# TRUCS ET ASTUCES

par André Chéramy

**5) Accéder au ê du Sédoric**

L'option ACCENT SET du sédoric permet l'utilisation des caractères "accentués" suivant:

**à** de code ASCII 64 (#40) en remplacement de @
**ç** de code ASCII 92 (#5C) en remplacement de \
**é** de code ASCII 123 (#7B) en remplacement de {
**ù** de code ASCII 124 (#7C) en remplacement de |
**è** de code ASCII 125 (#7D) en remplacement de }
**ê** de code ASCII 126 (#7E) en remplacement de ~

Mais voilà, le caractère de code 126 n'est pas accessible au clavier! Il faut redéfinir une touche de fonction, c'est à dire faire:

**VUSER** pour voir quelles définitions utilisateurs sont actuellement en usage. Supposons que la touche de fonction n° 6 soit libre.

**KEYDEF 6** puis appuyer sur **FUNCT** et sur ^.

**KEYUSE 6,CHR$(126)**.

**KEYSAVE"ACCENT.KEY"** (n'importe quel nom est valable).

Par la suite, il faut charger ACCENT.KEY (par exemple en ajoutant !LOAD"ACCENT.KEY" à l'INIST de la disquette). L'appui sur FUNCT et ^ affichera un **ê**.

**6) TRUE FALSE AND OR NOT**

Voilà des commandes trop peu utilisées! Et pourtant elles peuvent rendre bien des services:

**TRUE** est une simple constante dont la valeur est **1111 1111** en binaire soit #FF en hexadécimal ce qui correspond à -1 en langage courant (entier signé). On peut vérifier que PRINT TRUE ou PRINT ("B">"A") affichent -1.

**FALSE** est aussi une constante de valeur **0000 0000** en binaire, soit #00 en hexadécimal et 0 en langage courant. On peut vérifier que PRINT FALSE ou PRINT ("B"<"A") affichent 0.

Ces deux constantes sont utilisées comme drapeau lors d'un test pour savoir si le résultat est vrai (TRUE) ou faux (FALSE).

**AND**, **OR** et **NOT** effectuent des **opérations bit à bit** sur des nombres exprimés en binaires de 8 bits numérotés de 0 à 7 de droite à gauche. On ne peut pas comprendre AND OR et NOT sans passer par le binaire.

```
Exemple avec AND:        12          #0C          0000 1100
                  AND     9    AND  #09    AND   0000 1001
                  ----------    -----------    --------------------
                  donne   8    donne #08    donne  0000 1000

Exemple avec OR:         13          #0D          0000 1101
                  OR     24    OR   #18    OR    0001 1000
                  ----------    -----------    --------------------
                  donne  29    donne #1D    donne  0001 1101
```

| | | | |
|---|---|---|---|
| Exemple avec NOT: | 7 | #07 | 0000 0111 |
| NOT 7 | donne 248 | #F8 | 1111 1000 |

On voit que aussi bien avec AND qu'avec OR ou qu'avec NOT, il n'y a aucune logique lorsque les chiffres sont exprimés en décimal ou en hexadécimal. Mais lorsqu'ils sont exprimés en binaire et que l'on prend les bits un à un, on voit que avec AND seul le bit n 4 est à 1 dans les deux nombres de départ: **1 AND 1 donne 1**. Avec OR seul **0 OR 0 donne 0** enfin avec NOT chaque bit est inversé: **NOT 0 donne 1** et **NOT 1 donne 0**. On comprend donc que NOT FALSE donne TRUE et que NOT TRUE donne FALSE. Que seul TRUE AND TRUE donne TRUE alors que les autre combinaisons (exemple FALSE AND TRUE) donnent FALSE. A l'opposé seul FALSE OR FALSE donne FALSE, les autres combinaisons (exemple FALSE OR TRUE) donnent TRUE. Il est important de savoir que **pour la commande IF...THEN...[ELSE], toute valeur différente de zéro est considérée comme TRUE**:

IF (2+3) THEN PING fera entendre un ping!

Pour s'en sortir avec AND OR et NOT, il n'y a par d'autre moyen que de traiter l'opération en binaire bit à bit. Pour exprimer un nombre en binaire, si vous n'avez ni calculatrice scientifique, ni table de conversion, demandez un **PRINT HEX$()** à votre Oric/Atmos. Par exemple, pour convertir le nombre 26, faire PRINT HEX$(26), ce qui donnera #1A. Le 1 représente les 4 digits de gauche et s'écrit 0001 en binaire. Le A représente les 4 digits de droite et s'écrit 1010 en binaire. Le nombre 26 s'écrit finalement 0001 1010 en binaire. NOT 26 donne donc 1110 0101 soit #E5 en Hexadécimal et PRINT #E5 affiche 229. Encore un exemple pour montrer qu'**en dehors du binaire, il n'y a pas de logique**: Si A=1 NOT A donne -2 (on peut vérifier que PRINT NOT 1 affiche -2). En effet, 1 s'écrit 0000 0001 en binaire donc NOT 1 donne 1111 1110 soit #FE en hexadécimal, ce qui correspond à -2 en langage courant! On a ici un exemple qui semble particulièrement dément puisque **pour IF..THEN la valeur 1 qui est différente de zéro est vraie** et que NOT TRUE donne FALSE alors que NOT 1 donne -2 qui pour IF...THEN est toujours une valeur TRUE! La commande **IF NOT 1 THEN PING** n'est dont pas équivalente à la commande **IF NOT TRUE THEN PING**. La première sonne le ping l'autre pas!

Enfin, dernière **utilisation de AND et NOT**: pour forcer un bit à 0 ou à 1. Exemple à l'adresse #26A de la RAM se trouve un registre d'état de la console. Chacun des bits est un drapeau précis. Le bit n 0 (le plus à droite) sert à déterminer si le curseur doit être visible (s'il est à 1) ou pas (s'il est à zéro). J'ai déjà indiqué qu'un **POKE#26A,(PEEK(#26A)AND#FE)** permet d'effacer le curseur de manière certaine, tandis qu'un **POKE#26A,(PEEK(#26A)OR#01)** permet de l'afficher à coup sûr. Voici pourquoi et comment: Supposons que l'adresse #26A contienne la valeur 0000 1110, le curseur est caché. si l'on fait:

OR 0000 0001, c'est à dire POKE#26A,(PEEK(#26A)OR#01),

on obtient 0000 1111, ce qui revient à forcer le bit n 0 à 1 **sans affecter les autres bits qui restent comme ils étaient!** Supposons maintenant que nous voulions cacher à nouveau le curseur, on a toujours 0000 1111, on fait:

AND 1111 1110, c'est à dire POKE#26A,(PEEK(#26A)AND#FE),

on obtient 0000 1110, ce qui revient à forcer le bit n 0 à 0 **sans affecter les autres bits!**

Pour forcer tel bit à 0 ou 1, il suffit de connaître la valeur des différents bits:

n 0 vaut 1
n 1 vaut 2
n 2 vaut 4
n 3 vaut 8
n 4 vaut 16
n 5 vaut 32
n 6 vaut 64
n 7 vaut 128.

Ainsi pour forcer le bit n 5 à 1 faire OR 0010 0000, c'est à dire OR 32. Pour forcer ce même bit n 5 à 0 faire AND 1101 1111, c'est à dire AND 223 (soit 255-32).