PROGRAMMATION ASSEMBLEUR SUR PC-ENGINE

PAR EDDY BRIERE

En cours d'écriture - Mise à jour : 09/08/2024 00:00

En cours d'écriture - Mise à jour : 09/08/2024 00:00

TABLE DES MATIÈRES

Avant-propos

Lorsque j'ai commencé à m'intéresser au développement sur la console PC-Engine, mes premières notes étaient simples et destinées à m'aider à comprendre les bases de cette technologie. Au fil du temps, ces notes se sont enrichies et transformées en un véritable guide de référence. J'ai ajouté des illustrations et des exemples de code machine pour faciliter la compréhension de concepts complexes, tels que la gestion de la mémoire.

Je me suis dit qu'il était important de peaufiner ce guide pour donner envie de développer sur cette machine et inspirer les générations futures. La PC-Engine, bien que considérée comme une vieille technologie, recèle des trésors d'ingéniosité et d'optimisation qui peuvent encore nous enseigner beaucoup aujourd'hui.

Ce livre est le fruit de nombreuses heures de travail et de passion. J'espère qu'il saura éveiller l'intérêt des développeurs, qu'ils soient débutants ou expérimentés, et qu'il contribuera à préserver et à transmettre un patrimoine technologique précieux.

Bonne lecture et bon développement !

Eddy B.

Introduction

La PC-Engine, avec sa technologie 8 bits, a su se démarquer par ses performances remarquables, même face aux machines 16 bits qui commençaient à inonder le marché. Ce guide a pour ambition de vous initier au développement sur cette console emblématique, en abordant plusieurs thèmes essentiels pour comprendre et maîtriser cette technologie.

Tout d'abord, nous commencerons par une présentation générale de la PC-Engine, en explorant son architecture et ses composants clés tels que le HUC6280, le HUC6260 et le HUC6270. Nous nous pencherons également sur la gestion de la mémoire, avec un chapitre dédié à la gestion de la mémoire au niveau du MMU (Memory Management Unit), notamment la gestion des banques de 8 Ko. Un autre chapitre sera consacré aux échanges de données entre la RAM/ROM et la VRAM.

Nous approfondirons ensuite les concepts de VDC (Video Display Controller) et de VCE (Video Color Encoder), qui sont au cœur du rendu graphique de la console. Nous verrons comment gérer les images et les sprites, et comment exploiter le DMA (Direct Memory Access) pour optimiser les performances.

Après avoir abordé le VDC et le VCE, nous nous intéresserons à la gestion des interruptions, un aspect crucial pour le développement de jeux et d'applications performantes sur la PC-Engine.

La gestion de la musique et des capacités sonores de la PC-Engine sera également un sujet central. Nous explorerons les possibilités offertes par cette console en matière de création musicale.

Enfin, pour mettre en pratique l'ensemble des concepts abordés, nous terminerons par des exemples de codes sources en assembleur. Ces exemples vous permettront de tester et de comprendre concrètement les différentes techniques de développement sur la PC-Engine. Il est important de noter que les exemples de code sont écrits en assembleur et qu'il est nécessaire d'avoir déjà des notions dans ce langage. Ceux qui ont déjà utilisé l'assembleur, particulièrement ceux ayant manipulé des CPU tels que le 6502 ou le 6510, devraient s'en sortir sans difficulté.

Ce guide est conçu pour être à la fois une introduction et une référence complète pour ceux qui souhaitent se plonger dans le développement sur cette console légendaire. Bonne lecture et bon développement !

Le Huc6280

Le HuC6280 est un processeur 8Bits de la société japonaise Hudson Soft est compatible avec le jeu d'instruction du 6502 dont il est une amélioration : Jeu d'instruction enrichi, un contrôleur d'interruptions, un *timer*, un bus d'entrés/sorites 8bits, une unité de programmation de génération sonore et une unité de gestion mémoire (MMU) lui permettant de travailler sur une mémoire physique allant jusqu'à 2Mo.

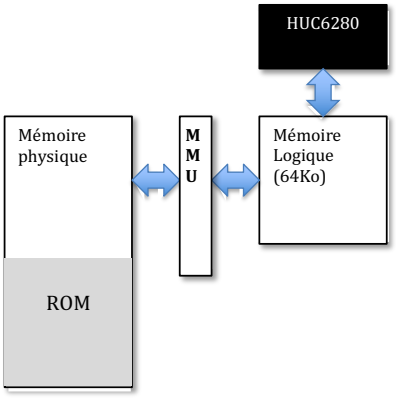
Le processeur est capable de fonctionner sur 2 fréquences d'horloge (1,79Mhz et 7,19MHZ) afin d'assurer la compatibilité avec d'autres composants (La cadences est ajustable via des instructions du processeur).

La gestion mémoire

Point intéressant du HuC6280 et qui risque d'en perturber plus d'un est sa gestion mémoire. En effet, le jeu d'instructions du 6502 intégré dans le HuC6280 ne permet pas d'adresse au-delà de 16bits de mémoire soit 64Ko. Pourtant nous venons de voir à l'instant que l'architecture pouvait bénéficier d'un espace mémoire de 2Mo. Pour pouvoir réaliser cet exploit, la PCE utilise un unité MMU (Unité de Gestion Mémoire).

L'unité MMU

L'unité MMU va créer une mémoire virtuelle (ou mémoire logique) de 64ko exploitable par le Huc6280. Le schéma ci-dessous est une représentation de ses deux « mémoires » intercalé entre l'unité MMU.



La stratégie retenue par les concepteurs de la PCE pour lier la mémoire logique à celle de découpé celle par block de 8ko.

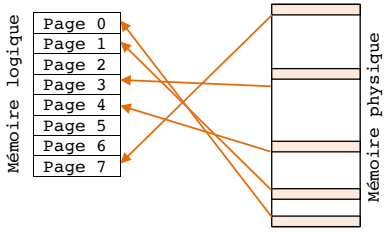
Ainsi la mémoire logique sera une succession de blocks de 8ko (8 au total pour bénéficier de 64ko accessible par le Huc6280). Chacun de ces blocks sera lié par le MMU à d'autres blocks de 8ko contenus dans la mémoire physique.

On nommera les blocks de 8ko de la mémoire logique Page et ceux de la mémoire physique Bank.

```
$0000 - $1FFF page 0
$2000 - $3FFF page 1
$4000 - $5FFF page 2
$6000 - $7FFF page 3
$8000 - $9FFF page 4
$A000 - $BFFF page 5
$C000 - $DFFF page 6
$E000 - $FFFF page 7
```

Le *mapping* mémoire n'est pas figé il peut-être changer à tout moment dans le programme exécuté et c'est grâce au MMU que l'on pourra accéder à plus de 64Ko de données.

Le schéma ci-dessous montre comment le MMU affecte des Banks de 8Ko aux Pages visibles par le Huc6280 :



***Note** : L'adressage de Banks de 8Ko sont forcément des multiples de 8192.*

L'instruction TAM du Huc6280

La configuration mémoire se réalise via l'instruction TAM.

Exemple :

```
lda #$FF ; numéro du Bank ($FF)
tam #$00 ; Page 0 associée à ce bank
lda #$F8 ; numéro du Bank ($F8)
tam #$01 ; Page 1 associée à ce bank
```

Pour associé il suffit de charger dans l'accumulateur la valeur du Bank désiré et de l'associer à la page n via l'instruction TAM n.

Dans notre exemple le Bank $FF (soit 255 en décimale) sera associé à la page 0 :

```
255 * 8192 soit = 2 088 960 soit en hexadécimal $1FE000
```

Ci-dessous la configuration des pages après l'exécution de notre programme exemple :

```
$0000 - $1FFF page 0 -> $1FE000-$1FFFFF
$2000 - $3FFF page 1 -> $1F0000-$1F1FFF
$4000 - $5FFF page 2
$6000 - $7FFF page 3
$8000 - $9FFF page 4
$A000 - $BFFF page 5
$C000 - $DFFF page 6
$E000 - $FFFF page 7 -> $000000-$001FFF (voir note)
```

Ainsi l'écriture du code suivant aura pour conséquence d'écrire la valeur $12 à l'adresse $1FE000 de la mémoire physique :

```
lda #$12 ;
sta #$0000 ;
```

Tandis que le code suivant écrira la valeur $34 à l'adresse de la mémoire physique $1F0000

```
lda #$34 ;
sta #$2000 ;
```

***Note** : Au démarrage de la PCE, la page 7 est automatiquement associée au Bank 0 ($000000-$001FFF).*

Autre particularité de l'instruction TAM est que lorsque que vous écrivez TAM #n dans votre assembleur et qu'ensuite vous vérifier via un éditeur hexadécimal votre fichier binaire vous observerez ceci :

```
Assembleur :
lda #$ff;
tam #$01;

Desassembleur :
$1000 A9 FF 53 02
```

Afin de simplifier l'écriture du code, les assembleurs Huc6280 nous laisse inscrire la valeur de Page dans l'instruction TAM. En réalité le numéro est traduit en puissance de 2 en mémoire.

L'instruction TAM a aussi sont inverse TMA qui permet de changer dans l'accumulateur la valeur du Bank d'une page.

La ROM & la RAM de la PCE

Quand nous regardons les caractéristiques de la PCE nous observons que la console, e possède 8ko RAM. Au premier abord, celle-ci peut sembler bien faible par rapport aux capacités de la console de jeu.

Ceux qui ont déjà développé sur des micros ordinateurs 8bits de l'époque de la PCE peuvent se demander mais comment va-t-il être possible de réaliser quoique ce soit avec si peu de mémoire RAM ? Sous PCE nous ne développons en RAM mais en ROM !

N'oubliez pas que la PCE est une console de jeu et que l'essentiel de ses programmes sont livrés sur des cartouches ROM (HuCard) ou CD-ROM. La RAM ne sert qu'à placer des données susceptibles d'être modifié dans le temps : Le Score du jeu, mémoriser les objets collectés dans un jeu, le résultat de calcul, etc.

Organisation de la mémoire physique de la PCE

Addresses physiques	Bank #	Description	Chip Enable Signal
1FFC00 - 1FFFFF	FF	Reserved for Expansion	
1FF800 - 1FFBFF		Reserved for Expansion	
1FF400 - 1FF7FF		Interrupt Req./Disable Registers	(/CECG)
1FF000 - 1FF3FF		I/O Ports	(/CEIO)
1FEC00 - 1FEFFF		TIMER Ports	(/CET)
1FE800 - 1FEBFF		PSG Ports	(/CEP)
1FE400 - 1FE7FF		HuC6260 Ports	/CEK
1FE000 - 1FE3FF		HuC6270 Ports	/CE7
1FC000 - 1FDFFF	FE		
1FA000 - 1FBFFF	FD		
1F8000 - 1F9FFF	FC		
1F2000 - 1F7FFF	F9 - FB		/CER
1F0200 - 1F1FFF	F8	Base "scratchpad" RAM	
1F0100 - 1F01FF		Stack Page	
1F0000 - 1F00FF		Zero Page	
1EE000 - 1EFFFF	F7	Last page of HuCard memory (ROM)	
004000 - 1EDFFF	02 - F6	HuCard storage (ROM)	
002000 - 003FFF	01		
001FFE - 001FFF	00	Reset Vector	
001FFC - 001FFD		NMI Vector	
001FFA - 001FFB		TIMER Vector	
001FF8 - 001FF9		IRQ1 Vector	
001FF6 - 001FF7		IRQ2 Vector (for BRK)	
000000 - 001FF5		First page of HuCard memory	

Le BOOT

Maintenant que nous avons vu comment s'organise et se gère la mémoire de la PCE nous allons voir de manière simplifiée comment un programme s'exécute.

Après avoir enfilé une cartouche et démarré la console le HuC6280 va consulter l'adresse $**FFFE** (vecteur reset du Huc6280) de la mémoire logique afin de connaître l'adresse de code à exécuter. Seulement voilà à quoi correspond l'adresse $**FFFE** de la mémoire logique ?

Souvenez-vous de ce qui a été dit plus haut. Au démarrage le *bank* 0 est automatiquement associé à la page 7 de la mémoire logique. Par conséquent, l'adresse se trouvant à l'adresse $**FFFE** de la mémoire logique correspond à l'adresse $**001FFE** de la mémoire physique.

Exemple :

Si le contenu de l'adresse $**001FFE** et $**001FFF** de la mémoire physique était respectivement $**00** et $**D0** alors cela signifierait que le code se trouvant à l'adresse $**D000** de la mémoire logique serait exécuté au démarrage de la console.

L'adresse $d000 étant inscrit dans la page 7 de la mémoire logique, le code serait alors inscrit à l'adresse physique $001000 (n'oubliez pas que vous travaillez toujours sur des blacks de 8ko).

Attention : *Ici $D000 est compris entre $E000 et $FFFF et donc bien contenu dans la page 7 visible par le Huc6280. Si vous aviez un pointeur en dessous de l'adresse $E000, il faudrait que les pages en dessous de 7 soient configurées or c'est impossible : nous venons de démarrer la console et seul le Bank 0 est visible par le Huc6280. C'est ensuite que les instruction TAM configurerons la mémoire de la PCE.*

Convertir les adresses mémoires

Si vous avez du mal à réaliser les conversions entre mémoire logique est physique voici une technique pour vous en sortir :

Ce qui est important de toujours savoir quel est le Bank est associé à un page. Imaginons que nous travaillons sur la page 7 soit entre l'adresse $**D000**-$**FFFF** de la mémoire logique et qu'à cette page est associé le Bank 0 ($**000000**-**001FFF**).

Un code se trouvant à l'adresse $d100 en mémoire logique se trouva alors en mémoire physique à l'adresse :

```
Page 7 -> Bank 0
Bank * 8192 + ($d100 AND $1fff) soit $001100
```

Inversement si nous avons un code inscrit dans le Bank 1 à l'adresse physique $002400 est que ce Bank est associé à la page 6 alors son adresse logique sera :

```
($002400 AND $1FFF) + Page * 8192 soit = $c400
```

Le point difficile à assimiler lorsqu'on commence le développement sur PCE est cette distinction entre mémoire physique et mémoire logique. N'oubliez pas que le HuC6280 ne peut travailler que sur la mémoire logique et tout moment vous pouvez réorganiser l'association entre les Banks et les pages.

Initialisation d'un programme PCE

Nous venons de voir comment définir l'adresse de démarrage d'un programme. Mais la question que l'on peut se poser est : Que devons-nous écrire avant de commencer ?

Il n'y pas d'obligation et le développeur à la liberté de choisir comment il désire commencer son programme. Cependant, la majorité des développeurs PCE ont l'habitude de démarrer leur code ainsi :

```
startup:
      sei;        // disable interrupts
      csh;        // select the 7.16 MHz clock
      cld;        // clear the decimal flag
      ldx #$FF;   // initialize the stack pointer
      txs;

      lda #$FF;   // map the I/O bank in the first page
      tam #0;
      lda #$F8;   // and the RAM bank in the second page
      tam #1;

      lda #$00;   // affecte le bank 0 à la page 7
      tam #7;     // au démarrage c'est la valeur par défaut
                  // donc pas nécessaire à écrire

      stz $2000;  // clear all the RAM
      tii $2000,$2001,$1FFF;
      …
```

Les instructions STZ et TTI permettent ici de remplir la RAM de zéro. Honnêtement je vous avouerai ne pas vraiment comprendre le but de cette manœuvre à part révéler le coté maniaque de certains développeurs.

La pile et la page zéro du HuC6280

Deux points importants sur la programmation du HuC6280 est de connaître l'emplacement de la pile et de la page zéro du processeur.

La page zéro et se trouve de l'adresse physique $**1F0000** à $**1F00FF** pour le HuC6280. La page zéro est un héritage du 6502 qui se trouvait de l'adresse mémoire $**0000** à $**00FF** d'où son nom.

La pile du HuC6280 (comme pour le 6502) se trouve après la page zéro soit de l'adresse mémoire physique $**1F0100** à $**1F01FF** est à une taille de 256 octets.

Cependant, ce qui se trouve d'après le code ci-dessus la RAM se trouve en page 1. Par conséquence la page zéro commence en $2000 et la pile en $2100.

Conclusion.

La gestion mémoire du HUC6280 bénéficie d'une grande souplesse de part à son unité MMU. La gestion des blocs mémoire permet de ne pas garder inutilement des données en mémoire logique : une fois les données d'un bloc exploiter, nous pouvons le remplacer par un nouveau bloc.

Les 8Ko RAM qui pouvait en effrayer certain comprendrons que l'on peut facilement développer sur PCE du fait de la possibilité de pouvoir inscrire énormément de données en ROM.

Néanmoins la programmation sur PCE demande une grande rigueur dans la gestion mémoire. Vous pouvez vite vous perdre si vous n'adoptez pas une stratégie de gestion de Page/Bank

Nous verrons plus loin dans ce guide une idée pour gérer la mémoire de notre PCE.

Le HuC6270 (VDC)

La puce graphique HuC6270 de la PCE est une puce 16bits possédant 64Ko de RAM que nous appellerons VRAM afin de la distinguer des 8Ko RAM du système.

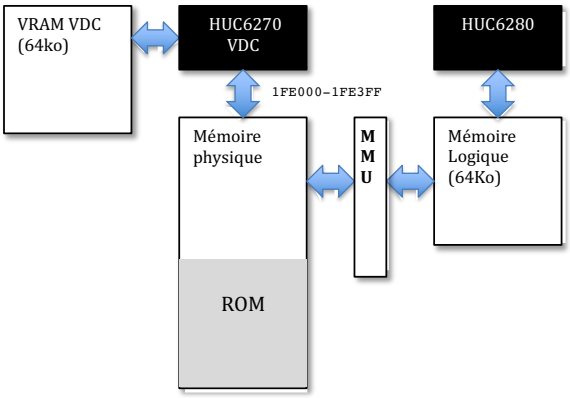
La HuC6270 est un processeur indépendant mais ce dernier sera esclave du HuC6280 qu'il le contrôlera via 3 registres accessibles aux adressages suivants : $1FE000, $1FE002 et $1FE003.

Le HuC6280 dispose de 3 instructions (ST0, ST1 et ST2) qui simplifieront l'écriture dans les 3 registres ci-dessus du HuC6270.

Note : Le HuC6270 est souvent appelé VDC (Video Display Controler) et c'est ainsi que nous allons continuer à l'appeler dans le document.

Les registres du VDC

Afin de mieux comprendre le fonctionnement du VDC, il faut avoir conscience que nous avons ici 2 unités indépendantes : le HuC6280 et le HuC6270 (VDC).



Une partie de ce schéma nous est maintenant familière. La nouveauté est la présence du VDC et de sa VRAM.

Pour dialoguer avec le VDC le Huc6280 pourra utiliser les instructions ST0, ST1 et ST2 en écriture. Dans le cas de la lecture des registres ne pourront être accessible que par adressage.

Les adressages $1FE000, $1FE002 et $1FE003 se trouve être localisé dans le Bank $**FF** (255) de la mémoire physique. Par convention, ce Bank est souvent mappé dans la page 0 de la mémoire logique :

```
lda #$FF;   // map the I/O bank in the first page
tam #0;
```

Ainsi les 3 registres pourront être accessible en mémoire logique avec les adresses $0000, $0002 et $0003. Pour écrire dans les 3 registres nous aurons donc 2 façons de procéder :

Façon 1:

```
lda #$05;
sta $0000;
lda #$00;
sta $0002;
sta $0003;
```

Façon 2:

```
st0 #$05;
st1 #$00;
st2 #$00;
```

A vous de choisir celle qui vous plaira.

```
lda #$05;
sta $0000;
stz $0002; //stz equivalent à lda #$00;sta #$0002;
stz $0003;
```

Le registre de commande

Nous venons de vous vous présenter les 3 registres de VDC mais à quoi servent-il ?

Dans un premier temps nous allons nous intéresser au premier :

En lecture

Le premier permet en lecture de connaître l'état du VDC :

Bit	Description
7	Inutilisé
6	Permet de savoir si le VDC est occupé (Action DMA)
5	Égal 1 lorsque que la synchro vertical se réalise
4	DV : pas d'information
3	DS : pas d'information
2	RR passe à 1 lorsque qu'une interruption de type scanline se réalise
1	OR : pas d'information
0	CE : pas d'information

Pour lire le registre exécuté les instructions suivantes :

```
lda #$ff;   // pointons la page 0 sur le I/O bank
tam #$00;

lda $0000;  // charge l'état du VDC dans l'accumulateur
```

En écriture

En écriture il permet l'accès aux 18 commandes du VDC.

Commande	Description
0	Pointeur de mémoire pour l'écriture en VRAM ($0000-$7fff)
1	Pointeur de mémoire pour la lecture en VRAM ($0000-$7fff)
2	Accès à la VRAM lecture/écriture
3	Inutilisé
4	Inutilisé
5	Registre de contrôle
6	Fixe la scanline qui déclenchera une interruption du VDC
7	Scroll en X (décalage de l'affichage)
8	Scroll en Y (décalage de l'affichage)
9	Largeur de l'écran ou des Sprites
10	Registre de synchronisation horizontal
11	Registre d'affichage horizontale
12	Registre de synchronisation verticale
13	Registre d'affichage verticale
14	Registre d'affichage de fin de position
15	Registre de contrôle DMA
16	Pointeur d'adresse source en VRAM pour le DMA
17	Pointeur d'adresse destination en VRAM pour le DMA
18	Largeur du bloc mémoire pour le DMA
19	Table d'attribution des Sprites

Le tableau ci-dessus décrit les commandes disponibles du VDC. Certaines sont explicites, tandis que d'autres le sont beaucoup moins. Tout au long de ce chapitre, nous détaillerons chacune d'entre elles, accompagnées d'exemples concrets.

Les registres des données

Le VDC est une puce 16bits et rien d'étonnant que ce dernier possède deux registres de données 8bits.

Le premier registre de données ($1FE002) correspond au poids faible des données et le second ($1FE003) correspond au poids fort des données.

Exemple :

```
Si vous voulez inscrire la valeur $1234 dans le registres des donnes
du VDC vous vous y prendrez comme suit :

st1 #34;    poids failbe
st2 #12;    poids fort
```

C'est l'écriture dans le troisième registre (st2) du VDC enclenchera l'exécution de la commande de VDC.

Échange de données avec le VDC

Il n'est pas possible d'accéder directement à la mémoire VRAM. Nous devons passer par une passerelle pour échanger entre le RAM et la VRAM. Voyons ensemble comment réaliser ces échanges entre les deux espaces mémoire.

Copie de mémoire logique vers VRAM

Afin de réaliser des échanges de données entre la RAM/ROM et la VRAM nous allons utiliser 3 commandes du VDC et des instructions du Huc6280 dédié à cette tâche :

```
Commandes VDC : $00, $01 et $02
Instructions du Huc6280 : TIA & TAI
```

Les données en mémoire RAM/ROM devront être accessibles par le CPU (Huc 6280) car les instruction TIA et TAI ne peuvent lire que la mémoire logique vue par le Huc 6280.

Pour écrire des données en VRAM il nous faudra donc définir la source et la destination :

- Adresse mémoire en RAM
- Adresse mémoire en VRAM

Imaginons que nous voulions écrire 512 octets en mémoire VRAM à l'adresse $1000 des données se trouvant en mémoire ROM à l'adresse $4000 :

- Adresse source en ROM -> $4000
- Adresse cible en VRAM -> $1000
- Nombre d'ocatet à copier -> 512 (soit $200 en hexadécimale)

Via la commande $00 du VDC nous allons indiquer l'adresse mémoire VRAM cible comme ceci :

```
st0 #$00; // Commande 0 -> Définir le pointeur d'écriture en VRAM
st1 #$00; // poid faible $00 de l'adresse en VRAM
st2 #$10; // et son poid fort $10 ($1000)
```

Maintenant que l'adresse cible est définie, c'est au tour du Huc6280 d'activer la copie via l'instruction TIA. Avant, d'utiliser l'instruction TIA, vous devrez écrire l'instruction st0 #$02. Afin d'activer l'accès en VRAM.

```
st0 #$02;              // Commande 2 -> Activer l'accès à la VRAM
tia $4000,$0002,$0200: //  tia copié 512 octets de l'adresse $4000(
                           mémoire logique) vers l'adresse $4000 de la
                           VRAM
```

Ces dernières lignes de code assembleur permettront de transférer 512 octets ($200) de données commençant à l'adresse $4000 de la mémoire logique vers l'adresse $1000 de la VRAM.

Copie VRAM vers mémoire logique

De manière analogue, la copie de données de la VRAM vers la RAM s'effectue comme suit :

```
st0 #$01; // Commande 1 -> Définir le pointeur lecture en VRAM
st1 #$00; // poids faible $00
st2 #$20; // et poids fort $20 donc adresse $2000

st0 #$02;              // Commande 2 -> Activer l'accès à la VRAM
tai $0002,$4000,$0200;  // copie de 512 octets
```

Ici nous copions donc 512 octets de l'adresse $2000 de la mémoire VRAM vers l'adresse $4000 de la mémoire RAM.

Noté que pour la copie de la VRAM vers le RAM nous utilisons la commande $01 du VDC.

Plus loin dans les échanges mémoire

Maintenant que nous maitrisons parfaitement les échanges de données entre la RAM et la VRAM j'aimerai revenir sur notre code :

```
st0 #$00;   // Pointeur d'écriture en VRAM
st1 #$00;   // poids faible $00
st2 #$10;   // et poids fort $10 donc adresse $1000
st0 #$02;   // activé l'échange de données
…
```

Juste après la commande $02 nous avions utilisé l'instruction TIA. Mais ici nous allons ajouter les instructions ST1 et ST2 comme suit :

```
st1 #$34; // poid faible $34 et poid fort $12 donc adresse $1234
st2 #$12;
```

Contrairement à l'exemple précédant, ici nous allons écrire uniquement une donnée de 16bits à l'adresse $1234 de la VRAM. Continuons et ajoutons à notre code les instructions suivantes :

```
st0 #$02; // Ecrire dans la VRAM les données suivantes
st1 #$98; // poid faible $89 et poid fort $67 donc adresse $6789
st2 #$67;
```

À nouveau nous allons inscrite en VRAM la données 16bits $6798 en VRAM. LA question que l'on pourrait se poser et dans quelle adresse mémoire de la VRAM cette données a été inscrite ? Et bien tout simplement à l'adresse $1002 de la VRAM.

En effet, l'appel successif de la commande $02 du VDC va incrémenter dans ce cas le pointeur de mémoire VRAM décrit précèdent pas la commande $00 du VDC. Ceci est valable pour la commande $01 du VDC et c'est ce qu'il passe lorsque les instructions TIA et TAI sont utilisées.

Exécution de l'instruction TIA :

```
Mécanisme de copie de l'instruction: tia $1000,$0002,$0200;

$1000 -> $0002
$1001 -> $0003
$1002- > $0002
$1003 -> $0003
…
```

Note : les instructions ST1 et ST2 sont des écritures dans les registres $1FE002 et $1FE003 du VDC. Elles permettent de simplifier l'écriture du code.

L'adresse de destination subit une incrémentation de type :

```
dest+A (A=A XOR 1);
```

Détail sur les commande TIA and TAI

On pourrait à ce moment du guide se demander pourquoi en destination de l'instruction TIA nous plaçons l'adresse $0002 ?

La raison est simple, lorsque la Bank $FF est associée à la Page 0 nous avons vu qu'à partir de l'adresse $0000 nous pointions sur les registres du VDC. Or, l'adresse $0002 (ou $1FE002 en mémoire physique) corresponds au registre de lecture ou d'écriture de donnée vers la VRAM.

A chaque octets copiés l'instruction TIA va réaliser les cycles suivants n fois :

```
Data0 -> $0002
Data1 -> $0003
Data2- > $0002
Data3 -> $0003
…
```

Il est de même pour TAI en revanche l'adressage de la source qui réalisera ses cycles :

```
$0002 -> Data0
$0003 -> Data1
$0002 -> Data2
$0003 -> Data3
…
```

Idéal pour copier des données de la VRAM vers la RAM.

Registre d'incrémentation lecture/écriture

Il est toutefois possible de changer la valeur d'incrémentation automatique via la commande **$05** du VDC et de le fixer aux valeurs suivants (ce registre est décrit plus précisément plus loin dans ce document) :

- +1
- +32
- +64
- +128

Les autres registres du VDC

Le registre $05 du VDC

REGSITRE DE CONTROL $05		
Mode	Bits	Description
Écriture	0	Activation du drapeau de collision du Sprite (Entre le Sprite 0 et tous les autres Sprites)
	1	Activation de l'interruption de surcharge d'affichage de Sprite (lorsqu'il y a plus de 16 Sprites sur une ligne)
	2	Activation du drapeau d'interruption de la scanline (raster)
	3	Activation du drapeau d'interruption Vertical Blank
	4-5	*Réservés*
	6	Activation des Sprite
	7	Activation de l'arrière-plan (background)
	8-10	*Réservés et toujours à 0*
	11-12	Réglage de la valeur d'auto-incrémentation pour les registres $00 et $01 du VDC : 00 = +1 01 = +32 10 = +64 11 = +128

15	14	13	12	11	*10*	*9*	*8*	7	6	*5*	*4*	3	2	1	0

Le registre $06 du VDC

REGSITRE DE COMPTAGE DE LIGNE (RASTER) $06		
Mode	Bits	Description
Écriture	0-9	Ligne de balayage sur laquelle une interruption doit être déclenchée (la première ligne de balayage de l'écran est la ligne de balayage numéro 64).

15	*14*	*13*	*12*	*11*	*10*	9	8	7	6	5	4	3	2	1	0

Le registre $07 du VDC

REGSITRE DE SCROLLING EN X $07		
Mode	Bits	Description
Écriture	0-9	Position horizontale de l'écran dans BAM virtuelle. Chaque fois que le VDC restitue une ligne de balayage, il obtient la position X de départ de l'écran à partir de ce registre (des effets assez intéressants peuvent être obtenus en changeant simplement ce registre à chaque ligne de balayage : distorsion, défilement sinusoïdal, etc.).

15	*14*	*13*	*12*	*11*	*10*	9	8	7	6	5	4	3	2	1	0

Le registre $08 du VDC

REGSITRE DE SCROLLING EN Y (RASTER) $08		
Mode	Bits	Description
Écriture	0-8	Position verticale de l'écran dans la carte virtuelle BG. Même principe que la caisse $07 mais pour la position verticale.

15	*14*	*13*	*12*	*11*	*10*	*9*	8	7	6	5	4	3	2	1	0

Le registre $09 du VDC

REGSITRE DE LA TAILLE DE LA BAM (RASTER) $09		
Mode	Bits	Description
Écriture	*0-3*	*Réservés*
	4-6	Permet de fixer la taille de la BAM. Les valeurs ci-dessous correspondent au nombre de tuiles (8x8) : 000 - 32 x 32 001 - 64 x 32 010 - 128 x 32 011 - 128 x 32 100 - 32 x 64 101 - 64 x 64 110 - 128 x 64 111 - 128 x 64 *Note : On peut remarquer des doublons pour 128x32 et 128x64. Je n'ai pas d'explication à ce jour.*
	7	*Réservés*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Les registres $0A à $E0 du VDC

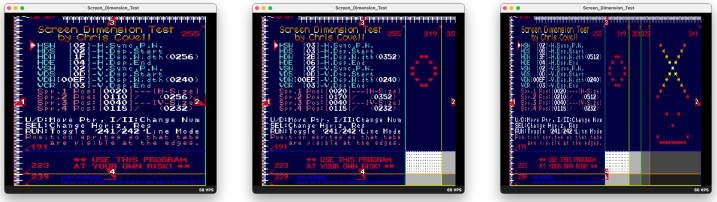
Les registres $0A à $0E contiennent les valeurs utilisées pour ajuster l'affichage. Ils contiennent des réglages pour les signaux de synchronisation verticale et horizontale, ainsi que pour la géométrie de l'écran.

Jouer avec ces registres pourrait endommager votre téléviseur ou votre moniteur si vous saisissez des valeurs incorrectes. Ils sont documentés ici par souci d'exhaustivité, mais attention si vous voulez jouer avec eux... Des valeurs de référence pour la mise en place d'un écran 256x240 sont données pour chaque registre.

Ci-dessous un exemple de code avec des valeurs suggérées pour un affichage de résolution 256x240 :

```
st0 #$0A ;
st1 #$02 ;
st2 #$02 ;

st0 #$0B ;
st1 #$1F ;
st2 #$03 ;

st0 #$0C ;
st1 #$02 ;
st2 #$0F ;

st0 #$0D ;
st1 #$EF ;
st2 #$00 ;

st0 #$0E ;
st1 #$03 ;
st2 #$00 ;
```

Sachez, qu'il existe un petit utilitaire homebrew *« Screen Dimension Test »* créé par Chris Covell permettant de jouer les réglages de ces registres :



Notez que la maniputlation de cet outil est à votre risque et péril. Perso rien à exploser sur mon pvm 😊

Le Registre $0A Synchro Horizontal - HSW et HDS

REGSITRE DE SCROLLING EN Y (RASTER) $0A HSW et HDS		
Mode	Bits	Description
Écriture	0-4	HSW : Largeur de la synchro horizontale
	8-14	HDS : Position de départ de l'affichage horizontal

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Le Registre $0B Affichage Horizontal - HDW et HDE

REGSITRE DE SCROLLING EN Y (RASTER) $0B HDW et HDE		
Mode	Bits	Description
Écriture	0-6	HDW : Largeur de l'affichage horizontal
	8-14	HDE : Position de fin de l'affichage horizontal

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Le Registre $0C Synchro Verticale - VSW et VDS

REGSITRE DE SCROLLING EN Y (RASTER) $0C VSW et VDS		
Mode	Bits	Description
Écriture	0-4	VSW : largeur de la synchro verticale
	8-15	VDS : Position de départ de l'affichage vertical

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Le Registre $0D Affichage Vertical - VDW

REGSITRE DE SCROLLING EN Y (RASTER) $0D VDW		
Mode	Bits	Description
Écriture	0-8	VDW : Largeur de l'affichage vertical

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Le Registre $0E Affichage Fin de Position - VDE (VCR)

REGSITRE DE SCROLLING EN Y (RASTER) $0E VDE (VCR)		
Mode	Bits	Description
Écriture	0-7	VDE : fin de l'affichage vertical

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Accompagnement au code VDC

Après lecture des chapitres précédents, l'initialisation du VDC peut vous paraître quelque peu complexe. C'est pourquoi dans cette partie nous allons rédiger un premier code d'initialisation de la PC-engine et du VDC :

Afin de vous accompagner dans l'élaboration de votre premier code en assembleur :

```
        .bank 0 ;
        .org $E000 ;

startup :

        sei ; innibition des interruptions
        csh ; cadence l'horloge du Huc6280 en mode hight (7Mhz)

        ldx #$FF ;
        txs ; initialise le pointeur de pile

        ; initilaisation des bank mémoire

        lda #$FF ;  page $1FFC00-$1FFFFF
        tam #$00 ;  associé à l'espace mémoire $0000-$1FFF du Huc6280

        lda #$F8 ;  page $1F0000-$1F1FFF (RAM de la PCE (8ko)
        tam #$01 ;  associé à l'espace mémoire $2000-$3FFF du Huc6280

        lda #$00 ;  page $000000-$001FFF (Page 0)
        tam #$07 ;  associé à l'espace mémoire $E000-$FFFF du Huc6280

        stz $2000 ;
        tti $0000,$0000,$2000 ; mettre à zero la mémoire RAM de $0000 à $1FFF

        ; interruptions
        lda #$FD ; desactive toutes les intterruptions sauf celle du VDC
        sta $1402 ;
        stz $1403 ; mise à zero du registre d'état des interuptions

        ; timer
        lda #$01 ;
        sta $0C01 ;démarrage du timer

        cli ; mise à zéro des masques d'interruption

        ; definition de la couleur de fond
        stz $0402 ; poitneur sur l'index zero de la palette de couleurs
        stz $0403 ;

        lda #%11000111 ; creation d'une couleur cyan pour le fond
        sta $0404 ;
        lda #%00000001 ;
        sta $0405

        ; initialisation du VDC
        st0 #$0A ; registre HSW & WDS
        st1 #$02 ;
```

```
      st2 #$02 ;

      st0 #$0B ; registre HDW & HDE
      st1 #$1F ;
      st2 #$03 ;

      st0 #$0C ; regsitre VSD & VDS
      st1 #$02 ;
      st2 #$0F ;

      st0 #$0D ; regsitre VDW
      st1 #$EF ;
      st2 #$00 ;

      st0 #$0E ; registre VDE (VCR)
      st1 #$03 ;
      st2 #$00 ;

loop :

      jmp loop ; boucle sans fin

      ; Vecteurs du Huc6280

      .org $FFF6
vectors :

      .dw _irq2  ; IRQ2 (BRK and external int)
      .dw _irq1  ; IRQ 1 (VDC)
      .dw _timer ; TIMER
      .dw _nmi   ; NMI
      .dw startup  ; RESET
```

Les pointeurs d'interruptions ne sont pas écrits dans ce code mais voici comment procéder. Ici un exemple pour l'interruption IQR2 :

```
_irq2:
       pha ; sauvegarde des registres du Huc6280 dans la pile
       txa ;
       pha ;
       tya ;
       pha ;

       ; ici le code de mon interruption

       pla ;
       tax ;
       pla ;
       tay ;
       pla ; restitution des registres du Huc6280

       rti ; retour d'interruption
```

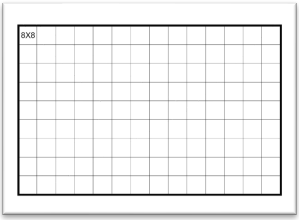
Ce programme doit afficher un écran cyan.

Réalisation graphique avec le VDC

Dans ce chapitre nous allons découvrir plus en profondeur les caractéristiques du VDC est voir en détail l'ensemble des fonctionnalités disponibles.

L'affichage du VDC (background)

Contrairement à ce qu'on pourrait imaginer la VRAM n'est pas un bitmap comme c'est le cas dans les cartes vidéo actuelles. Le VDC organise ses données graphiques sous forme de tuile d'une taille de 8x8 pixels pouvant afficher 16 couleurs (Tiles en anglais).

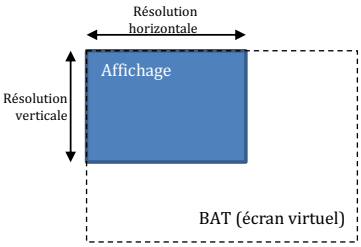


Pour connaître le nombre de tuiles affichage pour une résolution données il vous faudra simplement diviser celle-ci par 8.

```
256x200 pixels = 32x25 tuiles
```

Affichage et BAT

L'affiche est tout simplement votre écran. Ce dernier possède une résolution et vous permet de visualiser des données graphiques. La BAT est une matrice de tuiles (ou de caractère) de 8x8 pixels. Elle n'a pas totalement visible et peut être parcellement masquée. L'exemple ci-dessous nous montre un BAT plus grande que l'affichage :



La mémoire 16bits

La technologie de la PC-Engine jongle entre des adressages 8bits pour la partie CPU et 16bits comme par exemple avec le VDC.

Nous allons donc faire un rappel pour éviter les confusions dans les chapitres qui vont suivre.

Adresse	Data 8 bits
$2000	$10
$2001	$FF
$2003	$7F

Mémoire 8 bits

Adresse	Data 16 Bits
$2000	$10FF
$2001	$7FEE
$2002	$33AE

Mémoire 16 bits

Il faut bien comprendre qu'ici le nombre d'octets entre deux adresses mémoire qui se suivent diffère en fonction du type de la mémoire.

Ainsi entre l'adresse $2000 et $2001 sur une mémoire 8 bits nous avons 1 octet. Or pour une mémoire 16 bits nous auront 2 octets.

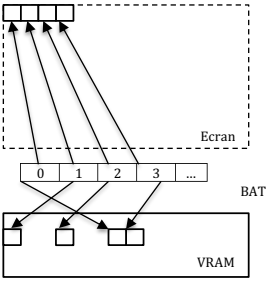
Si la VRAM est annoncée avec une taille de 64Ko elle est aussi d'une taille de 32Km (32768 mots).

La BAT ou comment organiser ses tuiles

A ce stade de la présentation de l'affichage du VDC, on pourrait s'imaginer qu'il suffit de fixer une résolution et inscrire directement nos données graphiques dans les tuiles pour voir jaillir de jolies images sur notre écran.

Oubliez cette idée, pour organiser nos tuiles à l'écran il faudra passer par la BAT (Block Attribute Table an anglais).

La BAT est une table organisée de gauche à droite et de haut en bas et dont chaque cellule va pointer sur une tuile stockée en VRAM.



En plus de pointer sur une tuile en VRAM, les cellules de la BAT permettent d'associer une palette de 16 couleurs pour chaque tuile parmi 16 palettes possibles.

La BAT et ses pointeurs

Nous venons de voir que la BAT était une table de pointeurs. Ces derniers ont une taille de 16bits et permettent de contenir l'adresse d'une tuile en VRAM et le numéro de palette utilisée par celle-ci.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P	P	P	P	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4

Les bits 15 à 12 permettent de définir le numéro de palette utilisée pour la tuile (0 à 15). Les 12 bits restants code l'adresse ou plus exactement l'index de la tuile en VRAM.

C'est 12bits vont nous permettre d'indexer 4096 tuiles en VRAM. Ceci est largement suffisant car une tuile occupe en mémoire 8x8x4 bits soit 32 octets (soit 16 mots).

Nous pouvons donc indexer 4096x32 soit 128Ko octets… on est très large finalement.

Cependant il ne sera pas possible d'index en deçà de l'adresse $1000 (on chevauchera l'adresse de la BAT) et au-delà de l'adresse $8000 (le bit 11 de la BAT ne nous est donc pas utile).

Exemple :

Ici nous allons déterminer la valeur à inscrire dans un pointeur de la BAT pour une tuile stockée à l'adresse $2000 en VRAM utilisant la palette de 16 couleurs numéro 3 :

```
$3 = 0011 en binaire
$2000 = 0010 0000 0000 0000 en binaire

$2000/10 = $200 soit 0010 0000 0000 en binaire
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0

```
En decoupant ce pointeur 16bits en poids faible/poids fort
on obtient :

$00 poids faible du pointeur de BAT
$32 poids fort du pointeur de BAT
```

Cette conversion vous sera utile pour utiliser les registres VDC concernés.

***Note** : Il ne sera pas possible d'avoir des tuiles stockées en dehors des adresses comprise entre $1000 et $8000. D'ailleurs c'est 12bits d'index vont au-delà de la capacité de la VRAM.*

Définir la taille de la BAT

Nous avons vu plus haut que la BAT pouvait être considérée comme un écran virtuel de taille variable. D'ailleurs le VDC offre une grande souplesse en ce qui concerne l'affichage et ceci est détaillée dans le chapitre Gestion de l'affichage.

La taille de la BAT est liée à la taille de notre écran virtuel. Bien qu'il soit possible de choisir le nombre de tuile que l'on veut afficher en largeur et hauteur sur l'écran, le VDC lui proposera un nombre déjà défini de tailles d'écrans virtuels (aucune autres valeurs possibles) :

La commande $09 du VDC permettra de définir la taille de l'écran virtuel via les bits de 4 à 6.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	H	W	W	x	x	x	x

WW (tuiles):

```
00 = 32 (256 pixels)
01 = 64 (512 pixels)
10 = 128 (1024 pixels)
11 = 128(1024 pixels) oui il y a une redondance 😊
```

H (tuiles):

```
0 = 32 (256 pixels)
1 = 64 (512 pixels)
```

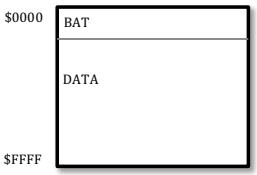
Le calcul de la taille de la BAT en mémoire se réalise de la manière suivante : Si nous avons un écran virtuel de 1024x512 pixels notre BAT occupera en mémoire :

```
(1024/8)*(512/8)*2 = 16Ko ( *2 car un pointeur de BAT est sur 16bits)
```

Écrire dans la BAT

A présent la BAT n'a plus de secret pour nous… enfin juste un dernier : Où se trouve-t-elle en mémoire ?

La BAT se trouve dans la VRAM à l'adresse $0000, du moins elle commence à cette adresse. En fonction de la taille choisie, celle-ci occupera plus ou moins d'espace dans la VRAM.



LA BAT DANS LA VRAM

Autre point important, plus votre écran virtuel sera grand moins il vous restera d'espace pour vos données graphiques. Pour un écran virtuel de 1024x512 pixels nous avons vu que celui-ci prenait 16Ko de la VRAM. Il nous restera donc 48 Ko pour nos données graphiques.

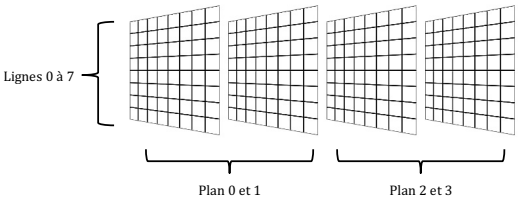
Les tuiles

Nous venons de voir que la BAT permettait d'inscrire l'index des tuiles en VRAM mais aussi le numéro de la palette de 16 couleurs de chacune d'entre elle.

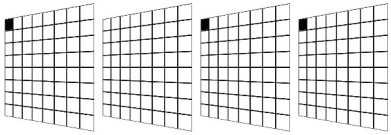
Bien que la notion de palette reste encore floue à ce stade du document, nous allons voir comment nous codons une couleur (ou plus exactement un numéro de couleur) dans les données d'une tuile.

Codage de la couleur d'un pixel (Planar mode)

Avant d'aller plus loin, il va falloir imaginer une tuile comme étant 4 matrices de 8x8 superposées :



Chaque matrice peut avoir un pixel activé (1) ou pas (0) :



Le codage utiliser pour affecter le numéro d'une couleur à un pixel de cette manière s'appelle le *mode planar*. En effet, c'est l'état du bit de chaque matrice pour un même point qui déterminera le numéro de la couleur.

Sur l'illustration on remarquera que le premier bit/pixel est allumé (1) sur les plans 1, 3, 4 et éteints (0) sur le plan 2 :

```
1 sur le plan 0
0 sur le plan 1
1 sur le plan 2
1 sur le plan 3

L'index de couleur du pixel dans la palette sera alors
1101 en binaire
```

On déterminera ainsi le numéro de la couleur de ce pixel : 1011 en binaire soit 13 en décimal.

Taille d'une tuile

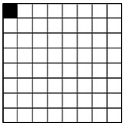
Dans le chapitre consacré à la BAT nous vous avions annoncé qu'une tuile occupée 32 octets en mémoire VRAM. Ce calcul est calculé comme suit :

```
8*8*4 bits = 256 bits soit 32 octets

8*8 taille de ma matrice
4 nombre de plan
```

Maintenant que nous connaissons l'espace occupé par une tuile en VRAM, intéressons-nous à son organisation en VRAM.

Organisation des données en VRAM

Revenons sur notre tuile mais vu de face cette fois :



Il vous est facile maintenant d'imaginer que cette matrice est constituée 4 groupes de 8 octets qui se suivent ?

Le premier octet codant les 8 premiers pixels de la première ligne du mon premier plan. Le second la deuxième ligne et ainsi de suite jusqu'à ma huitième ligne.

Mon huitième octet lui code pour la première ligne de mon deuxième plan et je vous laisse deviner la suite.

Il aurait simple que le VDC organise ses octets en VRAM ainsi mais celui-ci si prend comme suit :

0	Plan 1 & 2 de la ligne 0
2	Plan 1 & 2 de la ligne 1
4	Plan 1 & 2 de la ligne 2
6	Plan 1 & 2 de la ligne 3
8	Plan 1 & 2 de la ligne 4
10	Plan 1 & 2 de la ligne 5
12	Plan 1 & 2 de la ligne 6
14	Plan 1 & 2 de la ligne 7
16	Plan 3 & 4 de la ligne 0
18	Plan 3 & 4 de la ligne 1
20	Plan 3 & 4 de la ligne 2
22	Plan 3 & 4 de la ligne 3
24	Plan 3 & 4 de la ligne 4
26	Plan 3 & 4 de la ligne 5
28	Plan 3 & 4 de la ligne 6
30	Plan 3 & 4 de la ligne 7

Maintenant, si je voulais écrire en assembleur ma tuile avec mon premier pixel codant la couleur 1011 en binaire j'aurais les octets suivant pour la première ligne de mes plans comme suit :

```
Plan 1 - Octet 1 : 1000 000 en binaire soit $80 en hexadécimal
Plan 2 - Octet 1 : 0000 000 en binaire soit $00 en hexadécimal
Plan 3 - Octet 1 : 1000 000 en binaire soit $80 en hexadécimal
Plan 4 - Octet 1 : 1000 000 en binaire soit $80 en hexadécimal
```

Les autres octets n'ayant aucun point allumé seraient tous à nul

```
// donnée d'une tuile en assembleur
db $80,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
db $80,$80,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00
```

Exemple : Créer et copier une tuile en VRAM

Dans cet exemple, nous allons créer une tuile représentant un visage et fournir le code assembleur pour le copier dans la VRAM.

0000 Blanc 0001 **Noir** 0011 **Rose**
0100 **Bleu** 0101 **Rouge** ***(valeurs en binaire)***

Voyons comment réaliser à la main le codage de notre tuile afin de le reporté plus tard dans notre code:

Prenons la première ligne avec des pixels blanc et noir. Ensuite superposons, les 4 octets de nos plans pour cette première ligne :

Ligne 1, plan 1	0	0	**1**	**1**	**1**	**1**	0	0
Ligne 1, plan 2	0	0	0	0	0	0	0	0
Ligne 1, plan 3	0	0	0	0	0	0	0	0
Ligne 1, plan 4	0	0	0	0	0	0	0	0

Octets correspodants en héxadécimal : $7E,$00,$00,$00

Faisons de même pour la deuxième ligne (blanc, noir et rose):

Ligne 2, plan 1	0	**1**	**1**	**1**	**1**	**1**	**1**	0
Ligne 2, plan 2	0	0	**1**	**1**	**1**	**1**	0	0
Ligne 2, plan 3	0	0	0	0	0	0	0	0
Ligne 2, plan 4	0	0	0	0	0	0	0	0

Octets correspodants en héxadécimal : $7E,$3C,$00,$00

Puis la troisième (blanc, noir, bleu et rose):

Ligne 3, plan 1	0	**1**	**0**	**1**	**1**	**0**	**1**	0
Ligne 3, plan 2	0	0	**0**	**1**	**1**	**0**	0	0
Ligne 3, plan 3	0	0	**1**	0	0	**1**	0	0
Ligne 3, plan 4	0	0	0	0	0	0	0	0

Octets correspodants en héxadécimal : $7E,$14,$24,$00

Enfin la sixième (les autres étant identiques à la première ou deuxième ligne):

Ligne 6, plan 1	0	**1**	**1**	**1**	**1**	**1**	**1**	0
Ligne 6, plan 2	0	0	**1**	**0**	**0**	**1**	0	0
Ligne 6, plan 3	0	0	0	**1**	**1**	0	0	0
Ligne 6, plan 4	0	0	0	0	0	0	0	0

Octets correspodants en héxadécimal : $7E,$24,$18,$00

De manière analogue vous déterminerez les octets suivants de cette tuile. Coté assembleur voyons comment nous allons mettre ça en place :

```
face:
      //plan 1 et 2
      db $7E,$00,$7E,$3C,$7E,$14,$7E,$3C,$7E,$3C,$7E,$24,$7E,$3C,$7E,$00;
      //plan 3 et 4
      db $00,$00,$00,$00,$24,$00,$00,$00,$00,$00,$18,$00,$00,$00,$00,$00;

tovram:

      st0 #$00;   // Pointeur d'écriture en VRAM
      st1 #$00;   // poids faible $00
      st2 #$40;   // et poids fort $40 donc adresse $4000

      st0 #$02;   // activé l'échange de données
      tia face,$0002,$20;

      rts;
```

Définir l'adresse des données graphiques

Une fois que nous avons défini la taille de notre BAT où doit-on stocker nos données graphiques ?

Dans un souci d'optimisation mémoire, le plus judicieux serait de les inscrire juste après la BAT. Évidemment, à chaque projet ses contraintes mais regardons dans un cas d'optimisation.

Si nous partons sur un BAT de 32x25 pour travailler sur une résolution de 256x200 pixels.

Dans ce cas, notre BAT occupera 32x20x2 (x2 car nous avons à faire à des mots) soit 1600 octets (800 mots).

Écrire dans la BAT

Charger les données graphiques

La gestion des couleurs HuC6260 (VCE)

Le HuC6260 est appeler plus communément VCE (Video Color Encoder) est connecté directement au VDC et permet la gestion des palettes de couleurs ainsi que le prise en charge du signal vidéo.

Il possède 4 registres attachés aux adresses physiques $1FF400, $1FF402, $1FF404 et $1FF406. Lorsque la BANK $FF est associé à la page 0 les adresses sont respectivement les suivantes pour le HUC6280 : $0400, $0402 et $0404.

Palettes de couleurs

Le VCE possède une table de 1024 Ko permettant de définir 512 couleurs. Cette table est ensuite divisée en 32 palettes :

- Les 16 premières sont destinées aux couleurs des tuiles.
- Les 16 autres sont destinées aux couleurs des Sprites.

Chacune de ces 16 palettes contiennent 16 définitions de couleurs et sont codé sur 16 bits comme suit :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	G	G	G	R	R	R	B	B	B

```
Les bits de 0 à 2 codent les composantes bleues d'une couleur
Les bits de 3 à 5 codent les composantes rouges d'une couleur
Les bits de 6 à 8 codent les composantes vertes d'une couleur
```

Chaque composante couleur pouvant avoir une valeur de luminosité comprise entre 0 et 7 soit 512 nuances possibles (8x8x8).

Codage d'une couleur

Pour indication, la couleur ici sera codé ainsi dans la table d'une palette commet suit :

```
//couleur jaune
Rouge = 7 (111 en binaire)
Verte = 7 (111 en bianire)
Bleue = 0 (000 en bianire)

Soit 0000 0001 1111 1000 sur un mot de 16bits du VCE
Soit $01F8 en héxadécimal

Représentation poids faible/ poids fort :
Poids faible : $f8
Poids fort : $01
```

Note : *Le codage des couleurs de Sprite se réalisera de manière analogue.*

La couleur 0

Le VCE offre donc 16 palettes de 16 couleurs destinées aux couleurs des tuiles et idem pour les Sprites. Chaque palette est indexée de 0 à 15 avec une particularité pour concernant l'index 0.

En Effet, sur sa table de 512 couleurs, le VDC utilise la première valeur comme couleur de fond. En ce qui concerne les palettes de 16 couleurs du VDC, l'index 0 sera toujours une couleur de transparence. Ceci signifie que la première couleur inscrite dans la table de 512 du VDC déterminera la couleur 0 de toutes les palettes de 16 couleurs.

Dans l'exemple ci-dessus nous avons créé deux palettes de 16 couleurs :

- La première pour notre texte
- La seconde pour notre image

Sur la couleur 0 de la première palette nous avons créé un violet. En revanche sur la seconde la couleur 0 est du noir. Comme nous pouvons le constater, le VCE n'en tient pas compte.

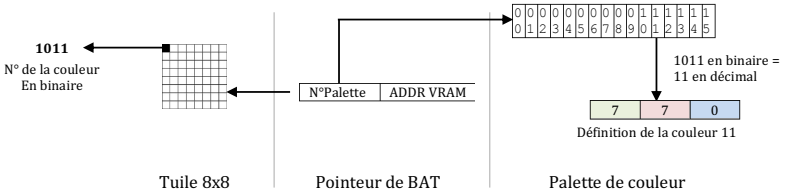
Conclusion, la première palette de 16 couleurs permet de définir réellement 16 couleurs, les autres auront leur couleur 0 fixées par la première. Par conséquence nous pouvons afficher que (256-15) soit 241 couleurs en simultané sur la BAT (sans bidouille).

Correspondante palettes avec Tuiles/Sprites

A ce stade du document nous savons associer un numéro de couleur à un pixel d'une tuile (voir chapitre Les tuiles).

Nous savons aussi comment affecter le numéro d'une palette de 16 couleurs à une tuile via la BAT (chapitre

Ici, Nous venons compléter la chaine complète de correspondance entre tuile, BAT est palette :



Les registres du VCE

Maintenant que nous avons vu en détail les palettes de couleurs nous allons pouvoir regarder de plus près les registres du VCE.

Nous allons nous intéresser à 3 de ses registres de 16bits :

```
$01FE400 -> $0400
$01FE400 -> $0402
$01FE404 -> $0404
```

Note : Toujours pour un mapping du Bank $FF en Page 0 nous travaillerons sur les adresses du VCE via l'adresse $040x.

Le registre de controle $0400

REGSITRE DE CONTROLE		
Mode	Bits	Description
Écriture Lecture	0-1	Horloge de fréquence pixel 00=5.3693175 Mhz 01=7.15909 Mhz 10=10.738635 Mhz 11=10.738635 Mhz
	2	Cadre/champ (Frame/field) 0 = 262 lignes 1 = 263 lignes
	3-6	Non renseignés
	7	Strip ColorBurst 0 = Colorburst intact 1= Strip colorburst
	8-15	Non utilisés

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Temps pour 1 pixel est de 200ns pour une horloge pour 5,3693175 Mhz
- Temps pour 1 pixel est de 139,68ns pour une horloge à 7,15909 Mhz
- Temps pour 1 pixel est de 93,12ns pour une horloge à 10,738635 Mhz

Le registre d'index de la table des couleurs $0402

REGSITRE D'INDEX DE LA TABLE DES COULEURS		
Mode	Bits	Description
Écriture Lecture	0-8	Index de palette de 512 entrées : • 0-255 pour les 16 palettes de la BAT • 256-511 pour les 16 palettes des Sprites
	9-15	Non utilisés

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Le registre de données de la table des couleurs $0404

REGSITRE DE DONNEES DE LA TABLE DES COULEURS		
Mode	Bits	Description
Écriture Lecture	0-2	Composante bleue
	3-5	Composante rouge
	6-8	Composante verte
	9-15	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Définir une palette de couleur

Comme nous l'avons vu pour l'écriture de données dans la VRAM, l'écriture dans la table des palettes va se réaliser de la même manière.

```
        stz $0402 ; fréquence d'horloge fixée à 5Mhz
        stz $0403 ; Index de palette à 0

        tia palette,$0404,40 ; copy les données pointé par _pal
                             ; dans la table de couleurs

        …

_pal:
        db $7E,$00,$7E,$3C,$7E,$14 …
```

Comme pour l'écriture en VRAM, l'index de palette s'auto incrémente à chaque écriture dans le registre $0404 du VDE. Ceci nous permet l'utilisation de l'instruction TIA.

Le registre de lecture de la table

De façons analogue mais en utilisant l'instruction TAI il est possibles de lire des données contenues dans la table des couleurs.

Quelques exemples de couleurs

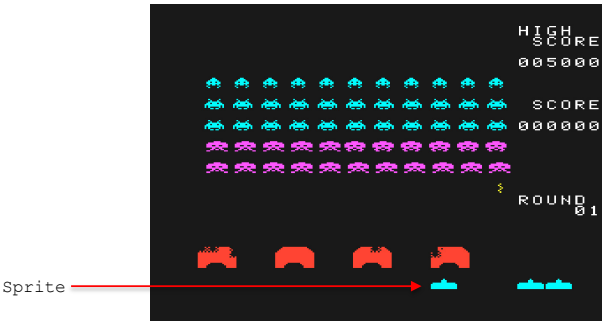
Couleur	Rouge	Vert	Bleu	Poids faible	Poids fort
Blanc	7	7	7	$ff	$01
Noir	0	0	0	$00	$00
Gris 50%	4	4	4	$24	$01
Jaune	7	7	0	$f8	$01
Rouge	7	0	0	$38	$00
Bleu	0	0	7	$07	$00
Vert	0	7	0	$c0	$01
Magenta	7	0	7	$c7	$01
Cyan	0	7	7	$3f	$00
Orange	7	4	0	$38	$01
Marron	4	2	0	$a0	$00
Bordeaux	4	0	4	$24	$00
Violet	4	0	7	$27	$00
Vert foncé	2	4	0	$10	$01

Les sprites

Les sprites sont des matrices des pixels indépendants du background. Ils peuvent se placer devant ou derrière le background (tuiles).

Leurs tailles est de 16x16 pixels et permettent d'exploiter, comme pour les tuiles, une palette de 16 couleurs. Comme pour les tuiles, la couleur 0 est transparente et donc ces derniers ne peuvent afficher réellement que 15 couleurs.

De manière générale, les sprites sont utilisés dans les jeux pour créer des entités qui viennent se mouvoir dans un décor (le background).



Le VDC permet l'affichage de 64 sprites simultanément.

La SATB

La SATB est ne table destinée à contenir l'ensembles des propriétés des sprites. Elle est ce qu'est la TAB pour les tuiles. On y renseignera la palette, les positions et autres caractéristiques du sprite.

Note : lorsque le canal VRAM-SATB est affecté le VDC attends la prochaine synchronisation vertical pour enclencher le transfert. Dans le cas où le bit 4 du DMA_CONTROL_REGISTER est activé, le transfert se réalise à chaque synchro verticale.

Cette synchronisation hardware ne permet pas le multiplexage des sprites. C'est à dire qu'il n'est pas possible de donner deux positions à un sprite sur un balayage. Cette technique est souvent utilisée pour afficher plus de sprite qu'il n'est prévu mais heureusement ce bridage ne frustra pas complétement le développeur car la pc-engine car le nombre de sprite d'affichage est honorable.

Structure d'un Sprite dans la VRAM

Comme pour les tuiles, les sprites doivent êtes décrit en mémoire VRAM du VDC. La structure des données graphique elle légèrement différente de celle d'une tuile (voir le chapitre « Les tuiles » page 37).

Gestion de l'affichage

Dans ce chapitre nous allons voir comment configurer son affichage et son écran. Pour rappel, ce que nous appelons l'affichage ici est produit à la sortie vidéo.

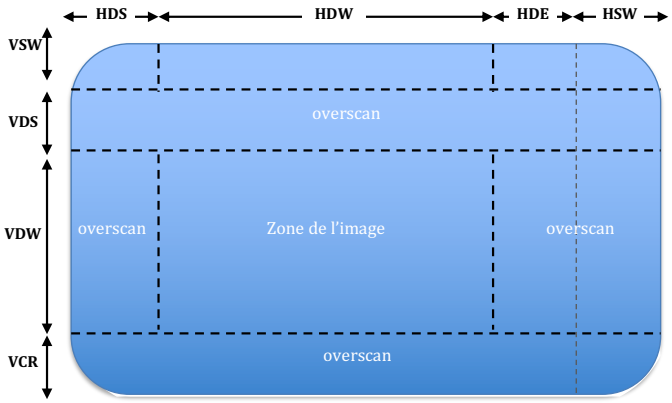
La configuration et réaliser par les registres de $0A à $0E du VDC décrits dans au chapitre : « Les registres $0A à $E0 du VDC ».

Voyons ensemble la configuration de ses derniers.

Affichage/Écran (rappel)

Taille de l'écran

Résolution de l'affichage



HDS : Position de début d'affichage horizontal -1
HDW : Largeur Horizontal d'affichage de tuiles -1
HDE : Fin de la période d'affichage horizontal -1
HSW : Largeur d'impulsion synchrone horizontale
VSW : Largeur d'impulsion synchrone verticale
VDS : Position de début d'affichage verticale -2
VDW : Hauteur de la zone de l'image en pixel
VCR : Fin de positon d'affichage verticale

```
33          ^end hsync
34          ^start vsync (30 cycles after hsync)
```

Overscan

L'overscan est la zone dans laquelle on ne peut afficher ni tuile ni sprite et dont la couleur est associé à la couleur 0 de la palette.

Les scrollings

Les scrollings est une technique qui permet de décaler horizontalement ou verticalement l'ensemble de l'affichage. Le VDC de la PC-Engine possède des registres permettant de décaler l'affichage de n pixels horizontalement et verticalement.

Taille des Sprites

Le DMA du VDC

Le DMA (Direct Access Memory) du VDC permet le Transfer de bloc mémoire sur deux canaux :

- Transfère de la VRAM vers VRAM
- Transfère de la VRAM vers la SATB

Il peut être utile de copier des données en VRAM à un autre endroit de la VRAM. Notamment cela peut-être très pratique pour des fonctions de scrolling d'écran,ou encore pour dupliquer des block mémoire.

Les registres DMA

Les registres du VDC qui vont nous être utile pour réaliser la copie via le DMA sont les suivants : $0F, $10, $11 et $12.

Nous utiliserons aussi le registre $00 pour contrôler l'activité du DMA. Mais voyons ça en détail :

Le registre $00

REGSITRE I/O DE D'INDEX ET D'ÉTAT $0000		
Mode	Bits	Description
Lecture	0	Drapeau de collision avec le Sprite #0 et les autres
	1	Drapeau de surcharge de Sprites sur de plus de 16 sur une ligne.
	2	Drapeau d'interruption de ligne (scanline)
	3	Drapeau de fin de transfert DMA de la VRAM vers la SATB
	4	Drapeau de fin de transfert DMA entre VRAM vers VRAM
	5	Activation des Sprite
Écriture	0-4	Registre d'index du VDC entre 0 et 19 qui sera accessible via le port I/O $0002 ou st1 et st2.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Ce registre du VDC est déjà connu, mais ici ce qui nous intéresse c'est sa lecture et notamment les bit 3 & 4.

Lorsqu'on réalise un copie DMA celle-ci prends un certain temps. Il est donc important de connaître la fin d'une copier afin de synchroniser avec d'autres tâches. Le regsitre $0000 en lecture nous permettra de surveiller l'activité du DMA.

Les modes de copie

Les concepteurs du VDC ont ajouté quelques fonctions sophistiquées au DMA comme par exemple définir le sens de lecture des données en mémoire. Avant de d'entrer dans le vif du sujet, regardons le registre $0F du VDC :

REGSITRE DE CONTROLE DMA $0F		
Mode	Bits	Description
Écriture	0	Activer l'interruption de fin de transfert VRAM vers s SATB (1=activé)
	1	Activer l'interruption de fin de transfert VRAM vers s VRAM (1=activé)
	2	Direction de copie de la source • 1=décrémenter • 0=incrémenter
	3	Direction de copie de la destination • 1=décrémenter • 0=incrémenter
	4	Drapeau de l'auto-transfert du DMA SATB (1 = activé)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

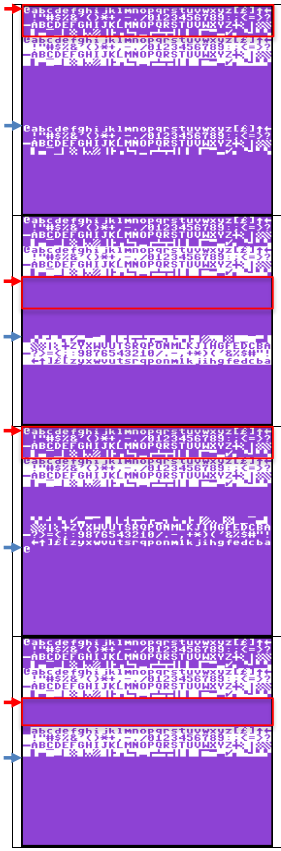
Si l'on porte attention au bit 2 & 4, nous voyons qu'il est possible au niveau de la source des données (ou celle de la destination) de choisir une direction de copie.

Pour simplifier l'explication je vous propose un tableau permettant d'illustrer chacun des cas possibles

Sur l'écran de la PC-Engine j'ai donc affiché 256 caractères. J'ai choisi d'utiliser le jeu de caractère du Commodore 64 car il possède par défaut 128 caractères inversés dans sa table ce qui va nous aider visuellement à comprendre.

Mon bloc mémoire de source contient toujours 128 caractères (ou tuile) à copier. En fonction du mode (incrémenté ou décrémenté) je changerai l'adresse de source afin que le DMA y trouve des données (sinon il irait prendre des données qui ne sont pas mes caractère dans la mémoire).

Si l'on porte attention au bit 2 & 4, nous voyons qu'il est possible au niveau de la source des données (ou celle de la destination) de choisir une direction de copie.



Source = $0000 (incrémenté) Destination = $0200 (incrémenté)	
Source = $0100 (décrémenté) Destination = $0200 (incrémenté)	
Source = $0000 (incrémenté) Destination = $0200 (décrémenté)	
Source = $0100 (décrémenté) Destination = $0200 (décrémenté)	

Ce mode permet de résoudre un problème dans certain type de copie : Lorsque que les adresses du bloc de source chevauchent les adresses de destination, il est alors possible avec le sens de copie de ne pas altérer les données pendant le transfert.

A1	Dat1
A2	Dat2
A3	Dat3
A4	
A5	
A6	

Dans ce cas de figure, si je copie en incrémentale je vais avoir un souci pour la donnée en A5.

En effet, la donnée en A3 est altérée par la première étape de copie A0→A3. La de l'étape A3→A5 on aura une erreur.

Écriture d'un code de transfert DMA

Dans notre exemple nous allons prendre un bloc de 128 caractères (ou tuile) de la BAT pour le copier 512 caractère plus bas (voir premier exemple du tableau des sens de copie) :

```
; exemple pour un BAT configurée en 32x32

_dma;

    st0 #$0F;           active le mode VRAM to VRAM
    st1 #%00000010;     je copie de VRAM à VRAM est en incrémentale
    st2 #$00;

    st0 #$10;           je choisi mon adresse de source
    st1 #$00;           soit $0000 (début de la BAT
    st2 #$00;

    st0 #$11;           je choisi mon adresse de destination
    st1 #$00;           soit $0200 (512 caractères plus bas)
    st2 #$02;

    st0 #$12;           Enfin je choisie le nombre de mot à déplacer
    st1 #$7F;           (la VRAM est une mémoire 16bits)
    st2 #$00;           ici ça sera 128-1 soit 127($7F)

wait_dma:

    lda $0000           notre fameux registre $0000 dans lequel
    and #%00010000;     nous consultons le bit 4
    beq wait_dma;       fin de copie de VRAM vers VRAM
```

Le déclenchement de la copie est réalisé après l'instruction st2 #$00.

Retour sur le HuC6280

Dans le premier chapitre nous avons eu une brève présentation du HuC6280. Bien que ce document ne soit pas dédié au processeur, nous ne pouvions pas passer à coté de certaines fonctionnalités qui deviendront pour certains d'entre vous incontournables à la programmation sous PC-Engine.

Les interruptions

Les interruptions sont des mécanismes extrêmement utiles en programmation assembleur. Elles permettent, lorsque le processeur est en cours de traitement, de faire face à une situation nouvelle (extérieur au programme) qui nécessite à traitement urgent.

Dans la vie, ce type de situation est assez courant : Vous êtes devant votre TV lorsque quelqu'un sonne à votre porte.

Synchrone et asynchrone

On appellera une interruption de type synchrone lorsqu'elle attaché à un évènement lié au code. Par exemple une division par zéro ou encore la rencontre d'une instruction illégale. Ce type d'interruption est appelé *Exception*.

Les interruptions asynchrones, ne dépendent pas du code et peuvent se produire à n'importe quel endroit du code (frappe d'une touche sur un clavier par l'utilisateur).

Système sans interruptions

Il est possible de réaliser des programmes pouvant se passer d'interruptions. Dans ce cas, c'est au programme de prendre l'initiative d'aller interroger de manière périodique les périphériques.

Ce n'est pas toujours une bonne solution : Imaginez que vous attendez un ami, mais pendant que vous l'attendez-vous avez décidé de mettre un peu d'ordre dans votre cave.

Imaginons que vous n'avez pas de sonnette à votre porte. Vous allez devoir monter régulièrement vérifier si votre ami n'est pas arrivé. Finalement votre tâche de rangement de la cave n'est pas optimale.

SI vous avez une sonnette les choses se passent beaucoup mieux. Vous vous impliqué à 100% dans votre tâche de rangement et c'est la sonnette qui vous permettra de savoir que votre ami est arrivé. Ici, la sonnette est une interruption.

Mécanisme d'interruption

Maintenant que nous savons ce qu'est une interruption, voyons ensemble comment cela fonctionne.

Le HuC6280 permet donc de mettre en place les mécanismes d'interruptions. Il possède 5 types d'interruptions :

- IRQ1 (VDC)
- IRQ2 (BRK et interruptions externes)
- NMI
- TIMER
- RESET

Ces 5 types d'interruptions sont associés à 5 vecteurs. Ceci signifie que lors qu'un événement se produit, le système arrêtera le traitement en cours pour exécuter le nouveau code pointé par l'un de ces 5 vecteurs (en fonction de l'interruption provoqué).

Sur ces 5 types, le RESET et le NMI ne peuvent être inhibé de manière logicielle. Seul les IRQ1, IRQ2 et TIMER peuvent être ignorer par le système. Pour ce faire nous utiliserons le registre d'interruption ($1402).

7	6	5	4	3	2	1	0
x	x	x	x	x	T	I1	I2

```
Mettre à 1 pour désactiver l'interruption, 0 pour
l'activer.

T : TIMER
I1 :IRQ1 (VDC)
I2 : IRQ2
```

***Note** : par défaut l'interruption NMI n'est pas connecté sur la PC-Engine*

Il est possible de lire l'état du registre d'interruption via l'adresse ($1403)

7	6	5	4	3	2	1	0
x	x	x	x	x	T	I1	I2

```
Est à 1 lorsque l'interruption est suspendue.

T : TIMER
I1 :IRQ1 (VDC)
I2 : IRQ2
```

Comme nous l'avons vu, lorsqu'une interruption est générée, le programme en cours est interrompu. Ensuite, en fonction du type de l'interruption, c'est le code pointé par le vecteur d'interruption correspondant qui sera exécuté.

Il est important d'inscrit dans le code de l'interruption au début et à la fin un mécanisme de sauvegarde et de restitution des registres A, X, Y et d'état:

```
start_irq: //début du code de d'interruption

      pha; //sauvegarde de l'accumuleur dans la pile
      phx; //sauvegarde du registre X dans la pile
      phy; //sauvegarde du registre Y dans la pile
      plp; //sauvegarde du registre d'états dans la pile
      // mon code

      ...

end_irq: //fin du code de d'interruption
      plp; //récupération du registre d'état
      ply; //récupération du registre Y
      plx; //récupération du registre X
      pla; //récupération dde l'accumulateur
      rti; //retour d'interruption
```

Après le traitement de l'interruption (appel de la commande RTI), le programme principal reprendra où il avait été interrompu. Les mécanismes de sauvegarde et de restitution permettent au programme de retrouver son "environnent" dans l'état qu'il était avant l'interruption.

L'interruption RESET

L'interruption de type RESET est automatiquement générée au démarrage à la mise sous tension de la console.

Le vecteur d'interruption RESET se trouve au l'adresses physiques (poids faible/poids fort) $1FFE et $1FFF.

Ce vecteur devra toujours pointer sur le premier programme du système.

L'interruption NMI

Les interruptions de type NMI sont des interruptions dites non masquables. C'est dire qu'il n'y à pas de moyen logiciel pour les inhiber.

Le vecteur des interruptions de type NMI se trouve au l'adresses physiques (poids faible/poids fort) $1FFC et $1FFD.

L'interruption Timer

Écrire dans le registre ($1403) permet à l'interruption TIMER de prendre en considération les prochains événements du TIMER. Si vous ne réalisez pas cette action après une interruption du TIMER vous allez rentrer dans une boucle infinie d'appel de l'adresse pointée par le vecteur TIMER.

```
stz $1403 //
```

The TIMER generates an interrupt on a single clock roll over. One clock revolution bis 1023 cycles of 7.159mhz source input. The fast setting is therefore 6.999khz.

The TIMER is interfaced through the hardware bank at address $C00. The TIMER speed is
set by a 7bit value. This value is multiplied by 1023 cycles (0=1023cycles).

$C00 7bit time value (7bit value * 1023cycles).
$C01 0=disabled, 1=enabled

When an interrupt is signaled by the TIMER unit, you must write to $1403 to reset the interrupt call or you give go into an infinite interrupt call loop. Writing any value to $1403 clears the interrupt state of the TIMER. The TIMER automatically resets on roll over and not upon writing to $1403.

L'interruption Raster (VDC)

L'interruption DMA (VDC)

Le VDC possède un DMA qu'il lui permet de réaliser des transfère au sein de la VRAM ainsi de que réaliser une synchronisation entre la VRAM vers SATB.

Chacune de ses opérations ont un temps d'exécution et lorsque que la tâche est terminée le DMA peut émettre une interruption via le VDC.

C'est ensuite à l'adresse pointer par l'interruption du VDC qui devra determiné l'origne de celle-ci.

Pour se faire, nous interrogerons le registre .....

Sauvegarde des données

La PC-Engine possède un système de sauvegarde de données. Celui-ci permet notamment la sauvegarder de vos avancée dans les jeux.

La technologie utilisé pour la sauvegarde est une RAM alimenté par une pile permettant de conserver les données même lorsque la console est éteinte.

Memory Map

Adresses Physiques	Segment #	Description	Chip Enable Signal
1FFC00 - 1FFFFF	FF	Reserved for Expansion	
1FF800 - 1FFBFF		Reserved for Expansion	
1FF400 - 1FF7FF		Interrupt Req./Disable Registers	(/CECG)
1FF000 - 1FF3FF		I/O Ports	(/CEIO)
1FEC00 - 1FEFFF		TIMER Ports	(/CET)
1FE800 - 1FEBFF		PSG Ports	(/CEP)
1FE400 - 1FE7FF		HuC6260 Ports	/CEK
1FE000 - 1FE3FF		HuC6270 Ports	/CE7
1FC000 - 1FDFFF	FE		
1FA000 - 1FBFFF	FD		
1F8000 - 1F9FFF	FC		
1F2000 - 1F7FFF	F9 - FB		/CER
1F0200 - 1F1FFF	F8	Base "scratchpad" RAM	
1F0100 - 1F01FF		Stack Page	
1F0000 - 1F00FF		Zero Page	
1EE000 - 1EFFFF	F7	Last page of HuCard memory	
004000 - 1EDFFF	02 - F6	HuCard storage	
002000 - 003FFF	01		
001FFE - 001FFF	00	Reset Vector	
001FFC - 001FFD		NMI Vector	
001FFA - 001FFB		TIMER Vector	
001FF8 - 001FF9		IRQ1 Vector	
001FF6 - 001FF7		IRQ2 Vector (for BRK)	
000000 - 001FF5		First page of HuCard memory	