# La Pile



**En assembleur, le première chose qu'on manipule, c'est la pile. Voici comment utiliser la pile et voir comment les gens de HP ont programmé les fonctions internes tels le SWAP, le DUP ou le DROP pour les plus simples...**

## Préliminaires

Quand on pose un objet sur la pile, ce n'est pas véritablement l'objet qu'on mais l'adresse où il est stocké en mémoire. C'est à dire que si on pose un objet de 10 Ko sur la pile et que vous réalisez un **DUP**, la mémoire ne diminuera pas de 10 Ko supplémentaires, mais de 2,5 octets puisqu'une adresse ne consomme que 5 quartets. Ceci pour montrer qu'on ne manipule les objets directement, mais bel et bien les adresses des objets. Enfin, rappelons que le pointeur D1 pointe toujours l'adresse du premier objet de la pile, et que D champ A contient le nombre de quartets libres en mémoire. Avec ça, il est possible de gérer à peu près n'importe quoi sur la pile.

## SWAP

En imaginant que nous ayons deux objets sur la pile (sur les niveaux 1 et 2), on va tenter de réaliser l'instruction **SWAP** sans s'occuper de la présence des objets (chose qu'on verra dans une autre fiche). Comme le pointeur D1 pointe toujours le niveau 1 de la pile quand il est utilisé par le système, il suffit simplement d'inverser les deux adresses des deux premiers niveaux comme ceci :

```
"A=DAT1.A   % on charge l'adresse de l'objet 1
 D1+5       % on se déplace sur l'objet 2 avec D1
 C=DAT1.A   % on charge l'adresse de l'objet 2
 DAT1=A.A   % on écrit l'adresse de l'objet 1
 D1-5       % on recule pour revenir sur l'objet 1
 DAT1=C.A   % on écrit l'adresse de l'objet 2
 RPL        % on retourne au RPL
@"
```

Nous avons déjà dit que le pointeur D1 pointe les adresses des objets, c'est pourquoi on travaille sur le champ A (5 quartets), et c'est aussi pourquoi on avance de 5 quartets pour passer à l'adresse suivante. En fait, les adresses des objets placés sur la pile sont mises les unes derrière les autres. Dernière chose pour cet exemple, comme nous ne modifions pas les registres et pointeurs utilisés par le système, il n'est pas nécessaire de sauver les registres au début et les restaurer au final.

## Infos

**adresse** : donne la position d'un objet en mémoire. Une adresse est définie sur 5 quartets.

**champ** : espace de 1 à 16 quartets avec lequel on travaille lorsqu'on utilise les registres (ex. : A est un champ de 5 quartets).

**DUP** : fonction RPL permettant de copier l'objet placé sur le niveau 1 de la pile.

**objet** : terme générique pour désigner n'importe quoi : réel, entier, matrice, chaîne de caractères, library data...

**pointeur** : au nombre de deux (D0 et D1), ils permettent de pointer à des adresses en mémoire.

**quartet** : composé de 4 bits (0 à F en hexadécimal).

**registre** : permet de contenir des données (16 quartets). Deux types de registres existent : les registres de calcul (A, B, C et D) et les registres de sauvegarde (R0 à R4).

**SWAP** : fonction RPL qui inverse la position des deux premiers objets de la pile.

**%** : dans une source, remarquons que chaque commentaire se place derrière le caractère %.

Philippe Pamart © 200
[phpamart@nordnet.fr]
http://home.nordnet.fr/~phpamart

Pour utiliser cette source, il suffit de l'assembler avec la commande **ASM**, la stocker sous le nom de votre choix et l'exécuter comme un simple programme RPL. Attention : bien poser des objets sur les deux premiers niveaux de la pile pour éviter un sévère plantage !!!
A noter qu'on aurait pu écrire cette source en mettant plusieurs instructions par ligne, de cette façon :

```
"A=DAT1.A D1+5 C=DAT1.A   % on lit les adresses
 DAT1=A.A D1-5 DAT1=C.A   % on les écrit dans l'ordre inverse
 RPL                      % on retourne au RPL
@"
```

Sur le même principe, tentez de reconstruire la fonction **ROT**.

## DUP

Autre commande RPL, nous allons voir comment réaliser une fonction **DUP**, soit la copie de l'objet placé sur le niveau 1 de la pile. Pour cela, il faut dire à la machine qu'on réserve 5 quartets de mémoire supplémentaire en décrémentant le registre D. Ensuite, on lit l'adresse de l'objet placé sur le niveau 1 de la pile, et on la recopie sur le niveau "zéro" de la pile, ou plutôt le nouveau niveau 1 puisqu'on décale tout !

```
"D-1.A      % on décrémente la mémoire restante
 A=DAT1.A   % on lit l'adresse de l'objet à copier
 D1-5       % on se place sur le nouveau niveau 1
 DAT1=A.A   % et on écrit l'adresse
 RPL        % retour au RPL
@"
```

Sur le même principe, tentez de reconstruire la fonction **DUP2**.

## DROP

Dernière commande RPL intéressante, voici comment réaliser une fonction **DROP**, c'est à dire effacer l'objet placé sur le niveau 1 de la pile. Dans cet exemple, nous ne testerons pas la présence d'objet sur la pile, donc veillez à ce qu'il y ait bien un objet au moins sur la pile, sinon un “Try to recover memory” arrivera... Avec cette commande, nous avons l'inverse de la commande **DUP**, c'est à dire qu'on va incrémenter la mémoire libre et remonter le pointeur de pile D1 d'un étage, exactement comme ceci :

```
"D1+5       % on se place sur le nouveau niveau 1
 D+1.A      % on décrémente la mémoire restante
 RPL        % retour au RPL
@"
```

Sur le même principe, tentez de reconstruire la fonction **NIP**, fonction qui cache la séquence **SWAP DROP**.

### Infos

**ASM** : fonction qui permet d'assembler la source pour obtenir un exécutable de type **Code**.

**DROP** : fonction RPL qui efface l'objet du niveau 1 de la pile.

**DUP2** : fonction RPL qui copie les deux premiers objets posés sur la pile.

**NIP** : fonction RPL qui efface l'objet du niveau 2 de la pile.

**ROT** : fonction RPL qui déplace l'objet du niveau 3 vers le niveau 1 de la pile.

**RPL** : Reverse Polish Language pour la petite traduction qui est le langage inverse polonais in french.


Philippe Pamart © 200[illegible]
[phpamart@nordnet.fr]
http://home.nordnet.fr/~phpamart

## Lire et écrire un objet sur la pile

Imaginons que nous ayons un objet de type entier système (System Binary) sur le niveau 1 de la pile qui a une valeur quelconque. La structure d'un tel objet est la suivante :
`11920xxxxx` où `xxxxx` est le contenu. C'est à dire que l'entier système `# 5h` est construit ainsi : `1192050000`. Ici, on remarque que même en assembleur il est nécessaire de saisir les données à l'envers !
Donc, pour lire le contenu de cet objet, il suffit de lire son adresse, de pointer l'objet et d'avancer de 5 quartets pour pouvoir charger ce contenu. Simple, non ?
Si maintenant on désire écrire (ou modifier) cette valeur, il suffit de réécrire à l'endroit où on se trouve.
Comme exemple, nous allons incrémenter de 1 l'entier système placé sur le niveau 1 de la pile.

```
"SAVE       % on sauve registres et pointeurs
 A=DAT1.A   % on charge l'adresse de l'entier système
 D0=A       % on pointe le début de cet entier système
 D0+5       % on pointe maintenant le contenu
 A=DAT0.A   % on charge le contenu pointé par D0 sur 5 quartets
 A+1.A      % on l'incrémente
 DAT0=A.A   % et on écrit la nouvelle valeur
 LOADRPL    % récupération des registres et pointeurs, et retour au RPL
@"
```

Une fois la source assemblée avec **ASM**, vous pouvez incrémenter sans soucis l'entier système placé sur le niveau 1 de la pile. Petite remarque : comme on modifie directement l'objet, il n'est pas possible d'utiliser **UNDO** pour récupérer le précédent résultat. Pour pouvoir le faire, il faut utiliser la commande **NEWOB** pour recréer l'objet avant de le modifier.
Comme exercice, faîtes l'addition de deux entiers système posés sur les deux premiers niveaux de la pile.

## Notes

### Infos

**H→** : permet de tranformer une chaîne de caractère en son objet. Pour transformer la chaîne correspondant à un entier système, il suffit de faire `"1192050000" H→`.

**NEWOB** : commande RPL qui permet de recréer un objet à une adresse différente.

**UNDO** : sous cette touche ce cache la commande **LAST** ou **LASTARG** qui permet de récupérer les derniers objets avant une quelconque action.