly Source File". Nommez ce fichier comme bon vous semble... (En général, je donne à mon fichier source principal le même nom qu'au projet, soit, ici, "prog1").
TIGCC IDE va alors vous créer un fichier, qui ne contiendra que ceci :

```
; Assembly Source File
; Created 11/02/2003, 23:18:07

    section ".data"
```

Il va à présent falloir taper notre code source...

## B: Ce qu'il faut absolument mettre

Avant tout, une petite remarque : Il est possible d'insérer, dans le texte de notre programme, des commentaires. Un commentaire est un texte, qui est généralement destiné à faciliter la compréhension du source, et qui n'est pas pris en compte par le logiciel Assembleur.
En A68k, les commentaires doivent être précédés d'un point-virgule, qui signifie que tout ce qui est écrit après, jusqu'à la fin de la ligne, est un commentaire.
Sous TIGCC IDE, avec les paramètres par défaut, les commentaires sont écrits en vert.

Précisons une petite chose supplémentaire tout de suite : en Assembleur, la majorité des instructions doit être précédée d'au moins un espace (en règle générale, nous utiliserons un caractère de tabulation). En fait, seules les étiquettes ne sont pas précédées d'espace(s). Petit "truc" simple pour repérer les étiquettes : elles sont assez souvent suivie d'un caractère ":" (Bien que celui-ci soit facultatif, nous le noterons toujours, en particulier au début de ce tutorial, par habitude, et par soucis de clarté).

A présent, voici comment commencer l'écriture de notre code source :

Tout d'abord, puisque nous allons utiliser des données (déclarations de fonctions, constantes, et autres) "standard", il nous faut les inclure. Pour cela, il faut écrire cette ligne dans notre fichier source :

```
include "OS.h"
```

Ensuite, puisque nous avons décidé de programmer en mode Nostub, il faut dire au logiciel Assembleur que c'est le cas, afin de celui-ci sache quel type d'exécutable générer. Pour cela, il convient de rajouter à notre fichier source :

```
xdef _nostub
```

A présent, il faut préciser pour quelle calculatrice vous voulez programmer.
Si vous destinez votre programme à une TI-89, écrivez :

```
xdef _ti89
```

Si c'est pour TI-92+, ou alors pour V200, que vous rédigez votre programme, écrivez :

```
xdef _ti92plus
```

Bien entendu, vous pouvez mettre les deux, dans le cas où vous destineriez votre programme aux deux machines simultanément. C'est ce que nous ferons généralement dans nos exemples, du moins quand ils n'utiliserons pas de fonctionnalités, telles le clavier ou l'écran, de façon spécifique à l'un ou l'autre des modèles.

En mode Nostub, l'exécution du programme débute au début du code source que nous écrivons. Donc, le code que nous écrirons "en haut" du fichier source sera le premier exécuté. Cependant, par pure habitude, plus pour faire joli et faciliter le repérage qu'autre chose, nous placerons toujours notre code après un label (une étiquette) nommée "_main", que nous ajouterons ainsi à notre source :

```
_main:
```

Après ce label, nous pouvons écrire le code du programme. Notre premier, celui-ci, ne fera absolument rien. Nous n'avons donc pas énormément de code à rajouter.
Cependant, puisque le TIOS (le Système d'exploitation de la TI) donne le contrôle de la machine au

programme au moment où il le lance, il faudra que celui-ci repasse la main au TIOS !
Pour cela, notre programme doit se terminer par cette instruction :

```
  rts
```

Cette instruction signifie "ReTurn from Subroutine". Elle signifie quitter la routine courante. Ici, la routine courante est notre programme ; la quitter signifie donc rendre le contrôle au TIOS.
Nous verrons un usage plus général de cette instruction lorsque nous travaillerons sur l'utilisation de fonctions, puisqu'elle nous permettra de quitter une fonction pour revenir à l'appellant...

## C: Résumé

Voila, nous avons terminé l'écriture de notre programme. Le voici, écrit en un seul bloc, pour faciliter la lecture :

```
; Assembly Source File
; Created 21/09/2002, 19:02:03

    include "OS.h"
    xdef    _nostub
    xdef    _ti89
    xdef    _ti92plus

_main:
    ; Lors de nos prochains programmes, c'est ici que nous insérerons notre
code.
    rts
```

Exemple Complet

A présent, il nous faut lancer l'Assemblage de ce programme. Pour cela, cliquez sur "Project", puis "Build", ou alors, enfoncez la touche F9 de votre clavier.
Une fois l'assemblage terminé, vous pouvez tester votre programme. Pour ce faire, je vous conseille de toujours utiliser un émulateur (tel VTI, disponible sur http://www.ticalc.org ), et de n'envoyer le programme sur votre vraie calculatrice que lorsque vous aurez pu constater qu'il était, à première vue du moins, dépourvu de Bug. Souvenez-vous en pour la suite de ce tutorial, ainsi que pour les programmes que vous serez amené à développer par vous-même, mais aussi, par prudence, pour ceux que vous pouvez télécharger sur Internet (On n'est jamais trop prudent) !

# Chapitre 4

Maintenant que nous savons écrire et assembler des programmes ne faisant rien, nous allons pouvoir (mieux vaut tard que jamais !) réaliser un programme... faisant "quelque chose".
Principalement, pour commencer du moins, nous nous intéresserons à l'utilisation des fonctions intégrées à la ROM de la TI. Lorsque nous appellerons l'une de ces fonctions, nous réaliserons un "appel à la ROM", traditionnellement appelé "ROM_CALL". Par extension, et abus de langage (ça ne fait qu'un de plus... ça doit être ce qu'on appelle l'informatique **:-)**), nous emploierons généralement ce terme pour désigner les fonctions inclues à l'AMS en elles-mêmes.
L'ensemble des ROM_CALLs constitue une librairie de fonctions extrêmement complète, et qui, en règle générale, s'enrichit à chaque nouvelle version de ROM. Cet ajout de nouvelles fonctions peut être source d'incompatibilités entre votre programme et des anciennes versions d'AMS. (Jusqu'à présent, il n'y a que de très rares fonctions qui aient disparues, et celles-ci ont été supprimées parce qu'elles présentaient un danger potentiel pour la machine, tout en n'ayant qu'une utilisé limitée.) ; cependant, le plus souvent possible, nous veillerons à conserver la plus grande compatibilité possible.
A titre d'exemple, les ROM 2.0x sont toutes dotées de plus d'un millier de ROM_CALLs, qui permettent de faire tout ce que le TIOS fait ! Nous n'utiliserons qu'une infime partie de ces fonctions pour ce tutorial, et il est quasiment certain que vous n'utiliserez jamais plus du tiers de tous les ROM_CALLs ! (tout simplement parce que vous n'aurez pas l'usage des autres, à moins de développer des programmes un peu particuliers).

La TI est dotée d'un Table, qui contient les adresses de tous les ROM_CALLs. Pour appeler l'un d'entre eux, nous devrons passer par cette table. Dans ce chapitre, nous ferons ceci de façon assez basique ; nous verrons plus tard comment optimiser ceci.

Il est possible de passer des paramètres à un ROM_CALL, si celui-ci en attend. Par exemple, pour une fonction affichant un texte à l'écran, il sera possible de préciser quel est ce texte ; pour un ROM_CALL traçant une ligne entre deux points, il faudra passer en paramètres les coordonnées de ces points.
Pour savoir quels sont les paramètres attendus par un ROM_CALL, je vous invite à consulter la documentation de TIGCC, fournie dans le pack que vous avez installé.
Nous verrons tout ceci au fur et à mesure de notre avancée dans ce chapitre...

# I:\ Appel d'un ROM_CALL sans paramètre

## A: Un peu de théorie

Pour pouvoir appeler un ROM_CALL, il nous faut déterminer son adresse. Celle-ci peut être obtenue à partir de la Table dont nous venons de parler.
Pour cela, il nous faut placer l'adresse de cette table, située en $C8, dans un registre. Ici, nous choisirons le registre a0. Pourquoi a0 ? Simplement parce qu'il nous faut un registre d'adresse (de a0 à a6, plus a7 que nous ne pouvons pas utiliser pour cela), et que a0 est le premier de ceux-ci.
L'instruction correspondante est celle-ci :

```
move.l $C8,a0
```

Ensuite, il nous faut déterminer, à partir de cette table, l'adresse du ROM_CALL, et, bien entendu, la mémoriser dans un registre (d'adresse, encore une fois !). Il sera possible d'utiliser le même que ci-dessus ; cela sera effectué grâce à cette instruction :

```
move.l nom_du_rom_call*4(a0),a0
```

Naturellement, il faut remplacer *nom_du_rom_call* par le nom du ROM_CALL que vous voulez appeler !
A présent, le registre a0 contient l'adresse du ROM_CALL désiré. Il ne nous reste plus qu'à sauter vers celui-ci. Pour cela, nous utiliserons l'instruction suivante :

```
jsr (a0)
```

L'instruction `jsr` ("**J**ump to **S**ub**R**outine") va placer sur la pile l'adresse de l'instruction qui la suit, pour que, une fois la fin du ROM_CALL atteinte, l'exécution du programme reprenne son cours de façon correcte, et va ensuite sauter vers l'adresse contenue dans le registre a0.

## B: Un exemple simple

Sur nos TIs, il existe un ROM_CALL, nommé `ngetchx`, qui attend un appui sur une touche, et qui ne prend pas de paramètre. Nous allons écrire un programme, dans lequel nous utiliserons ce que nous avons dit au chapitre précédent, ainsi que la théorie que nous venons d'expliquer. Ce programme, une fois lancé, attendra que l'utilisateur appuie sur une touche (sauf les modificateurs, tels que [2nd], [<>], [shift], [alpha] sur 89, et [HAND] sur 92+/V200), et rend le contrôle au TIOS. Voila le code source de ce programme :

```
; Assembly Source File
; Created 21/09/2002, 19:02:03

    section ".data"
    include "OS.h"
    xdef    _nostub
    xdef    _ti89
    xdef    _ti92plus

_main:
    move.l  $C8,a0
    move.l  ngetchx*4(a0),a0
    jsr     (a0)
    rts
```

Exemple Complet

# II:\ Appel d'un ROM_CALL avec paramètres

## A: Un peu de théorie

Appeler un ROM_CALL en lui passant des paramètres est chose extrêmement facile, une fois qu'on a compris le principe. Pour chaque ROM_CALL connu (et documenté), la documentation de TIGCC vous fourni la liste des paramètres qu'on doit lui passer. Il vous suffit de suivre cette liste, en ordre inverse, et d'envoyer chaque paramètre sur la pile.
Pour cela, souvenez-vous que le pointeur de pile est représenté, sur nos TIs, par le registre a7.
Ensuite, utilisez l'instruction voulue en fonction de la taille, et du type, du paramètre.
Malheureusement, les appellations C des types et tailles de données ne sont pas les mêmes qu'en Assembleur... Si vous ne connaissez pas du tout le C, voici un bref tableau de correspondance (liste non exhaustive) :

| Type C | Type ASM |
|---|---|
| (unsigned) char | Octet (1 octet). Cependant, le pointeur de pile devant pointer sur une adresse paire, le passage se fait sous la forme d'un mot de deux octets. |
| (unsigned) int<br>(unsigned) short<br>HANDLE | Mot (2 octets) : move.w en général |
| (unsigned) long | Mot-long (4 octets) : move.l ou autre instruction plus spécifique ou plus optimisée. |
| *quelque_chose* * (*quelque_chose* étant à remplacer par ce qui est, le cas échéant, nécessaire !) | Adresse (Mot-long, sur 4 octets) : move.l ou autre instruction plus spécifique, telle pea.l |

Je suis convaincu qu'un exemple sera bien nécessaire pour que vous voyez comment se fait réellement un appel de ROM_CALL nécessitant un ou plusieurs paramètre(s). Nous verrons tout d'abord comment appeler un ROM_CALL admettant un "*quelque_chose* *" en paramètre, puis nous verrons comment appeler des ROM_CALLs nécessitant des entiers en paramètres.

## B: Appel d'un ROM_CALL travaillant avec un *quelque_chose* *

Nous allons ici partir sur l'exemple d'un ROM_CALL permettant d'afficher à l'écran une chaîne de caractères que nous déterminerons. Pour que le ROM_CALL sache quelle est la chaîne de caractères à afficher, nous devrons lui passer l'adresse de celle-ci en paramètre.
Ce ROM_CALL, nommé ST_helpMsg nous permettra d'afficher notre message dans la "Status Line" (barre de statut : c'est la ligne fine en bas de l'écran de la TI, juste en-dessous de la ligne de saisie).
La documentation de TIGCC nous indique qu'il est déclaré ainsi :

```
void ST_helpMsg (const char *msg);
```

Pour l'instant, ne tenez pas compte du mot (ici, "void") situé devant le nom du ROM_CALL... Nous verrons plus tard quelle est sa signification, et comment, lorsque c'est possible, l'utiliser, mais, pour le moment, nous ne en soucierons pas : cela ferait beaucoup trop de nouveautés à assimiler simultanément.

Pour appeler le ROM_CALL, nous procéderons de la même façon que dans la partie précédente... mais il nous faudra, avant de l'appeler, placer sur la pile l'adresse de notre message... Avant cela, nous devons, bien entendu, déclarer ce message.
Pour cela, il convient de placer, tout à la fin de notre fichier source, une ligne ressemblant à celle-ci :

```
mesage: dc.b "Ceci est mon message !",0
```

Naturellement, vous pouvez remplacer le texte entre guillemets par celui que vous voulez : c'est lui qui sera affiché. Au début de la ligne, nous avons placé une étiquette, que nous avons appelé "message" ; elle sert à dire que la chaîne de caractères "Ceci est mon message !" est connue par le logiciel Assembleur sous le nom de "message".
Généralement, nous dirons que message est une variable de type chaîne de caractères, qui contient "Ceci est mon message !"
Vous pouvez remarquer que la ligne est terminée par un 0. Ceci est obligatoire sur nos TIs, qui veulent que les chaînes de caractères soient terminées par un octet valant 0. Pour les curieux, ceci est la norme admise en C, et qui est celle utilisée par Texas Instrument sur nos machines, du fait que la ROM soit programmée en C.

A présent, il nous faut apprendre comment dire que cette chaîne que nous venons de déclarer doit être utilisée comme paramètre pour le ROM_CALL : il nous faut placer son adresse sur la pile.
Pour cela, nous utiliserons, avant d'appeler le ROM_CALL, l'instruction suivante :

```
  pea.l message(pc)
```

L'instruction pea ("**P**ush **E**ffective **A**ddress") signifie que l'adresse qui suit doit être placée sur la pile.
*message* correspond au nom de notre variable.
Le "(pc)" qui est rajouté après le nom de l'étiquette n'est pas obligatoire, mais il est recommandé pour une question d'optimisation. Nous prendrons donc l'habitude de toujours l'écrire.

Une chose est très importante lors de l'exécution d'un programme en Assembleur : il faut que le pointeur de pile (le registre a7) pointe, une fois la fin du programme atteinte, sur le même endroit que lors du lancement, sans quoi le TIOS plantera une fois que le programme lui rendra le contrôle de la machine.
La méthode la plus fiable pour être certain que la pointeur de pile soit bien restauré à la fin du programme est de le restaurer à chaque fois que l'on a appelé un ROM_CALL, ou une fonction, qui

nécessitait des paramètres : une fois l'appel effectué, on restaure la valeur de a7.
Pour cela, il nous faut compter combien d'octets on a placé sur la pile lors des passages de paramètres (cela peut être fait à partir du tableau de correspondance C/ASM donné plus haut), et utiliser l'instruction lea ("**L**oad **E**ffective **A**ddress") de la fa&ccedilt;on suivante :

```
  lea nombre_d_octets(a7),a7
```

Ici, nous avons envoyé 4 octets (un paramètre en "*quelque_chose* *", soit une adresse) sur la pile en paramètre... Une fois le ROM_CALL appelé, il faudra donc exécuter l'instruction suivante :

```
  lea 4(a7),a7
```

L'appel du ROM_CALL en lui-même reste identique à ce que nous avons appris plus haut.

Pour résumer, l'appel d'un ROM_CALL nécessitant un paramètre de type chaîne de caractère se fait en trois (plus une) étapes :

- Etape 0 : Déclaration de la variable de type chaîne de caractères.
- Etape 1 : passage des paramètres sur la pile.
- Etape 2 : Appel du ROM_CALL.
- Etape 3 : Restauration du pointeur de pile.

Pour finir cette sous partie, voici un exemple, qui affichera le texte "Hello World !" dans la Status line :

```
; Assembly Source File
; Created 21/09/2002, 19:02:03

    section ".data"
    include "OS.h"
    xdef    _nostub
    xdef    _ti89
    xdef    _ti92plus

_main:
    pea.l    texte(pc)
    move.l   $C8,a0
    move.l   ST_helpMsg*4(a0),a0
    jsr      (a0)
    lea      4(a7),a7
    rts

texte: dc.b "Hello World !",0
```

Exemple Complet

Remarque : Il existe d'autres méthodes pour passer un paramètre de type "*quelque_chose* *", ainsi que pour restaurer le pointeur de pile, et vous les avez sans doute rencontrés si vous avez déjà eu l'occasion de lire d'autres tutoriaux. Les instructions que j'ai utilisé ici sont les plus parlantes de par leur nom (qui spécifient clairement qu'il s'agit de travail sur des adresses ("Push Effective Address", Load Effective Address""), mais ne sont pas forcément les plus optimisées. Nous verrons plus loin comment, et quand, une optimisation est possible... Mais, puisque ce n'est pas toujours possible, j'ai préféré commencer par vous enseigner ceci, qui fonctionne toujours. (D'autant plus d'un programme tel que celui-ci n'a pas réellement besoin d'être optimisé, puisqu'il n'a pas besoin de la plus grande rapidité qui soit (celle-ci est généralement à réserver pour les jeux), et que perdre quelques octets n'est pas gênant par rapport au gain pédagogique obtenu.).

## C: Appel d'un ROM_CALL attendant plusieurs paramètres, de type entier

Maintenant que nous avons vu les bases des appels de ROM_CALL avec paramètres, nous allons étendre notre connaissance au passage de plusieurs paramètres, et, pour varier, nous les prendrons cette fois de type entier.
Nous travaillerons ici avec le ROM_CALL `DrawLine`, qui permet de tracer une ligne entre deux points de l'écran, et qui nous permet de choisir le mode d'affichage que nous souhaitons.

Voici la façon dont ce ROM_CALL est déclaré dans la documentation de TIGCC :

```
void DrawLine (short x0, short y0, short x1, short y1, short Attr);
```

Comme vous pouvez le voir, ce ROM_CALL prend en paramètres 5 valeurs de type "short", ce qui correspond, d'après notre tableau de correspondance, à des mots de deux octets.
Nous tracerons une ligne entre le point A de coordonnées (x0 ; y0) et le point B de coordonnées (x1 ; y1), et ce dans le mode Attr. Ici, nous poserons x0=10, y0=15, x1=50, y1=60, et Attr=1 (ce qui correspond au mode normal, noir sur blanc).

Le principe à appliquer est le même que celui que nous avons suivi précédemment, à quelques détails prêt :

- On place sur la pile nos paramètres.
  Attention : ils doivent être placé sur la pile dans l'ordre inverse de leur déclaration dans la documentation, ce qui signifie que nous placerons d'abord Attr, puis y1, puis x1, puis y0, et enfin x0 !
- On appelle le ROM_CALL.
- Et on fait le ménage de la pile, afin de restaurer le pointeur de pile.

Pour placer sur la pile des données de type mot, sur deux octets, nous utiliserons l'instruction suivante :

```
  move.w  #valeur,-(a7)
```

Le dièse devant "valeur" signifie que c'est une valeur que nous voulons envoyer sur la pile. Nous verrons plus tard comment envoyer le contenu d'une variable de type entier.

Pour tracer notre ligne, il nous faudra donc écrire le code qui suit (remarquez que, pour que le dessin soit visible, nous avons rajouté un appel au ROM_CALL `ngetchx`, qui attendra que vous ayez pressé une touche avant de permettre au programme de rendre le contrôle au TIOS) :

```
; Assembly Source File
; Created 21/09/2002, 19:02:03

    section ".data"
    include "OS.h"
    xdef    _nostub
    xdef    _ti89
    xdef    _ti92plus

_main:
    move.w  #1,-(a7)    ; On a envoie sur la pile les paramètres... En ordre
inversé !
    move.w  #60,-(a7)
    move.w  #50,-(a7)
    move.w  #15,-(a7)
    move.w  #10,-(a7)
    move.l  $C8,a0
    move.l  DrawLine*4(a0),a0
    jsr     (a0)
    lea     10(a7),a7 ; On a envoyé 5 paramètres de 2 octets chacun => 10
octets au total

    move.l  $C8,a0
    move.l  ngetchx*4(a0),a0
    jsr     (a0)
    rts
```

Exemple Complet

Voila, à présent, vous savez comment appeler des ROM_CALLs...
Vous pouvez remarquer, dans le dernier exemple que nous avons étudié, que, pour appeler deux ROM_CALLs, nous sommes forcés d'exécuter deux fois l'instruction suivante :

```
  move.l  $C8,a0
```

Bien sûr, pour deux ROM_CALLs dans un programme, ce n'est pas extrêmement grave... mais, pour des programmes plus gros comme nous devons généralement en écrire (il est rare d'écrire des programmes qui ne fassent que quelques lignes !), cela devient une vrai perte de temps, et de mémoire !
Nous verrons au chapitre suivant comment éliminer ceci... en optimisant un peu nos programmes. Pour cela, nous travaillerons sur un exemple assez similaire à ceux que nous avons étudié ici ; nous utiliserons un ROM_CALL qui nécessitera à la fois une chaîne de caractère et des entiers en paramètre, ce qui nous permettra de réviser ce que nous venons d'apprendre, tout en constatant que ce n'est pas plus complexe qu'ici : il nous suffira de mélanger les deux types de paramètres, de la façon la plus naturelle qui soit.

# Chapitre 5

Maintenant que nous savons comment appeler des ROM_CALLs prenant en paramètres des entiers, ou une chaîne de caractère, nous allons apprendre à optimiser ces appels ; nous en profiterons pour montrer qu'utiliser des ROM_CALLs prenant plusieurs types d'arguments différents n'est pas plus difficile que ce que nous avons jusqu'à présent étudié.

## I:\ Appel d'un ROM_CALL avec des paramètres de types différents

Nous allons en premier lieu voir comment appeler le ROM_CALL `DrawStr`, qui vous sera probablement souvent utile, puisqu'il permet un affichage de texte à l'écran, à la position, exprimée en pixels, que vous désirez.
Comme nous l'indique la documentation de TIGCC, voici comment ce ROM_CALL est déclaré :

```
void DrawStr (short x, short y, const char *str, short Attr);
```

Le premier paramètre doit être un entier, codé sur deux octets, puisqu'il s'agit d'un short, et correspond à la position en abscisse à laquelle le message sera exprimé à l'écran. Comme pour `DrawLine`, que nous avons étudié au chapitre précédent, la position est exprimée en nombre de pixels par rapport au coin supérieur gauche de l'écran (Le pixel tout dans le coin haut gauche de la machine a pour coordonnées en abscisse 0, et même chose pour la coordonnée en ordonnée).
Le second paramètre est de même type, mais désigne la position en ordonnée, c'est à dire la ligne (toujours en pixels), ou sera affiché le message.
Le troisième paramètre est de même type que celui attendu par le ROM_CALL `ST_helpMsg`, que nous avons aussi utilisé pour exemple au chapitre précédent. Il désigne la chaîne de caractères que nous souhaitons afficher à l'écran, et s'utilise exactement comme vu plus tôt.
Enfin, le quatrième et dernier paramètre est un entier, comme les deux premiers, et peux prendre cinq valeurs différentes, selon la façon dont nous voulons afficher notre message ; ci-dessous, un tableau de correspondance entre les valeurs et les modes d'affichage (à titre d'information, puisque c'est ce qui est utilisé dans la documentation de TIGCC, nous indiquons, pour chaque valeur, le code correspondant utilisé en langage C ; nous n'en tiendrons pas compte ici, mais vous saurez à quelles valeurs ils correspondent, si vous les "croisez" dans la documentation):

| ASM | C | Résultat |
|---|---|---|
| 0 | A_REVERSE | Le texte est affiché en blanc sur fond noir. |
| 1 | A_NORMAL | Le texte est affiché en noir sur fond blanc. |
| 2 | A_XOR | Le texte est dessiné en couleurs inversant le fond (là où le texte est noir et le fond aussi, ce sera affiché en blanc, par exemple : c'est un OU exclusif). |
| 3 | A_SHADED | Le texte est dessiné en "ombré" ; seuls la moitié des pixels le constituant sont affichés. |
| 4 | A_REPLACE | Le texte est écrit en effaçant l'arrière plan. |

Pour ce ROM_CALL comme pour tous les autres, il convient de placer les paramètres sur la pile, en

ordre inverse de l'écriture C.
Nous allons afficher les texte "Salut !" à partir du pixel dont le couple de coordonnées est [10;25], en mode normal (qui a pour code 1), ce qui correspond à une écriture en noir sur blanc. Pour laisser à l'utilisateur du programme le temps de lire le message, nous attendrons un appui sur une touche à l'aide du ROM_CALL `ngetchx` avant de rendre le contrôle au TIOS.
Pour cela, nous agirons comme nous l'avons déjà fait au chapitre précédent : la seule différence est que, ici, nous alternons les types de paramètres. C'est pour cela que nous ne fournirons pas d'explications détaillées sur notre exemple, que vous trouverez ci-dessous. Si vous ne comprenez pas exactement ce qu'il fait, nous vous conseillons de réétudier le chapitre précédent de ce tutorial, car comprendre, et maîtriser les appels de ROM_CALL, et donc, le passage de paramètres, est indispensable à la programmation en langage d'Assembleur, du moins pour l'usage que nous ferons de ce langage dans ce tutorial.
(Si, une fois la lecture de ce langage terminée, vous continuez pendant pas mal de temps la programmation en ASM, vous écrirez grand nombre de fonctions, qui s'appellent généralement de la même façon que les ROM_CALL, et vous aurez encore besoin de ce que nous apprenons ici.)

Voici le code source du programme permettant d'afficher notre petit message :

```
; Assembly Source File
; Created 11/02/2003, 23:18:07

    section ".data"
    include "OS.h"
    xdef    _nostub
    xdef    _ti89
    xdef    _ti92plus

_main:
    move.w  #1,-(a7)          ; Mode d'affichage
    pea.l   texte(pc)         ; Le message qu'on veut afficher
    move.w  #25,-(a7)         ; Ordonnée
    move.w  #10,-(a7)         ; Abscisse
    move.l  $C8,a0
    move.l  DrawStr*4(a0),a0
    jsr     (a0)
    lea     10(a7),a7

    move.l  $C8,a0
    move.l  ngetchx*4(a0),a0
    jsr     (a0)

    rts

texte:      dc.b    "Salut !",0
```

Exemple Complet

# II:\ Une petite optimisation... pas si petite que cela !

Comme nous l'avons déjà souligné à la fin du chapitre précédent, et comme vous avez de nouveau pu le constater dans l'exemple que nous venons de prendre, à chaque fois que nous avons appelé un ROM_CALL, nous avons du au préalable exécuter l'instruction suivante :

```
move.l  $C8,a0
```

Cela n'est certes pas gênant pour deux ROM_CALLs, mais cela le deviendra pour un plus grand nombre : d'une, cela prend du temps à écrire, de deux, ça augmente la taille du programme, et, de trois, cela ralenti le programme (puisque chaque instruction prend de la place en mémoire et nécessite un petit temps pour s'exécuter).
L'optimisation que nous allons ici présenter sera plus une pessimisation qu'autre chose pour un exemple aussi bref, mais elle sera absolument non négligeable pour des programmes un peu plus long ; je vous conseille donc de l'utiliser aussi souvent que possible (Selon toute logique, vous ne vous limiterez pas à des programmes ne comportant que deux ou trois ROM_CALL !).

L'optimisation est simple : l'astuce est de mémoriser $C8 dans un registre (comme nous le faisions à présent, dans a0), mais pas a0. Nous allons maintenant voir pourquoi... et quelles sont les opérations à effectuer en plus de cela, pour laisser la calculatrice dans un état stable une fois l'exécution du programme terminée.

## A: Registres modifiables... et registres modifiés

Registres que le prog a le droit de modifier
=> Les RC ont le droit de modifier les mêmes
=> On ne peut pas stocker d'information dans ces registres entre plusieurs appels de RC
=> Il va falloir en utiliser un autre

## B: Sauvegarde, puis restauration de registres

Comment faire pour sauvegarder un registre.
Comment faire pour le restaurer

## C: Enfin, l'optimisation

A présent, nous pouvons voir l'optimisation effective.

Pour stocker $C8, il nous faut, comme a0, un registre d'adresse. Puisque nous ne pouvons pas utiliser les registres de bas numéros, qui peuvent être effacés par les ROM_CALL, ni le registre a7, qui est le pointeur de pile (le modifier sans faire grandement attention entraînera un plantage quasi-immédiat ! ), il nous faut en choisir un de numéro "intermédiaire".
Nous pourrions choisir d'utiliser le registre a6... mais il est souvent utilisé pour autre chose, que ce soit en C ou en ASM...
Donc, par convention, si l'on peut aller jusqu'à dire cela, nous utiliserons le registre a5. Ce parce que c'est celui-ci qui est utilisé par TIGCC lorsque l'on programme en C, et parce que c'est celui que les programmeurs ASM ont l'habitude d'employer. En utilisant ce registre, vous rendrez votre programme plus facilement compréhensible par d'autres programmeurs.

```
; Assembly Source File
; Created 11/02/2003, 23:18:07

    section ".data"
    include "OS.h"
    xdef    _nostub
    xdef    _ti89
    xdef    _ti92plus

_main:
    move.l  a5,-(a7)          ; Sauvegarde du registre a5

    move.l  $C8,a5

    move.w  #1,-(a7)          ; Mode d'affichage
    pea.l   texte(pc)         ; Le message qu'on veut afficher
    move.w  #25,-(a7)         ; Ordonnée
    move.w  #10,-(a7)         ; Abscisse
    move.l  DrawStr*4(a5),a0
    jsr     (a0)
    lea     10(a7),a7

    move.l  ngetchx*4(a5),a0
    jsr     (a0)

    move.l  (a7)+,a5          ; Restauration du registre a5

    rts

texte:    dc.b    "Salut !",0
```

Exemple Complet

# Conclusion

Cette page n'a pas encore été rédigée.
Conclusion du tutorial ASM