# 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 :

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