Astuces de programmation!

# ROUTINES D'ACCÈS-MÉMOIRE

## DOKE & DEEK

Cette série de commandes RSX vous permettra, d'abord d'apprendre à créer ce type de commandes si vous l'ignorez, et surtout d'avoir à votre disposition des routines en langage machine supprimant la lenteur du Basic et de nombreux POKE et PEEK assez lourds à utiliser.

L'un des défauts du Basic de l'Amstrad est de ne pas posséder ces deux fonctions qui sont pourtant très utiles lors des accès directs à la mémoire. En effet, il peut être utile d'accéder en une seule fois à deux adresses mémoires consécutives, ne serait-ce que pour stocker une adresse ou un score dans le cas d'un jeu. Ces deux RSX nous permettront, en plus de leur intérêt non négligeable en programmation, de faire un petit rappel sur la création d'instructions supplémentaires et sur l'accès aux vàriables Basic depuis le langage machine.

### Connaissance nécessaires pour la constitution de RSX

Pour créer vous-même ce type d'instructions, vous devez posséder un minimum de connaissances en Assembleur Z80, c'est-à-dire être capable de créer des routines, même simples, en Assembleur. De plus, il n'est pas interdit de connaître certaines adresses mémoire utiles, au contraire. Si vous en manquez, nous vous recommandons des ouvrages tels que les *"Clefs pour Amstrad"* de chez PSI, ou *"La Bible du CPC"* de Micro-Application, qui combleraient ce manque.

### Programmation d'une RSX

Tout d'abord, il faut trouver un espace mémoire suffisamment grand pour stocker la routine. Etant donné la nécessité de travailler avec un Assembleur tel que DAMS ou ZEN, il ne faut pas perdre sur son espace mémoire. Ceci étant dit, passons à la programmation.

Une adresse primordiale est &BCD1 qui permet, grâce à une routine ROM, de créer une RSX. Son utilisation est très simple : il suffit, avant l'appel de cette adresse, de charger dans HL l'adresse d'une zone mémoire libre de quatre octets, et de charger dans BC l'adresse du ou des vecteurs de saut (HL et BC sont des registres du Z80, donc inaccessibles directement depuis le Basic). A la première adresse des vecteurs de saut doit figurer l'adresse (encore une !) de la table des noms de ou des instructions à créer.
Cela nous donne dans notre cas :
ORG &9000 : début du programme en &9000.
LD hl, tampon : charge dans HL l'adresse des 4 octets libres.
LD bc,vect : charge dans BC l'adresse de la table des vecteurs.
CALL &BCD1 : positionne les nouvelles instructions RSX.
RET : fin de la constitution.
tampon DEFS 4 : réserve 4 octets libres.
vectv DEFW table : fixe l'adresse de la table des noms.
JP doke : vecteur de la routine DOKE.
JP deek : vecteur de la routine DEEK.
table DEFM DOK : définition du nom de l'instruction.
DEFB &80+"E" : dernière lettre du nom.
DEFM DEE : idem pour DEEK.
DEFB &80+"K"
DEFB 0 : fin de la table des noms, suite du programme.

Attention, les noms des instructions doivent être en majuscules, car c'est sous cette forme que le Basic transcrit tous vos ordres, donc, si vos noms sont en minuscules, ils ne seront pas reconnus, et à l'appel de vos RSX vous obtiendrez un "UNKNOWN COMMAND".
Pour valider vos nouvelles instructions, vous devez depuis le Basic faire un CALL à l'adresse donnée par le ORG, CALL &9000 dans notre cas.

### Les paramètres

Une instruction RSX peut fort bien être accompagnée d'un ou de plusieurs paramètres. Dans notre cas, les commandes DOKE et DEEK doivent être accompagnées de deux paramètres. Lorsque l'ordinateur exécute une commande RSX, les registres A et IX sont très utiles. Le registre A contient le nombre de paramètres accompagnant la commande (0 s'il n'y en a aucun). On peut donc dans la routine Assembleur tester si ce nombre est correct ou non pour la suite. Le registre IX pointe sur la pile des paramètres. Ceux-ci étant codés sur deux octets, donc entiers, la valeur totale se trouve à l'adresse pointée par IX et celle pointée par IX + 1 (on voit déjà ici l'intérêt des DOKE et DEEK). L'adresse du second paramètre, s'il existe bien sûr, sera pointée par IX + 2 et par IX + 3 et ainsi de suite (remarque : IX pointe sur

l'octet bas du nombre, et IX + 1 pointe sur l'octet haut).
Nous avons donc maintenant tous les éléments nécessaires à la programmation du DOKE (voir le programme ci-après). Le DEEK est un peu plus complexe à réaliser. En effet, le deuxième paramètre est en fait l'adresse d'une variable entière (telle que a% par exemple). Nous obtenons cette adresse en Basic en faisant PRINT HEX$( @ a%), c'est pourquoi la syntaxe de notre commande sera |DEEK,adresse,à variable entière. Après appel de cette commande, la variable choisie contiendra la valeur donnée en Basic par Print HEX$(PEEK(adresse) + 256 × peek(adresse + 1)). Notre routine fonctionnera donc de la même façon que le DOKE, mis à part qu'il s'agira d'une lecture de la mémoire et non pas d'une écriture. Nous stockerons donc notre résultat directement dans la variable, ce qui est très facile puisque nous connaissons son adresse. Il est bien sûr possible d'écrire directement dans cette variable en Basic grâce à notre nouvelle instruction DOKE que nous venons de créer, mais est-ce vraiment utile ? Vous pouvez toujours essayer à titre d'exercice. Si vous n'y parvenez pas, voici quand même la solution : |DOKE, @ variable entière,nombre entier.

A propos, lorsque vous utiliserez la variable après le DEEK, ne vous étonnez surtout pas si vous obtenez un résultat négatif car le codage sur deux octets est du binaire signé, c'est-à-dire positif ou négatif suivant la valeur 0 ou 1 du bit de gauche de l'octet se trouvant à l'adresse la plus basse (vous avez suivi j'espère !).

Voilà, maintenant, à vos claviers, avec un peu d'entraînement, vous pourrez créer vous-même votre propre "bibliothèque" de RSX. Surtout, lisez bien les commentaires accompagnant le programme Assembleur, ils vous seront très utiles pour bien comprendre le fonctionnement de la routine.

ORG &9000 : adresse de début du programme, donc CALL &9000 pour valider les RSX.

LD hl,tampon : voir texte.
LD bc,vect : voir texte.
CALL &BCD1 : appelle la routine ROM validant les RSX.
RET : fin de la définition des RSX.
tampon DEFS 4 : réserve 4 octets libres.
vect DEFW table : voir texte.
JP doke : vecteur de la routine doke.
JP deek : vecteur de la routine deek.
table DEFM DOK : définition du nom de l'instruction.
DEFB &80 + "E" : dernière lettre du nom.
DEFM DEE : voir ci-dessus.
DEFB &80 + "K"
DEFB 0 : fin de la table des noms.
doke CP &2 : 2 paramètres ?
RET NZ : fin si non.
LD e,(IX + 0) : octet bas du nombre dans e.
LD d,(IX + 1) : octet haut du nombre dans d.
LD l,(IX + 2) : octet bas de l'adresse dans l.
LD h,(IX + 3) : octet haut de l'adresse dans h.
LD (hl),e : octet bas du nombre dans l'adresse.
INC hl : hl = hl + 1.
LD (hl),d : octet haut du nombre dans l'adresse + 1.
RET : fin du doke.
deek CP &2 : voir doke.
RET NZ
LD 1,(IX + 0) : octet bas de l'adresse de la variable dans l.
LD h,(IX + 1) : octet haut de l'adresse de la variable dans h.
LD e,(IX + 2) : octet bas de l'adresse dans e.
LD d,(IX + 3) : octet haut de l'adresse dans d.
LD bc,&7F0E : déconnexion de la ROM basse.
OUT (c),c : voir CAHIERS D'AMSTRAD N° 5 p. 13.
LD a,(de)
LD (hl),a
INC hl : on passe à l'adresse suivante.
INC de
LD a,(de)
LD (hl),a
RET : fin de la routine deek et du programme.

### Rappel de la syntaxe des commandes

|DOKE,adresse,nombre entier et |DEEK, adresse, @ variable entière. N'oubliez pas de sauvegarder le programme avant de la lancer par un CALL &9000, car une erreur de saisie pourrait être fatale et vous devriez tout recommencer.

```
10 REM ***** CHARGEUR DE RSX:    |DO [1092]
KE   &  |DEEK
20 ON ERROR GOTO 120:MODE 2          [1818]
30 nb=0:FOR ad=&9000 TO &904C        [1367]
40 READ da$:da=VAL("&"+da$):nb=nb+d [2401]
a:POKE ad,da
50 NEXT                              [350]
60 IF nb<>7824 THEN PRINT CHR$(7):  [4373]
LOCATE 15,10:PRINT "Erreur dans les
 DATA":END
70 LOCATE 21,10:PRINT"Pressez [S] p [3798]
our la sauvegarde"
80 WHILE k$="":k$=INKEY$:WEND        [2180]
90 IF UPPER$(k$)="S" THEN SAVE "dok [1994]
edeek.bas"
100 CALL &9000                       [445]
110 END                              [110]
120 PRINT CHR$(7):e=ERR:IF e<>13 AN [2763]
D e<>6 THEN 160
130 nld=210+10*((ad-&9000)\8)        [1437]
140 PRINT"Erreur dans la DATA No";1 [5418]
+(ad-&9000) MOD 8;" ligne ";nld
150 END                              [110]
160 IF e=4 THEN PRINT "Erreur en li [11284]
gne 30, ou il manque une ou plusieu
rs DATA":PRINT "Utilisez le verific
ateur V2.":END
170 PRINT "Erreur quelconque ligne  [3638]
";ERL;"."
180 PRINT                            [361]
190 END                              [110]
200 REM  ---------------- DATA ---  [2176]

------------------
210 DATA 21,0a,90,01,0e,90,cd,d1     [1458]
220 DATA bc,c9,00,00,00,00,16,90     [816]
230 DATA c3,1f,90,c3,32,90,44,4f     [1175]
240 DATA 4b,c5,44,45,45,cb,00,fe     [1189]
250 DATA 02,c0,dd,5e,00,dd,56,01     [1543]
260 DATA dd,6e,02,dd,66,03,73,23     [1016]
270 DATA 72,c9,fe,02,c0,dd,6e,00     [1393]
280 DATA dd,66,01,dd,5e,02,dd,56     [1188]
290 DATA 03,01,0e,7f,ed,49,1a,77     [1924]
300 DATA 23,13,1a,77,c9              [797]
```