# Partie 9 : La vidéo

## 9-1 Introduction

La vidéo, c'est l'ensemble des fonctions qui permettent l'affichage d'informations sur l'écran du CPC. Il y a 3 éléments de base qui permettent cet affichage :

1. **La mémoire vidéo :** c'est la zone de mémoire RAM dans laquelle sont mémorisés tous les points de l'écran (pixels) avec leur couleur. Par défaut elle se situe en &C000 et prend 16ko (jusqu'à &FFFF donc).

2. **Le VGA (Vidéo Gate Array) :** c'est un composant aux fonctions multiples, mais qui permet entre autre la définition du mode graphique et des couleurs.

3. **Le CRTC (Cathode Ray Tube Controller) :** c'est le circuit chargé de générer tous les signaux nécessaires à la production des images vidéos. Chose intéressante, il est paramétrable.

## 9-2 Résolution des modes graphiques

Que ce soient les modes graphiques 0,1 ou 2, ceux-ci occupent invariablement 16000 octets visibles. Mais &FFFF-&C000 ça ne fait pas 16ko soient 16384 octets ? où sont donc passés les 384 octets non visibles ? 1ère bizarrerie (on n'est pas au bout de nos peines) ! mais d'abord un petit rappel sur ces modes graphiques :

- **Mode 0** : l'écran fait 160x200 pixels en 16 couleurs. C'est le mode le plus coloré, même si en contrepartie les pixels paraissent étirés en longueur. Les couleurs sont codées sur 4 bits (d'où les 16 couleurs), donc l'écran fait bien 160x200x1/2 octets = 16000 octets. En mode texte, il permet l'affichage de 20x25 caractères de 8x8 pixels (20x25x8x8x1/2 octet = 16000 donc on s'y retrouve).
- **Mode 1** : l'écran fait 320x200 pixels en 4 couleurs. C'est le mode intermédiaire, avec des pixels un peu plus carrés mais avec moins de couleurs. Celles-ci sont codées sur 2 bits et 320x200x1/4 octet = 16000 octets toujours. Le mode texte permet l'affichage de 40x25 caractères de 8x8 pixels.
- **Mode 2** : l 'écran fait 640x200 pixels en 2 couleurs. C'est la plus haute résolution, mais la moins colorée, elle convient donc plus au texte. Dans ce cas, il est possible d'afficher 80x25 caractères de 8x8 pixels. Les pixels sont codés sur 1 bit seulement et 640x200x1/8 octet est bien égal à 16000 octets.

Pourtant malgré ces différentes résolutions, le système (que ce soit en basic ou avec les vecteurs graphiques) semble ne considérer qu'une seule résolution : le 640x400 ! et en mode 0 vous aurez donc du 640x400 16 couleurs (vous en rêviez AMSTRAD l'a fait !). Pour vous en persuader, tapez le programme BASIC suivant :

```
10 FOR m = 0 to 2
20 MODE m : PLOT 320,200
30 CALL &BB18
40 NEXT m
```

Au passage, le vecteur &BB18 attend simplement une touche (pour changer du inkey$). Vous aurez donc un point placé au milieu de l'écran quelque soit le mode. Mais ce point étant de taille variable, vous aurez toute suite démonté la supercherie. En effet, les résolutions réelles restent de 160x200 en mode 0, de 320x200 en mode 1 et de 640x200 en mode 2 (mais oui on l'aime comme il est notre CPC, pas besoin d'en rajouter).

## 9-3 Les vecteurs vidéo

Comme nous l'avons vu dans le chapitre 6, le système met à disposition tout un tas de sous-programmes appelés vecteurs. Ils font la distinction entre le texte, le graphique et l'écran, mais cette distinction est purement logicielle. Pour avoir la totalité de la liste des vecteurs, reportez-vous au document du firmware (en Anglais seulement par contre).

- **Vecteurs textes** (&BB4E à &BBB7) : affichage, lecture ou redéfinition de caractères. Définition de fenêtres d'affichage ou positionnement de curseur. Définition des encres de papier et de stylos.
- **Vecteurs graphiques** (&BBBA à &BBFC) : affichage ou lecture de points, traçage de lignes, définition de fenêtre graphique ou positionnement de curseur. Définition des couleurs.
- **Vecteurs écran** (&BBFF à &BC62) : initialisation d'écran, de mode ou d'encres. Calculs de coordonnées de point ou de lignes. Définition ou lecture des encres graphiques. Dessin de points, lignes ou rectangles. Scrolling d'écran ou de matrice de caractère.
- **Vecteurs bas niveaux** (&BD3A à BDED) : vecteurs généralement appelés par ceux ci-dessus. Ils en possèdent certaines fonctions, mais avec des paramètres  plus proches de la structure de la mémoire

vidéo. Quand on maîtrise cette structure, il peut être plus rapide de faire appel à eux directement.

Comme vous le voyez, les vecteurs suffisent généralement à faire tout ce qu'il y a à faire au niveau graphique. Mais ils manquent de rapidité et s'avèrent vite inefficaces dans les jeux d'action par exemple. Dans ce cas, il faut directement s'adresser au système tel qu'il est présenté ci-dessous.

### 9-4 Organisation de la mémoire vidéo

Comme je l'ai dit dans l'introduction, la mémoire vidéo est le bloc de RAM &C000-&FFFF. Essayons maintenant de la remplir octet par octet. Et pour mieux apprécier le résultat, on va le faire en BASIC :

```
10 MODE 1
20 FOR adr = &C000 to &FFFF
20 POKE adr,&FF
30 NEXT adr
```

Place l'octet &FF (4 pixels rouges) à &C000, puis &C001,… jusqu'à &FFFF. On fait RUN et que voit-on ? le remplissage démarre sur la 1ère ligne, puis sur la 9ème, puis sur la 17ème… Arrivé en bas, le remplissage reprend à la 2ème ligne, puis la 10ème, puis la 18ème,… étrange ! Pourtant on a bien remplit la mémoire de manière tout à fait linéaire, la mémoire vidéo ne le serait-elle pas, linéaire ? Et bien non, pour notre plus grand malheur. L'organisation des octets de mémoire en fonction de l'écran serait plutôt de ce genre (quelque soit le mode):

| | Octet 1 | Octet 2 | … | Octet 80 |
|---|---|---|---|---|
| Ligne 1 | &C000 | &C001 | … | &C04F |
| Ligne 2 | &C800 | &C801 | … | &C84F |
| … | … | … | … | … |
| Ligne 9 | &C050 | &C051 | … | &C09F |
| Ligne 10 | &C850 | &C851 | … | &C89F |
| … | … | … | … | … |
| Ligne 17 | &C0A0 | &C0A1 | ... | &C0EF |
| … | … | … | … | … |
| Ligne 200 | &FF80 | &FF81 | ... | &FFCF |

D'abord on voit bien qu'il y a 80 octets sur la largeur d'écran. On retrouve donc bien les résolutions en fonction du nombre de bits par couleur (mode 0 : 80x1/2 octets = 160 pixels, mode 1 : 80x1/4 octets = 320 pixels, mode 2 : 80x1/8 = 640 pixels). Ensuite, on voit bien la continuité entre l'octet 80 de la ligne 1 (&C04F) et l'octet 1 de la ligne 9 (&C050). Il en va de même pour les lignes 2 à 10, 9 à 17,… Ainsi, ligne 9 = ligne 1 + &50 et ligne 2 = ligne 1 + &800 et ainsi de suite.

On voit enfin que l'écran ne termine pas en &FFFF, mais en &FFCF. Il y a donc bien des octets non visibles, qu'on retrouve également entre chaque retour en haut de l'écran. Au total, il y a 384 octets non visibles et la boucle est bouclée. Bon heureusement, le BASIC et les vecteurs permettent de s'affranchir de cette organisation. Mais dans un prochain chapitre, nous verrons que pour afficher des sprites de manière rapide, il vaut mieux en tenir compte.

### 9-5 Codage des pixels dans les octets de mémoire vidéo

Bon ça va ? vous n'avez pas décroché et vous en voulez encore ? alors attachez bien vos ceintures, on passe au codage des pixels dans les octets de mémoire vidéo. Nous avons vu que selon le mode 0,1 ou 2, un octet permet l'affichage de 2,4 ou 8 pixels (ouf ! jusqu'ici tout va bien) :

| | OCTET DE MEMOIRE VIDEO | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Mode 0 | C0 | | | | C1 | | | |
| Mode 1 | C0 | | C1 | | C2 | | C3 | |
| Mode 2 | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |

Vous avez sûrement remarqué que je n'ai pas représenté les bits de l'octet de mémoire, ce qui aurait permis d'y voir plus clair par rapport aux numéros de couleurs. Seulement voilà, l'organisation des octets est un vrai bordel, qui plus est différente selon le mode.

<u>Mode 0</u>

Soient les 2 couleurs suivantes : C0 = C1 = 4 = %0100. Au final, la logique voudrait que l'octet donne C0C1 = %01000100 = &44, et bien non ! ce serait plutôt:

=> Octet = %00110000 = &30

C'est quoi le rapport me direz-vous ? Et bien le codage est en fait le suivant :

| | B3 | B2 | B1 | B0 |
|---|---|---|---|---|
| C0 | 0 | 1 | 0 | 0 |
| C1 | 0 | 1 | 0 | 0 |

| | OCTET EN MODE 0 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Codage | C0-B0 | C1-B0 | C0-B2 | C1-B2 | C0-B1 | C1-B1 | C0-B3 | C1-B3 |
| Résultat | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

Cherchez pas, il n'y a rien à comprendre, juste à appliquer bêtement. Vous ne me croyez pas ? alors on dit 2 pixels blancs (couleur 4 par défaut) en haut à gauche de l'écran, et voilà :

```
10 MODE 0
20 POKE &C000,&30
30 PRINT
```

Mode 1

Bon vous avez compris le principe du mode 0, alors pour le mode 1 on va changer un peu (si si, toujours la même chose c'est pas drôle). Prenons les couleurs C0 = C1 = C2 = C3 = 2 = %10. Le résultat ne donnera bien sure pas %10101010 = &AA comme vous commencez à vous en doutez, mais plutôt :

=> Octet = %00001111 = &0F

En effet :

| | B1 | B0 |
|---|---|---|
| C0 | 1 | 0 |
| C1 | 1 | 0 |
| C2 | 1 | 0 |
| C3 | 1 | 0 |

| | OCTET EN MODE 1 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Codage | C0-B0 | C1-B0 | C2-B0 | C3-B0 | C0-B1 | C1-B1 | C2-B1 | C3-B1 |
| Résultat | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Les poids forts sont à droite et les poids faibles à gauche. Et on dit 4 pixels bleus ciel en haut à gauche de l'écran, 4 !

```
10 MODE 1
20 POKE &C000,&0F
30 PRINT
```

Mode 2

Allez repos ! on passe maintenant au mode 2 qui est le plus simple de tous :

| | OCTET EN MODE 2 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Codage | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |

Vous voulez faire des pointillés, entrez donc %10101010.

Remarques

En BASIC ou avec les vecteurs textes classiques, le CPC s'arrange toujours pour afficher les caractères tels qu'ils doivent normalement apparaître. Mais dans ce cas, leur codage réel diffère selon le mode. Soit le programme BASIC suivant :

```
10 FOR m = 0 to 2
20 MODE m : PRINT "A"
30 CALL &BB18 : 'Attente de touche
40 NEXT m
```

Un 'A' s'affiche bien quelque soit le mode. Mais si on utilise le vecteur &BD1C qui change de mode sans réorganiser le codage du texte, le résultat est bien différent. Soit le programme assembleur suivant :

```
        ORG &4000

        XOR A
        CALL &BC0E      ; Mode 0 normal
        LD A,'A'
        CALL &BB5A      ; Affiche un 'A' à l'écran
        CALL &BB18      ; Attente de touche
        LD A,1
        CALL &BD1C      ; Mode 1 sans CLS ni réorganisation
        CALL &BB18      ; Attente de touche
        LD A,2
        CALL &BD1C      ; Mode 2 sans CLS ni réorganisation
        CALL &BB18      ; Attente de touche

        CALL 0          ; Reset complet
```

On compile le programme et on le lance sous BASIC avec un CALL &4000. Le vecteur &BB5A affiche alors un 'A' en fonction du mode 0 normal. Mais le changement de mode sans réorganisation fait que le A devient vite illisible.

### 9-6 Rafraîchissement de l'écran

50 fois par secondes, la mémoire vidéo est lue pour être affichée à l'écran (on parle alors de balayage 50 Hz). Ainsi quand on remplie un octet de mémoire vidéo, il sera lu et affiché au maximum 1/50$^{ème}$ de seconde après, ce qui peut paraître instantané. Si on pouvait ralentir très fortement le rafraîchissement de l'écran, on verrait un petit spot lumineux parcourir l'écran de gauche à droite et de haut en bas. A chaque fois qu'une ligne est terminée à droite, le spot revient à gauche sur la ligne juste en dessous et un signal HSYNCH (Horizontal Synchro.) est généré. Quand le spot arrive en bas à droite de l'écran, il remonte en haut à gauche et un signal VSYNCH (Vertical Synchro.) est généré. Les signaux HSYNCH et VSYNCH sont aussi transformés en signaux HBL (Horizontal BLanking) et VBL (Vertical BLanking).

Comme toujours, il est possible de modifier les divers temps et tailles d'écran pour obtenir des effets intéressants (voir plus bas). Mais sans ça, il peut être intéressant de s'aligner sur le balayage de l'écran, comme dans le cas d'affichage de sprites. Prenons le petit programme suivant (attention c'est une boucle sans fin qu'il faut terminer par un reset):

```
        ORG &4000

        LD A,1
        CALL &BC0E      ; Mode 1 normal
        LD DE,320
        LD HL,200       ; Position de départ = 320x200
AFFICHE PUSH DE
        CALL &BBC0      ; Positionne le curseur graphique en DE,HL
        LD A,'A'
        CALL &BBFC      ; Affiche A
        POP DE
        DEC DE          ; X <- X - 1
        LD A,D
        CP &FF          ; X < 0?
        JR NZ,AFFICHE   ; Non => AFFICHE
        LD DE,0
        CALL &BBC0      ; Curseur placé à gauche de l'écran
```

```
        LD A,' '
        CALL &BBFC       ; Efface le A avec un espace
        LD DE,632        ; On recharge DE à 632 (à droite de l'écran)
        JP AFFICHE
```

Affiche un 'A' en mode graphique qui se balade de droite à gauche de l'écran. Même si le caractère n'est pas bien gros, on voit bien qu'il subit une déformation au cours du déplacement. Et en général, cette déformation est d'autant plus visible que les sprites sont gros. Pourtant on affiche bien un caractère net et droit, alors d'où vient cette déformation? En fait, le programme ne reboucle pas tous les 1/50$^{ème}$ de seconde. Donc il arrive que le rafraîchissement de l'écran se fasse en plein affichage du sprite. Ce dernier sera alors coupé en 2, entre son ancienne position et sa nouvelle. Et comme le mouvement est constant, ce décalage se reproduira.

Si l'on veut que l'affichage paraisse beaucoup plus fluide, il faut pouvoir aligner le dessin des sprites sur le balayage de l'écran. Pour cela, on attend le top VBL, puis on affiche nos sprites. Le seul inconvénient, c'est que maintenant on n'a plus que 50 dessins par secondes, ce qui peut laisser penser que tout est ralenti. Il faut alors adapter les déplacements, mais au moins tout est net et fluide. Reprenons l'exemple ci-dessus avec l'attente VBL et un déplacement X – 2 (soit pixel – 1 en réalité) :

```
        ORG &4000

        LD A,1
        CALL &BC0E       ; Mode 1 normal
        LD DE,320
        LD HL,200        ; Position de départ = 320x200
AFFICHE PUSH DE
        CALL &BBC0       ; Positionne le curseur graphique en DE,HL
VBL     LD B,#F5         ; Attente de synchro verticale
        IN A,(C)
        RRA
        JR NC, VBL
        LD A,'A'
        CALL &BBFC       ; Affiche A
        POP DE
        DEC DE
        DEC DE           ; X <- X - 2 (mais pixel - 1)
        LD A,D
        CP &FF           ; X < 0?
        JR NZ,AFFICHE    ; Non => AFFICHE
        LD DE,0
        CALL &BBC0       ; Curseur placé à gauche de l'écran
        LD A,' '
        CALL &BBFC       ; Efface le A avec un espace
        LD DE,632        ; On recharge DE à 632 (à droite de l'écran)
        JP AFFICHE
```

Les instructions IN et OUT permettent l'accès aux ports, généralement réservés aux composants externes au Z80. Ici on interroge le port réservé à la VBL. La valeur est rapatriée dans A et si le bit 0 est à 1, alors il y a VBL et on peut afficher notre A. On voit bien que l'action est ralentie malgré le X-2, mais le A est parfaitement fluide.

La VBL sert aussi dans les scrollings d'écran (déplacement de l'écran tout entier). Par contre la HBL n'est pas accessible en temps que telle. Il faut alors créer des boucles de temporisation adéquates pour la simuler. Cependant, certaines fonctions se synchronisent dessus, comme le changement de mode graphique par exemple. Et c'est pour cela qu'on n'a jamais vu de changement de mode au milieu d'une ligne.

### 9-7 Les circuits vidéos annexes
### 9-7-1 Introduction

Le VGA et le CRTC sont des composants annexes aux Z80. Ils sont là pour gérer l'affichage des informations à l'écran. Comme tous les composants extérieurs au Z80, leur accès se fait sur les ports d'entrées/sorties de ce dernier, grâce aux instructions IN et OUT (INP et OUT en BASIC). Comme pour la mémoire vidéo, les adresses sont sur 16 bits et les données sur 8 bits :

```
        LD B,#F5         ; Port d'accès à la VBL = &F5xx
        IN A,(C)         ; Récupération de la VBL (bit 0)
        ...
        LD A,%11000100+1 ; Combinaison d'accès au 2ème bloc de RAM supérieure
```

```
LD BC,&7F00      ; Port d'accès au GATE-ARRAY
OUT (C),A        ; Accès à la requête
```

### 9-7-2 Le VGA (Video Gate Array)

Le Gate-Array (circuit de portes logiques) a été spécialement conçu pour le CPC. Il est accessible par le port &7Fxx et les bits de données 7 et 6 servent à sélectionner l'une de ses 4 fonctions principales :

| b7 | b6 | Fonction |
|---|---|---|
| 0 | 0 | Sélection d'une encre |
| 0 | 1 | Sélection d'une couleur |
| 1 | 0 | Réinitialisation du compteur d'interruption<br>Ou sélection des ROMs<br>Ou sélection du mode vidéo |
| 1 | 1 | Accès à la RAM supérieure ou supplémentaire |

**Sélection des encres et couleurs**

C'est en gros la commande INK, sauf que les couleurs en BASIC ne sont numérotées comme dans le Gate-Array. Attention toutefois de bien dévalider les interruptions avant de faire appel à cette fonction, car sinon le CPC réinitialise les couleurs de lui-même. La sélection d'une encre se fait de manière suivante :

Bits 7 et 6 à 0 : sélection d'une encre
Bit 5 : réservé au CPC+
Bit 4 : 1 = encre de bordure. 0 = couleur écran
Bits 3 à 0 : n° d'encre sélectionnée

La sélection de la couleur se fait ensuite de la manière suivante :

Bits 7 à 0 et bit 6 à 1 : sélection de couleur
Bit 5 : réservé au CPC+
Bits 4 à 0 : couleur de modification

Mais attention, le codage des couleurs n'est pas le même qu'en BASIC :

| Coul. | BAS | GA | Coul. | BAS | GA | Coul. | BAS | GA |
|---|---|---|---|---|---|---|---|---|
| | 00 | 20 | | 09 | 22 | | 18 | 18 |
| | 01 | 04 | | 10 | 06 | | 19 | 02 |
| | 02 | 21 | | 11 | 23 | | 20 | 19 |
| | 03 | 28 | | 12 | 30 | | 21 | 26 |
| | 04 | 24 | | 13 | 00 | | 22 | 25 |
| | 05 | 29 | | 14 | 31 | | 23 | 27 |
| | 06 | 12 | | 15 | 14 | | 24 | 10 |
| | 07 | 05 | | 16 | 07 | | 25 | 03 |
| | 08 | 13 | | 17 | 15 | | 26 | 11 |

Cela dit, pour ce qui est du changement de couleur, il est généralement préférable de passer par les vecteurs (Set ink = &BC32). Mais nous verrons dans un prochain chapitre comment obtenir des effets de démos intéressants avec le Gate-Array, comme ce qu'on appelle les rasters.

**Compteur d'interruption, ROMs et modes**

Bit 7 à 1 et bit 6 à 0
Bit 4 : à 1 réinitialise le compteur d'interruptions
Bit 3 : à 1 déconnecte la ROM supérieure (&C000-&FFFF)
Bit 2 : à 1 déconnecte la ROM inférieure (&0000-&3FFF)
Bits 1 et 0 : sélection du mode écran.

Le bit 4 remet à 0 le bit supérieur du diviseur qui génère les interruptions. Si les bits 3 ou 2 sont à 1, les ROMs correspondantes sont dévalidées. Mais regardons plutôt les bits 1 et 0 qui sélectionnent le mode

| b1 | b0 | Mode |
|---|---|---|
| 0 | 0 | 0 : 160x200 16 couleurs |
| 0 | 1 | 1 : 320x200 4 couleurs |

| | | |
|---|---|---|
| 1 | 0 | 2 : 640x200 2 couleurs |
| 1 | 1 | 3 : 160x200 4 couleurs |

Pardon ?! j'ai dit Mode 3 ? bah oui bien sur qu'il existe le mode 3 ! la preuve en image :

```
        ORG &A000

        DI              ; Interdire les interruptions
        LD BC,&7F8F     ; Désactiver les roms + mode 3 (#8C + 3)
        OUT (C),C       ; Accès au Gate-Array
        LD HL,0
        LD B,4
Wait    DEC HL          ; Boucle d'attente
        LD A,H
        OR L
        JR NZ,Wait
        DJNZ Wait
        EI              ; Réautoriser les interruptions -> le mode revient au
                        ; mode courrant automatiquement
        RET
```

**Accès à la RAM supérieure ou supplémentaire**

Le Z80 ne peut adresser que 64ko de mémoire, alors comment fait-on sur un 6128 avec 128ko de RAM ou un 464 avec une extension de mémoire DKTRONIC ? et bien entre &4000 et &7FFF, il est possible d'ouvrir une fenêtre sur un des 4 blocs de 16ko de la mémoire supérieure (4x16ko = 64ko). Le problème est que le bloc de mémoire basse &4000-&7FFF est alors momentanément inaccessible (mais pas écrasé par le bloc supérieur).

Bits 7 et 6 à 1 : demande d'accès à la RAM
Bit 4 et 3 : réservés aux extensions de 256ko
Bit 2 : 0 = accès à la mémoire basse normale. 1 = accès à la mémoire supérieure
Bits 1 et 0 : sélection du bloc de 16ko dont on veut accéder (%00 = 1$^{er}$ bloc, %01 = 2$^{ème}$,…)

Donc quand on met le bit 2 à 1, on accède à un bloc de 16ko de mémoire supérieure. Exemple, on veut accéder au 2$^{ème}$ bloc de mémoire supérieure :

```
        LD A,%11000100+1 ; 2ème bloc de RAM supérieure
        LD BC, &7F00     ; Demande d'accès au GATE-ARRAY
        OUT (C), A       ; Accès à la requète
```

**9-7-3 Le CRTC (Cathode Ray Tube Controller)**

Le CRTC 6845 est le circuit chargé de générer tous les signaux nécessaires à la production des images vidéos. Mais chose intéressante, il possède 19 registres permettant de paramétrer ces signaux. C'est grâce à lui qu'il est possible de faire des scrollings, d'utiliser toute la surface de l'écran (overscan), ou de faire des échanges d'écrans ultra-rapides. Malheureusement, car il faut bien un hic, Motorola le constructeur de ce composant a pris un malin plaisir à en sortir plusieurs versions, rendant certains CPC incompatibles entre eux à ce niveau. Les registres sont de 4 types :

1. **Registre d'adresse** : pour la sélection d'un des 18 autres registres.
2. **R0-R3** : programmation du format horizontal.
3. **R4-R9** : programmation du format vertical.
4. **R10-R17** : contrôle du curseur, de la RAM et du crayon optique.

Voici le tableau récapitulatif des registres. L/E = Lecture / Ecriture. Bits = nombre de bits significatifs, indiquant par la même occasion la valeur maximum entrée dans le registre.

| Reg. | L/E | Bits | Fonction |
|---|---|---|---|
| Adr. | E | 5 | Registre d'adresse |
| R0 | E | 8 | Temps total d'une ligne horizontale en nombre de caractères - 1 = temps d'affichage + temps de retour de spot – 1. Donc permet d'entrer une valeur comprise entre 1 et 256 caractères. |
| R1 | E | 8 | Nombre de caractères réellement affichables - 1. Mêmes règles que pour R0. |

| | | | |
|---|---|---|---|
| R2 | E | 8 | Temps, en nombre de caractères – 1, au bout duquel le signal HSYNCH est généré ( > R1). Mêmes règle que pour R0-R1. |
| R3 | E | 4 | Durée du signal HSYNCH en nombre de caractères – 1. |
| R4 | E | 7 | Temps total vertical en nombre de caractères – 1 (comprend le retour de spot en haut de l'écran). |
| R5 | E | 5 | Nombre de lignes d'écran à ajouter à R4 pour un réglage fin (0 à 31). En effet, les caractères peuvent faire plus de 8 lignes d'écran et dans ce cas il faut un ajustement. |
| R6 | E | 7 | Nombre de caractères réellement affichés verticalement – 1. |
| R7 | E | 7 | Temps, en nombre de caractères – 1, au bout duquel le signal VSYNCH est généré. Il dure ensuite 16 lignes d'écran. |
| R8 | E | 2 | Mode du CRT. 0 ou 2 = non entrelacé et 1 ou 3 = entrelacé. |
| R9 | E | 5 | Nombre de lignes d'écran par caractères – 1. |
| R10 | E | 2+5 | Bits 6 et 5 : aspect du curseur.<br><br>B6=0 : B5=0 : curseur fixe.<br>B6=0 : B5=1 : pas de curseur.<br>B6=1 : B5=0 : curseur clignotant rapidement.<br>B6=1 : B5=1 : curseur clignotant lentement.<br><br>Bits 4 à 0 : ligne écran de départ du curseur – 1 (1 à 32). |
| R11 | E | 5 | Ligne écran de fin du curseur – 1 (1 à 32). |
| R12 | E | 6 | Adresse haute de la mémoire vidéo (&C0 par défaut). |
| R13 | E | 8 | Adresse basse de la mémoire vidéo (&00 par défaut). |
| R14 | L/E | 6 | Adresse haute de la position du curseur. |
| R15 | L/E | 8 | Adresse basse de la position du curseur. |
| R16 | L | 6 | Adresse haute de la position du crayon optique. Cette option n'est pas standard sur le CPC. |
| R17 | L | 8 | Adresse basse de la position du crayon optique |

Attention, ces registres étant interconnectés, il faut les manier avec précaution sous peine d'obtenir un plantage de l'écran.

Le CRTC est accessible sur 4 ports :

1. **&BCxx** : registre d'adresse pour la sélection d'un des 18 autres registres. Exemple, sélection du registre 8 :

```
LD BC,&BC08   ; sélection du registre 8 du CRTC
OUT (C),C     ; envoi I/O
```

2. **&BDxx** : port d'écriture dans le registre sélectionné. Exemple, on veut écrire 6 dans le registre 9 :

```
LD BC,&BC09   ; sélection du registre 9 du CRTC
OUT (C),C     ; envoi I/O
LD BC,&BD06   ; accès au registre en écriture de la valeur &06
OUT (C),C     ; écriture dans le registre
```

3. **&BExx** : port d'interrogation du STATUT du CRTC. Seuls les bits 7, 6 et 5 sont actifs :

   B7 : à 1 si le CRTC est accessible.
   B6 : à 1 si le signal du crayon optique détecté. Mis à 0 si R16 et R17 ont été lus par le processeur
   B5 : BLANKING : 0 = le spot est en phase d'affichage. 1 = le spot est en phase de retour vertical

4. **&BFxx** : port de lecture des registres. Ne s'applique toutefois qu'aux registres 14 à 17.

```
LD BC,&BC0E   ; sélection du registre 14 du CRTC
OUT (C),C     ; envoi I/O
LD BC,&BF00   ; accès au registre en lecture
IN A,(C)      ; A = adresse haute du curseur
```

**Quelques remarques :**

En modifiant l'adresse de l'écran, par exemple de lignes en lignes, il est possible d'obtenir un scrolling vertical d'écran. En modifiant l'adresse de l'écran en plein balayage et en générant une fausse VBL à ce moment, il est

possible d’obtenir ce qu’on appelle une rupture d’écran. Mais on s’arrête là pour l’instant, car les démos seront développées lors d’un prochain chapitre.

### 9-8 Conclusion

La vidéo sur CPC n’est pas une chose simple. Aussi, si vous n’avez pas besoin de trop de rapidité, passez par les vecteurs qui en maîtrisent bien la structure. Sinon rien ne vaut l’expérimentation.