

Happy C64

HAPPY C64

Happy C64

# Happy C64

Librairie C pour le *Commodore 64*  
Par Loïc Lété (Jean Monos)

v 2.4.0

## Happy C64

### \* Documentation de Happy C64 \*

Version SDK :	0.2.4.0
Update de la doc:	21/09/2022
Programmeur :	Loïc Lété
Remerciement :	Eric Boez, Lionel Paul, Olivier Cappelar, Eric Cottencin, ma chérie, mon fils.
Credit	CC65 : Mcart : Markus Brenner CC1541 : JackAsser, Krill, Claus, Björn Esser CC65cart : Joseph Rose's

### \* Caractéristiques du C64 \*

Tableau des caractéristiques du Commodore 64	
CPU	Mos 6510 0,985 MHz (pal) / 1,023 MHz(NTSC)
Mémoire Ram	64ko
Mémoire Rom	20 ko (8ko basic, 8ko kernal, 4ko tiles pré-généré)
Résolution d'affichage	320x200 pixel sans le border.
Résolution d'un pattern	8x8 pixel.
Nombre de pattern à l'écran	40x25 pattern (1000)
Nombre de sprites machine	8
Taille d'un sprite	24x21 pixel
Fonction sur les sprites	Doubler la largeur, doubler la hauteur, détection de collision avec un autre sprite / un tile.
Scrolling Hardware	Horizontal / verticale sur une amplitude de 8 pixels
Joystick	2 ports avec contrôle Haut/BAS/GAUCHE/DROITE et bouton feu
Musique	Trois voix / 8 octaves / Triangle, Dents de scie, carré/bruit
Nuancier	16 couleurs

## Happy C64

### \* Vocabulaire \*

<b>Pattern</b>	C'est tout simplement l'encodage en octet d'un élément graphique. (Tiles ou Sprite). Un Tiles carré peut être encodé avec les valeurs binaire suivant ! 0B11111111,0b10000001,0b10000001,0b10000001,0b10000001,0b10000001,0b10000001,0b10000001,0B11111111
<b>Tiles</b>	Un tile c'est un élément graphique affichable sur l'écran dans un quadrillage. Il peut représenter une police de caractère, (texte), ou des éléments du décor. On peut utiliser le terme de character , tuile ou caractère.
<b>Tilemap</b>	C'est la représentation en mémoire de l'écran. Elle se situe dans la mémoire écran (screen mémoire) en organisation linéaire. Le premier octet de la mémoire écran c'est la position 0,0 (en case) et la valeur contenu dans cette octet représente l'index du tile à afficher à cette endroit.(La taille de la tilemap fait 1000 octets)
<b>Sprite</b>	Ou Lutin, MOB (Movable Object Block), BOB, sont des éléments graphiques qui peuvent être placé au pixel près sans effacé ce qui se trouve à l'arrière. Le C64 gère 8 sprites simultanément de 24x21 points. (ou 12x21 points logique doublé sur la largeur en mode multicolore).
<b>Numéros de BIT</b>	On parle souvent de numéros de bit dans le document. C'est la place du bit dans l'octet. Sur 1 octet on numérote les bits de 0 à 7 et on compte de droite à gauche. Exemple %00100000 dans cette exemple le bit à 1 c'est le 5em bit de l'octet. En langage C, une valeur binaire s'écrit avec un préfixe 0b dans la documentation, c'est le sigle % qui se trouve au début d'une valeur binaire.
<b>Registres</b>	Un registre est une adresse mémoire dans la Ram du C64 qui permet de contrôler un composant de celui si. Par exemple le registre \$0 permet d'activer ou non la Rom du Basic, du Kernal...

# Happy C64

## Table des matières

<b>* Documentation de Happy C64 *</b> .....3	<b>* Désactiver le Mode Bitmap *</b> .....23
* Caractéristiques du C64 *.....3	<b>* Afficher un Pixel à l'écran *</b> .....23
* Vocabulaire *.....4	<b>* Modifier la couleur d'un bloc 8x8 *</b> .....23
<b>* Présentation *</b> .....7	<b>* CLS COLOR MAP BITMAP *</b> .....24
* Introduction *.....7	<b>* Configuration du mode Bitmap *</b> .....24
* Mise en place de votre projet *.....7	<b>* Les Sprites *</b> .....25
* Fichier de compilation *.....8	* Copier les datas d'un sprites d'une adresse *
<b>* Le VIC II (Video) *</b> .....11	.....25
* Introduction *.....11	* Configurer le pointeur de pattern d'un
* Configurer la plage de lecture du VIC *...11	sprite *.....25
* Activer / Désactiver l'affichage video *...12	* Afficher un sprite à l'écran *.....25
* Attendre le début du Vblank *.....12	* Cacher un sprite à l'écran *.....26
* Attendre un certain nombre de vbl *.....12	* Configurer la position d'un sprite à l'écran *
* Activer/Désactiver les interruptions *.....12	.....26
* Modifier l'emplacement de la mémoire	* Doubler la hauteur d'un sprite *.....26
écran *.....13	* Désactiver le Doubler la hauteur d'un sprite
* Modifie le nombre de colonne à afficher *	*.....26
.....14	* Doubler la largeur d'un sprite *.....26
* Modifie le nombre de ligne à afficher *...14	* Désactiver le Doubler de largeur d'un sprite
* Scrolling *.....14	*.....26
* Modifier la couleur du border *.....15	* Priorité d'un sprite par apport au tile *....27
* Modifier la couleur du Background *.....15	* Détecter les collisions d'un sprite *.....27
* Modifier la couleur du Background 1 *...15	* Activer le Mode Multicolore pour les
* Modifier la couleur du Background 2 *...15	sprites *.....28
* Modifier la couleur du Background 3 *...16	* Désactiver le Mode Multicolore pour les
* PAL/NTSC *.....16	sprites *.....28
<b>* Gestion des tiles *</b> .....17	* Choix de la Couleur individuel du sprite *
* Introduction *.....17	.....28
* Modifier l'emplacement de lecture des tiles	* Choix de la Couleur 1 des sprites *.....28
*.....17	* Choix de la Couleur 2 des sprites *.....28
* Modifier l'adresse interne de la tilmap *...18	<b>* Les commandes *</b> .....29
* Transférer des patterns au bonne endroit ! *	* Tester les deux joystick *.....29
.....18	* Tester le clavier *.....29
* Afficher un tile à l'écran *.....18	* Attendre une touche du clavier *.....30
* Récupérer l'id du charset *.....19	* Activer le touche shift *.....30
* Modifier la couleur d'une case *.....19	* Désactiver la touche shift *.....30
* Récupéré la couleur d'une case *.....19	* Tester les paddles *.....30
* Remplir la color ram d'une même couleur *	* Tester le bouton fire des paddles *.....30
.....20	<b>* La mémoire *</b> .....31
* Afficher un tile et choisir sa couleur ! *...20	* Peek et Poke *.....31
* Dupliquer un pattern horizontalement *...21	* Désactiver la rom basic *.....31
* Dupliquer un pattern verticalement *.....21	* Active la rom basic *.....31
* Activer le mode Multicolore *.....21	* Désactiver la rom Kernal *.....32
* Désactiver le mode Multicolore *.....22	* Activer la rom Kernal *.....32
* Activer Mode Étendu *.....22	* Lire un bit dans un octet *.....32
* Désactiver le Mode Étendu *.....22	* Mettre à 1 le Nème bit *.....32
<b>* Le Mode Bitmap *</b> .....23	* Mettre à 0 le Nème bit *.....32
* Activer le Mode Bitmap *.....23	<b>* Le SID *</b> .....33

## Happy C64

* Modifier le volume sonore du C64 *	33	* Binaire / Décimale / Hexadécimale *	51
* Jouer un Son *	33	* Valeur Décimale *	51
* <b>Divers</b> *	<b>34</b>	* Valeur Binaire *	51
* Générateur Pseudo Aléatoire 8 et 16 bits *	34	* Valeur Hexadécimale *	51
* Décompression RLE *	34	* L'astuce des nombres *	52
* Compression RLE *	34	* Décalage de bit ! *	52
* <b>Cassette et Disquette</b> *	<b>35</b>	* <b>Commandes Basic utiles</b> *	<b>54</b>
* Sauvegarder des datas dans un fichier *	35	* Gestion des disquettes *	54
* Charger des datas dans un fichier *	35	* <b>Les Cartouches</b> *	<b>55</b>
* Gestion des erreurs *	36	* Introduction *	55
* Un petit mode d'emplois sur les pointeurs *	36	* 8k / 16k *	55
* <b>Les Interruptions</b> *	<b>37</b>	* Compilation pour faire des cartouches 8ko/16ko *	55
* Configurer la fonction irq *	37	* <b>Happy Map</b> *	<b>56</b>
* Instruction de fin de fonction *	37	* Introduction *	56
* Code Exemple *	37	* Configurer la taille de la carte *	56
* <b>Text Engine</b> *	<b>38</b>	* Configurer la position de départ de la carte *	56
* Ordre ASCII HappyC64 *	38	* Configurer des données du meta tile *	57
* Modifier le pointeur de texte *	38	* Configurer la tableau de la carte *	57
* Afficher un texte à l'écran *	39	* Afficher la carte à l'ecran *	58
* Afficher un bloc de text à l'écran *	39	* Information technique de Happy Map *	58
* Afficher une valeur à l'écran 8 bits et 16 bits *	40	* <b>Le Langage C</b> *	<b>59</b>
* <b>REU : RAM EXPANSSION UNITE</b> *	<b>41</b>	* Introduction *	59
* Introduction *	41	* La Fonction main() *	59
* Configurer l'adresse du C64 *	41	* Le Header happyc64.h *	60
* Configurer l'adresse du REU *	41	* Les variables *	60
* Plage d'octet à travailler *	42	* Déclaration des variables *	61
* Lancer le transfère *	42	* Modifier le contenu d'une variable *	64
* <b>Index des touches du clavier</b> *	<b>43</b>	* Les conditions *	65
* <b>La carte mémoire utile du C64 pour HappyC64</b> *	<b>46</b>	* Les boucles *	69
* Des variables exploitables avant le memory screen de base *	46	* La boucle while *	69
* Le Memory Screen en configuration de base *	46	* La boucle do while *	70
* Plage pour votre programme *	47	* La boucle for *	70
* Plage du VIC en Standard *	47	* La boucle infinie *	71
* Plage de 16ko de libre *	47	* switch *	72
* Deux bloc de 8ko sont disponibles *	47	* Les tableaux *	73
* Bloc de 4 ko de libre *	47	* Les tableaux en 2 dimensions *	74
* IO/Chara et Kernal *	48	* Les fonctions et procédures *	75
* <b>ASTUCES</b> *	<b>49</b>	* Interlude du C *	79
* Le VIC-II dans la bank du Kernal *	49	* Les structures *	79
		* Vos Header *	81
		* Les defines *	83

# Happy C64

## **\* Présentation \***

### **\* Introduction \***

Happy C64 est une petite bibliothèque pour programmer le commodore 64 avec le langage de programmation C et le compilateur CC65.

L'esprit de la bibliothèque se veut être proche du système avec des fonction évolués pour simplifier le développement de votre programme. (Module de texte, gestion des tiles/couleurs...) Mais n'a pas pour but de faire le café à votre place. Une connaissance de la machine est quand même primordiale et la connaissance du langage C est obligatoire car ce guide ne vous l'apprendra pas.

### **\* Mise en place de votre projet \***

#### **1 : Les outils**

Pour programmer le commodore c64 en langage C en cross plateforme, il vous faut des outils de programmation.

- Un émulateur C64 qui permet de tester votre programme sur votre PC. VICE est sympathique. Personnellement j'utilise la suite d'émulateur de **Cloanto** : [C64 Forever](#).
- Le compilateur CC65 qui permet de compiler votre programme. Le Git possède une version CC65 qui fonctionne mais n'est pas à jour. Je vous conseille de le re télécharger.
- Un éditeur de texte pour écrire vos programmes. En vrac, notepad++, Vscode, notepad...

#### **2 : l'organisation du dossier de l'api**

Ceci est un exemple d'organisation de dossier qui fonctionne directement avec mon fichier bat. Si vous avez de l'expérience dans ce domaine, vous pouvez vous confectionner un fichier makefile et chambouler l'organisation des dossier pour adapter tout ça à votre convenance. Si vous débutez, garder cette organisation le temps de vous faire la main.

Il y a quatre dossiers majeur pour happyc64.

Le dossier principale HappyC64 qui va regrouper les trois dossier majeur.

- Le dossier **CC65** qui va contenir le compilateur CC65. Placer directement le contenu de cc65 : les dossier "bin", "cfg", "include"...
- Le dossier **hpc** qui va contenir la librairie de happyc64.
- Le dossier **projets** qui va contenir vos projets.
- Le dossier **outils** qui contient les outils divers de happyc64

Le dossier **hpc** contient un dossier **header** avec *happyc64.h* dedans et un autre fichier. *happyc64.lib* qui est la librairie de l'API en elle même.

Le dossier **outils** contient deux sous dossiers, **cc1541** qui est le logiciel de création de disquette, **makefile** qui contient le logiciel make.exe , un fichier makefile et des .bat à copier à la racine du dossier de vos projets. **Mcart64** qui est un logiciel pour créer des programmes au format cartouche. **Wav-prg** le logiciel pour créer vos projets en mode cassette.

#### **3 : organisation du dossier de votre projet**

C'est dans le dossier projets que vous aller déposer vos créations. Un dossier par création.

L'organisation de votre dossier de jeu se fait en deux dossiers majeurs , le fichier make.bat et makefile.

## Happy C64

Le premier dossier se nomme **bin**. Il va accueillir vos fichier.prg, fichier.d64 fichier.bin fichier.crt ou fichier.tap en fonction de ce que vous aller faire.

Le dossier **source** est la pour votre code source du jeu.

En fonction de vos besoin d'autre dossier peut être crée comme par exemple un dossier data pour vos fichier data.sql ou autre...

### =====

#### \* Fichier de compilation \*

### =====

*note : HappyC64 contient un makefile dans le dossier outils pour mieux automatiser la compilation. Le fichier compilation.bat devient pratiquement inutile.*

Nous allons voir le fichier de compilation.bat. C'est un fichier exécutable qui va envoyer des ordres à votre ordinateur pc windows. Il va permettre de compiler votre programme, de créer le fichier.prg par exemple, de transformer ça en .d64, et pourquoi pas de lancer le programme automatique avec vice.

CC65 doit convertir votre programme écrit en C, dans le langage du commodore 64. Les étapes simplifiées sont : votre code source en C se transforme en Assembleur. Le code Assembleur en langage machine . Puis réunir tous les fichiers transformé en langage machine pour en faire un fichier .prg. C'est le rôle de CC65

Voici un simple exemple de fichier de compilation.bat.



## Happy C64

Code : Fichier compilation.bat

```
@echo off

echo *****
echo * Fichier bat de compilation hors makefile version 2.0 du 4/10/2021 *
echo *                               version prg                               *
echo * *****

echo -----
echo * Compilation pour C64 en prg *
echo -----

rem -----
rem * Configuration du fichier de compilation *
rem -----

rem -----
rem * Nom du programme *
rem -----
set prg_prog=c64.prg

rem -----
rem * Autre lien de configuration *
rem -----
set cl65=..\..\cc65\bin\cl65
set cc65=..\..\cc65\bin\
set adr_source=source\
set adr_out=bin
set adr_happy=..\..\hpc
set fichier= %adr_source%main.c

rem -----
rem * preparation du dossier exportation *
rem -----
if exist "%adr_out%\*.prg" del %adr_out%\*.prg
if exist "%adr_out%\*.d64" del %adr_out%\*.d64
if exist "%adr_out%\*.t64" del %adr_out%\*.t64
if exist "%adr_out%\*.rp9" del %adr_out%\*.rp9
if exist "%adr_out%\*.bin" del %adr_out%\*.bin
if exist "%adr_out%\*.CRT" del %adr_out%\*.CRT

rem -----
rem * compilation *
rem -----

%cl65% -o %adr_out%\%prg_prog% -u __EXEHDR__ -O -t c64 --include-dir
%adr_happy%/header %fichier% %adr_happy%\happyc64.lib

if exist %adr_out%\%prg_prog% echo %prg_prog% present dans le dossier %adr_out%

rem -----
rem * menage *
rem -----
echo -----
echo * Effacement des fichier .o *
echo -----
pause
if exist "%adr_source%*.o" del %adr_source%*.o
pause
```

### Explication :

Dans la zone configuration, vous pouvez modifier des dossiers et des liens. Normalement si vous suivez mon organisation, il y a seulement le nom du programme à changer.  
Note : On CC65 exporte que des fichiers programmes, sur c64 forever, il faut lui faire croire que c'est une disquette, avec un .d64  
mais ça reste finalement qu'un fichier programme.  
Sur un symple vice, vous devez refaire une "vrais disquette" avec par exemple dir master.

set\_fichier : Chaque fichier C de votre programme, doit être inscrit ici avec %adr\_source%nom\_fichier.c

## Happy C64

CC65 ne gère pas les \*c comme sur SDCC 32 bits malheureusement.

Si vous cliquez dessus des messages d'erreur de fichier prg par exemple doit apparaître la premier fois mais pas grave. Le fichier prg doit apparaître dans out. Si c'est le cas braves vous avez compilé votre premier programme pour le C64.

**Note :** Il est vivement conseillé d'utiliser la norme d'encodage **ANSI** ou **UTF8** pour vos fichier. CC65 semble avoir des problèmes avec d'autre type d'encodage comme le **UTF8-Bom** par exemple.

## Happy C64

### \* Le VIC II (Video) \*

#### \* Introduction \*

Le Video Interface Chip II (VIC II) est le contrôleur vidéo du commodore 64. Il ne peut adresse que 16 ko de Ram, il doit donc être positionné dans une des 4 plages mémoire de la Ram C64 prévus à cette effet. Il n'a pas sa propre mémoire comme les consoles ou certain micro ordinateur comme la gamme des MSX par exemple.

Il permet d'avoir une résolution de 320x200 points logique (ou 160x200 point logique en multi-couleur), afficher 40x25 tiles à l'écran. Mode "Bitmap" ou tiles, une palette de 16 couleurs, 8 sprites machines, scrolling, ...

Le vic II est contrôlé par les registres compris entre **\$D000** et **\$D3FF**.

#### \* Configurer la plage de lecture du VIC \*

Le Vic II doit être positionné dans une des 4 plages mémoire du C64. C'est dans cette plage mémoire que va se trouver la mémoire écran, la lecture des sprites et des tiles. Au démarrage du C64, le VIC II se positionne à l'adresse **\$0000**

***void set\_vic\_bank(unsigned char id\_bank)***

<b><i>unsigned char id_bank</i></b>	<i>valeur comprise entre 0 et 3 (valeur Réel du tableau ci dessous) pour avoir le choix entre 4 plages mémoire.</i>
-------------------------------------	---

Le gestion des banks du VIC II se fait par l'adresse **\$DD00**. Le bit 0 et 1 contrôle la place du VIC II dans la mémoire. (Voir valeur Réel du tableau si dessous)

#### Macro des Banks de mémoire du VIC II

#DEFINE	VALEUR REEL	PLAGE DE LECTURE DU VIC II
VIC_BANK_0	3	Adresse \$0000 -> \$3FFF
VIC_BANK_1	2	Adresse \$4000 -> \$7FFF
VIC_BANK_2	1	Adresse \$8000 -> \$BFFF
VIC_BANK_3	0	Adresse \$C000 -> \$FFFF

*Note : Les numéros des banks dans les defines sont inversées par rapport au numéros des banks dans les livres techniques du C64. C'est plus logique de partie en Bas de la Ram pour la Bank 0 (Adresse \$0) et du haut de la Ram pour la Bank 3.(\$C000)*

## Happy C64

### **\* Activer / Désactiver l'affichage video \***

Les deux fonctions jouent un rôle sur l'affichage vidéo . Activer ou non l'affichage vidéo. Quand l'affichage vidéo n'est pas activé, la surface de l'écran est remplacé par la couleurs du Cadre.(Border).

C'est le bit 4 du registre \$D011 qui contrôle l'affichage vidéo.

#### **SCREEN\_ON**

*Active l'affichage vidéo.*

#### **SCREEN\_OFF**

*Désactive l'affichage vidéo. L'écran prend la couleur du Border.*

*Note: Les deux "fonctions" sont des défines.*

### **\* Attendre le début du vblank \***

#### ***void wait\_vbl()***

*Le vbblank est le signale du retour de électrons de votre TV. (enfin il n'y a plus cela sur les tv moderne mais le signale existe toujours).*

### **\* Attendre un certain nombre de vbl \***

#### ***void wait\_time(unsigned char value)***

*Permet d'attendre un certain nombre de retour d'écran. (0-255) (5 secondes maximum)*

### **\* Récupérer le niveau de ligne du balayage ecran \***

#### ***unsigned int get\_raster()***

*Permet de connaître à qu'elle ligne se situe le faisceaux.*

### **\* Activer/Désactiver les interruptions \***

Les **Interruptions** sont géré par le registre \$DCOD

#### ***Void set\_interruption\_on()***

*Active les interruptions du Commodore 64*

#### ***Void set\_interruption\_off()***

*Désactives les interruptions du Commodore 64*

## Happy C64

### \* Modifier l'emplacement de la mémoire écran \*

Le C64 réserve 1024 octets pour stocker la tilemap(1000 octets) avec les pointeurs de sprite (8 octets). L'adresse de la mémoire écran au démarrage, se trouve à l'adresse 1024. (\$0400) Il est possible de choisir un autre emplacement pour la mémoire écran indexé sur le vic.

***void Set\_adresse\_screen\_memory(unsigned char screen\_memory\_pointeur)***

<b><i>unsigned char screen_memory_pointeur</i></b>	Adresse sur Screen Memory en fonction de l'adresse vicII
--	--

Le pointeur de la mémoire écran se trouve à l'adresse \$D018 dans les 4 derniers bits. (Bits 4 à 7) ce qui fait 16 possibilités.

### #Define du SCREEN MEMORY (Adresse du VIC2 + adresse du screen memory)

#DEFINE	REEL	ADRESSE_OFFSET
SM_0000 ou SM_0	0	Adresse \$0000
SM_0400	16	Adresse \$0400
SM_0800	32	Adresse \$0800
SM_0C00	48	Adresse \$0C00
SM_1000	64	Adresse \$1000
SM_1400	80	Adresse \$1400
SM_1800	96	Adresse \$1800
SM_1C00	112	Adresse \$1C00
SM_2000	128	Adresse \$2000
SM_2400	144	Adresse \$2400
SM_2800	160	Adresse \$2800
SM_2C00	176	Adresse \$2C00
SM_3000	192	Adresse \$3000
SM_3400	208	Adresse \$3400
SM_3800	224	Adresse \$3800
SM_3C00	240	Adresse \$3C00

Note : Ceci est un offset par apport à l'adresse du VIC.

Exemple ; si le vic II est branché à l'adresse \$8000. l'utilisation de SM\_0400 placera le screen memory à l'adresse \$8400.

## Happy C64

### =====

#### \* Modifie le nombre de colonne à afficher \*

### =====

Le VIC II permet deux types affichages écran sur la largeur. 40 ou 38 colonnes. L'utilité de passer en mode 38 colonnes est de permettre de préparer un scrolling. C'est le **Bit 3** du registre *\$D016* qui contrôle cette affichage. (1=40 colonnes. 0=38 colonnes). Au démarrage le C64 est en **40 colonnes**.

#### **void set\_38\_columns()**

Le VIC II affiche 38 colonnes de largeur.

#### **void set\_40\_columns()**

Le VIC II affiche 40 colonnes de largeur. (Par défaut)

### =====

#### \* Modifie le nombre de ligne à afficher \*

### =====

Le VIC II permet de configurer deux types d'affichages écran sur la hauteur. 25 ou 24 lignes. L'utilité de passer en mode 24 lignes est de préparer un scrolling. C'est le **Bit 3** du registre *\$D011* qui contrôle cette affichage. (1=25 lignes. 0=24 lignes). Au démarrage le C64 est en **25 lignes**.

#### **void set\_24\_rows()**

Le VIC II affiche 24 lignes en hauteur.

#### **void set\_25\_rows()**

Le VIC II affiche 25 lignes en hauteur. (Par défaut)

### =====

#### \* Scrolling \*

### =====

Un scrolling c'est l'effet de déplacer "l'écran". Le commodore 64 permet de faire du scrolling verticale et horizontal.

#### **void set\_scrolling\_horizontal(signed char scroll\_x)**

<b>unsigned char scroll_x</b>	Pas du scrolling Horizontal. valeurs entre -7 et 7 (pixel)
-------------------------------	--

Permet de déplacer l'écran horizontalement d'une valeur comprise entre -7 et 7  
Les bits 0-2 du registre *\$D016* permettent de contrôler le scrolling horizontalement.

#### **void set\_scrolling\_vertical(signed char scroll\_y)**

<b>unsigned char scroll_y</b>	Pas du scrolling verticale. valeurs entre -7 et 7 (pixel)
-------------------------------	---

Permet de déplacer l'écran verticalement d'une valeur comprise entre -7 et 7  
Les bits 0-2 du registre *\$D011* permettent de contrôler le scrolling verticale.

## Happy C64

### \* Modifier la couleur du border \*

***void set\_color\_border (unsigned char color\_id)***

<b><i>unsigned char color_id</i></b>	Couleur à afficher dans le cadre de l'écran. (Border)
--------------------------------------	---

Le registre \$D020 mémorise la couleur du Border. La valeur se trouve dans les bit 0-3. (16 possibilités)

L'écran du Commodore 64 possède un contour d'une couleur unis. Cette couleur peut être paramétré. Notons que la couleur du border prend place sur tout l'écran quand le contrôleur vidéo est désactivé.

### \* Modifier la couleur du Background \*

***void set\_color\_background (unsigned char color\_id)***

<b><i>unsigned char color_id</i></b>	Couleur à afficher dans la partie du fond de l'écran.
--------------------------------------	---

Le registre \$D021 mémorise la couleur du background. La valeur se trouve dans les bit 0-3. (16 possibilités)

Le Fond de l'écran possède une couleur unis qui peut être différent de la couleur du border. Le fond de l'écran portent différent nom. (screen, Background, paper (sur CPC)).

Un pixel d'un charset non allumé laisse passer la couleur du background.

### \* Modifier la couleur du Background 1 \*

***void set\_color\_background\_1 (unsigned char color\_id)***

<b><i>unsigned char color_id</i></b>	Couleur à afficher pour le <b>background 1</b> en mode <b>multicolore</b> et <b>étendu</b> .
--------------------------------------	--

Le registre \$D022 mémorise la couleur du background 1. La valeur se trouve dans les bit 0-3. (16 possibilités)

### \* Modifier la couleur du Background 2 \*

***void set\_color\_background\_2 (unsigned char color\_id)***

<b><i>unsigned char color_id</i></b>	Couleur à afficher pour le <b>background 2</b> en mode <b>multicolore</b> et <b>étendu</b> .
--------------------------------------	--

Le registre \$D023 mémorise la couleur du background 2. La valeur se trouve dans les bit 0-3. (16 possibilités)

## Happy C64

### =====

#### \* Modifier la couleur du Background 3 \*

### =====

La couleur du Background 3 est utilisé dans le mode étendu.  
Elle permet d'afficher plus de couleur dans un pattern au niveau du FOND  
Notons que hors Raster, La couleur est commune à tous les patterns.

***void set\_color\_background\_3 (unsigned char color\_id)***

<b><i>unsigned char color_id</i></b>	Couleur à afficher pour le background 3 en mode étendu seulement.
--------------------------------------	---

Le registre \$D024 mémorise la couleur du background 3. La valeur se trouve dans les bits 0-3. (16 possibilités)

### =====

#### \* PAL/NTSC \*

### =====

***unsigned char get\_system()***

Permet de connaître si le Commodore est une version NTSC (0) ou PAL (1)



## Happy C64

### \* Gestion des tiles \*

#### \* Introduction \*

Les tiles sont des éléments graphiques de 8x8 pixels qui s'affiche à l'écran sur un quadrillage invisible. Différents type de nom sont données au tiles. (font, caractère, characters, tuile, mosaïque...) mais c'est belle et bien la même chose. Ils permettent de représenter les graphismes de votre jeu. (Murs, sol, porte...)

Le Commodore possède un jeu de tile en Rom, mais il est possible de créer vos propre set graphique. (Ce qui est recommandé de tout façon)

Une ligne d'un tile est contenu dans un octet. ( 8 bits), chaque bit dans le mode normale représente un point. (allumé ou non) et comme il y a 8 lignes, il faut donc 8 octets pour représenter les graphismes d'un tile. L'organisation des 8 octets se nomme **pattern** ! Vous pouvez aussi retrouver le terme *Générateur de caractère*.

Le c64 possède des patterns près enregistré qui permet d'afficher les lettres quand on écrit en basic. Elle se situe à l'adresse \$1000. Mais il est possible de choisir un autre emplacement de la ram qui doit se situer dans la plage lisible du VIC.

#### \* Modifier l'emplacement de lecture des tiles \*

***void set\_location\_character(unsigned char id)***

<b><i>unsigned char id</i></b>	ID emplacements des patterns par apport à l'adresse du VICII
--------------------------------	--

Le pointeur de character se trouve dans le registre \$D018 au bit 1-3. Ce qui donne 8 emplacements de départ pour lecture des patterns.	
---	--

Cette fonction permet de choisir l'emplacement de lecture des tiles.

**Note :** Tout comme le screen Memory, l'adresse est un "offset" par apport à l'adresse de départ du VIC II.

Tableau emplacement du pointeur de pattern.

ID	Adresse du pattern 0
0	\$0000
2	\$0800
4	\$1000
6	\$1800
8	\$2000
10	\$2800
12	\$3000
14	\$3800

## Happy C64

=====

**\* Modifier l'adresse interne de la tilmap \***

=====

***void set\_adresse\_tilemap(unsigned int adresse)***

<b><i>unsigned int adresse</i></b>	Adresse physique du screen memory
------------------------------------	-----------------------------------

Permet de modifier la variable interne du sdk pour pointer la mémoire écran du C64. Cette variable est utilisée par les fonctions **draw\_character()** et **draw\_full\_character()**.

*Techniquement à chaque changement de bank du vicII et du pointeur de screen memory en passant par les fonctions du sdk, cette variable est recalculé.*

**Note :** C'est une fonction très peu utilisée manuellement mais qui existe en cas ou.

=====

**\* Transférer des patterns au bonne endroit ! \***

=====

***Void set\_data\_character(unsigned int adr\_cible, unsigned char\* data\_character, unsigned char nb\_pattern)***

<b><i>unsigned int adr_cible</i></b>	Adresse Cible du transfère de data.
<b><i>unsigned char* data_character</i></b>	Pointeur (adresse) ou se trouve le 1 <sup>er</sup> pattern.
<b><i>unsigned char nb_pattern</i></b>	Nombre de pattern à transférer.

Cette fonction permet de transférer des données d'un tableau à l'endroit voulu de la mémoire. Attention c'est bien le nombre de pattern à placer en arguments et non le nombre d'octet totale. (La fonction calcule de lui même cette dernière)

=====

**\* Afficher un tile à l'écran \***

=====

La résolution du commodore 64 est de 320 x 200 pixel. Ce qui permet d'afficher 40 tiles en largeur et 25 tiles en hauteur.

La mémoire écran permet de mémoriser l'index du tiles à afficher à l'écran. (Entre 0 et 255). L'encodage des tiles est linéaire. (1<sup>er</sup> octet de la mémoire écran = position 0,0. 2<sup>em</sup> octet = position 1,0...

***void draw\_character***  
***(unsigned char position\_x ,***  
***unsigned char position\_y ,***  
***unsigned char id\_character )***

<b><i>unsigned char position_x</i></b>	Position X du pattern à afficher. (0-39)
<b><i>unsigned char position_y</i></b>	Position Y du pattern à afficher. (0-24)
<b><i>unsigned char id_character</i></b>	ID du pattern à afficher. (0-255)

Cette fonction permet donc de poser un tile à l'écran en choisissant les coordonnées X et Y. (En case)

## Happy C64

### \* Récupérer l'id du charset \*

Fonction qui permet de récupérer l'id du charset posé au coordonnées choisies dans le screen\_map.

Unsigned char get_id_character (unsigned char position_x , unsigned char position_y )	
<i>unsigned char position_x</i>	Position X du pattern à afficher. (0-39)
<i>unsigned char position_y</i>	Position Y du pattern à afficher. (0-24)

### \* Modifier la couleur d'une case \*

En mode standard l'encodage d'un tile sur C64, n'indique pas les couleurs à afficher, mais indique seulement si un point est allumé (bit 1) ou éteint (bit 0).

La couleur du groupe de point allumé est définie dans la color ram. Notons avec ce system, c'est une couleur identique par groupe de 8x8 points.

void set_color_map (unsigned char position_x , unsigned char position_y , unsigned char color_id )	
<i>unsigned char position_x</i>	Position X de la case. (0-39)
<i>unsigned char position_y</i>	Position Y de la case. (0-24)
<i>unsigned char color_id</i>	Couleur à afficher. (0-15)
Le début de la color map se trouve à l'adresse \$D800.	

### \* Récupérer la couleur d'une case \*

Cette fonction permet de récupérer le numéros de la couleur à une position voulu dans la color map.

Unsigned char get_color_map (unsigned char position_x , unsigned char position_y)	
<i>unsigned char position_x</i>	Position X de la case. (0-39)
<i>unsigned char position_y</i>	Position Y de la case. (0-24)

## Happy C64

Tableau des couleurs			
#DEFINE	Index de la couleur	#DEFINE	Index de la couleur
C_BLACK	0	C_ORANGE	8
C_WHITE	1	C_BROWN	9
C_RED	2	C_LIGHT_RED	10
C_TURQUOISE	3	C_GREY	11
C_PURPLE	4	C_GREY_2	12
C_GREEN	5	C_LIGHT_GREEN	13
C_BLUE	6	C_LIGHT_BLUE_2	14
C_YELLOW	7	C_GREY_3	15

=====

**\* Remplir la color ram d'une même couleur \***

=====

void cls_color_ram(unsigned char color)	
<i>unsigned char color</i>	Index de la couleur à remplir dans la color ram

Permet tout simplement de remplir en totalité la colors ram avec la couleur de votre choix.

=====

**\* Afficher un tile et choisir sa couleur ! \***

=====

void draw_full_character(unsigned char position_x, unsigned char position_y, unsigned char id_character, unsigned char color_id)	
<i>unsigned char position_x</i>	Position x du pattern à afficher. (0-39)
<i>unsigned char position_y</i>	Position y du pattern à afficher. (0-24)
<i>unsigned char id_character</i>	ID du pattern à afficher. (0-255)
<i>unsigned char color_id</i>	Couleur à afficher. (0-15)

Une fonction utile qui permet d'afficher un pattern à l'endroit voulu et de choisir la couleur de la case.

## Happy C64

### **\* Dupliquer un pattern horizontalement \***

```
void draw_character_line_H(unsigned char px,unsigned char py,unsigned char size, unsigned char id_character,unsigned char color);
```

<b>unsigned char px</b>	Position X de départ du pattern à afficher.
<b>unsigned char py</b>	Position Y de départ du pattern à afficher.
<b>unsigned char size</b>	Nombre de fois que le pattern va être dupliqué
<b>unsigned char id_character</b>	Index du pattern à dupliquer
<b>unsigned char color</b>	Index de la couleur du pattern à dupliquer.

Permet de dupliquer Horizontalement un pattern dans le screen memory. Utile pour tirer un trait ou créer des HUD.

### **\* Dupliquer un pattern verticalement \***

```
void draw_character_line_V(unsigned char px,unsigned char py,unsigned char size, unsigned char id_character,unsigned char color);
```

<b>unsigned char px</b>	Position X de départ du pattern à afficher.
<b>unsigned char py</b>	Position Y de départ du pattern à afficher.
<b>unsigned char size</b>	Nombre de fois que le pattern va être dupliqué
<b>unsigned char id_character</b>	Index du pattern à dupliquer
<b>unsigned char color</b>	Index de la couleur du pattern à dupliquer.

Permet de dupliquer Verticalement un pattern dans le screen memory. Utile pour tirer un trait ou créer des HUD.

### **\* Activer le mode Multicolore \***

**SET\_MULTICOLOR\_MODE\_ON**

Active le mode multicolore. (C'est une DEFINE pas besoin de ())

Chaque point d'un tile peut prendre une des 4 configurations suivante en fonction de son encodage.

Encodage de bit	Couleur à afficher
%00	Couleur du Background
%01	Couleur du Background 1
%10	Couleur du Background 2
%11	Couleur contenue dans la color ram. (Mais seulement sur une valeur comprise entre 0 et 8)

Un tile peut donc afficher 4 couleurs sur une zone de 8x8 pixels.  
- Trois couleurs général, et la couleur de sa case.

La pattern d'un tile est toujours encodé sur 8 octets.  
En mode multicolore on peut définir que des tiles de 4x8 points, mais le pixel est doublé en largeur pour arriver à la taille de 8x8 pixels. (Double pixel en largeur.)

## Happy C64

### **\* Désactiver le mode Multicolore \***

**SET\_MULTICOLOR\_MODE\_OFF**

*Désactive le mode multicolore. (C'est une DEFINE pas besoin de ())*

Permet de désactiver le mode multicolor et de repasser dans le mode standard.

### **\* Activer Mode Étendu \***

**SET\_EXTENDED\_BACKGROUND\_COLOR\_ON**

*Active le mode étendu. (C'est une DEFINE pas besoin de ())*

Le mode étendu permet d'afficher une des 4 couleurs au "fond" + la color ram du bloc 8x8 pixel.(et de rester en mode 8x8 pixel comme le mode standard)

Le choix de la couleur du fond se fait dans les 2 dernier bit du screen memory.

Encodage de bit	Couleur à afficher
%00xxxxxx	Couleur du Background
%01xxxxxx	Couleur du Background 1
%10xxxxxx	Couleur du Background 2
%11xxxxxx	Couleur du Background 3

Il ne reste que 6 bits pour choisir l'index du pattern à afficher (64 possibilités)

### **\* Désactiver le Mode Étendu \***

**SET\_EXTENDED\_BACKGROUND\_COLOR\_OFF**

*Désactive le mode étendu. (C'est une DEFINE pas besoin de ())*

## Happy C64

### **\* Le Mode Bitmap \***

#### **\* Activer le Mode Bitmap \***

**SET\_STANDARD\_HIGHT\_RESOLUTION\_BMM\_ON**

*Activer le mode Bitmap. (C'est une DEFINE pas besoin de ())*

#### **\* Désactiver le Mode Bitmap \***

**SET\_STANDARD\_HIGHT\_RESOLUTION\_BMM\_OFF**

*Désactive le mode Bitmap. (C'est une DEFINE pas besoin de ())*

#### **\* Afficher un Pixel à l'écran \***

**Void draw\_pixel(unsigned int px,unsigned int py)**

*Permet d'afficher un pixel à l'écran.*

<b>Unsigned int px</b>	Position X du pixel
<b>Unsigned int py</b>	Position Y du pixel

#### **\* Modifier la couleur d'un bloc 8x8 \***

**Void set\_bitmap\_color\_map(unsigned char px, unsigned char py, unsigned char color\_ink, unsigned char color\_background)**

*Permet d'afficher un pixel à l'écran.*

<b>Unsigned char px</b>	Position X du bloc de couleur
<b>Unsigned char py</b>	Position Y du bloc de couleur
<b>unsigned char color_ink</b>	Couleur de l'encre du bloc
<b>unsigned char color_background</b>	Couleur de fond du bloc

## Happy C64

### =====

#### \* CLS COLOR MAP BITMAP \*

### =====

**Void cls\_bitmap\_color\_ram(unsigned char ink\_color,unsigned char background\_color)**

*Permet de remplir toute la color map de la couleurs de votre choix.*

<b>unsigned char ink_color</b>	Couleur de l'encre du bloc
<b>nsigned char background_color</b>	Couleur de fond du bloc

**Void cls\_bitmap()**

*Permet d'effacer la mémoire écran en mode bitmap*

### =====

#### \* Configuration du mode Bitmap \*

### =====

Le mode Bitmap s'utilise d'une autre manière que le mode charset. voici un petit guide.

La contrainte des couleurs reste identiques au mode charset.

Le "Screen Memory" se place dans la mémoire des patterns de charset qui se configure avec la fonction `set_location_character(id)`.

Le "Color Screen" se trouve dans le screen memory du mode charset qui se configure avec la fonction `set_adresse_screen_memory(id)`;

Comme le mode standard, chaque octet de la color screen représente une zone de 8x8 pixel à l'écran. Les **4 bits** du poids **fort** représente la couleur du pixel allumé, et les **4 bits** du poids **faible** représente le fond.



## Happy C64

### **\* Les Sprites \***

Les sprites (ou lutin,MOB..) sont des graphismes qui peuvent être placé au pixel près sur l'écran. Elles ont la particularités de ne pas effacer le background. C'est le VIC qui gère les sprites. Le commodore 64 peut gérer et afficher 8 sprites à l'écran. Chaque sprite à des propriétés. (Position X, Position Y, activer ou non le sprite à l'écran, s'afficher derrière tiles, doublé la taille en hauteur un sprite, doublé la taille en largeur, tester si le sprite est en collision avec un autre sprite ou tiles, le passer en mode multicolore, et choix de la couleur personnelle du sprite.

Note : A cause de l'overlay, pour être visible, le sprite doit se trouver en px : 24 et py 50

### **\* Copier les datas d'un sprites d'une adresse \***

```
void set_sprite_data ( unsigned int adr_cible, unsigned char* adr_data,  
unsigned char nb_sprite)
```

<b>unsigned int adr_cible</b>	Adresse de destination.
-------------------------------	-------------------------

<b>unsigned char* adr_data</b>	Adresse de lecture. (Tableau par exemple)
--------------------------------	---

<b>unsigned char nb_sprite</b>	Nombre de sprite à copier.
--------------------------------	----------------------------

Tous comme les tiles, la fonction suivante permet de copier un ou plusieurs sprite(s) contenu dans un tableau / une adresse / pointeur à une adresse que vous le souhaitez.

### **\* Configurer le pointeur de pattern d'un sprite \***

```
void set_pointers_sprite(unsigned char id_sprite,unsigned char value)
```

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

<b>unsigned char value</b>	valeur du pointeur. (0-255)
----------------------------	-----------------------------

Le pattern d'un sprite est encodé sur 64 octets. Le VIC à besoin de connaître l'emplacement de départ du pattern. Celui si doit se trouver dans la plage d'adresse accessible au VIC II.(Dans sa bank)

Le pointeur est défini pour chacun des 8 sprites. Il se situe dans les 8 derniers octets de la mémoire écran.

Pour configurer le pointeur de pattern au bonne endroit c'est simple. La formule c'est valeur du pointeur (0-255) \* 64 + adresse de départ du VIC. La fonction permet de simplifier la configuration du pointeur de sprite.

### **\* Afficher un sprite à l'écran \***

```
void show_sprite(unsigned char id_sprite);
```

```
void show_sprite(unsigned char id_sprite)
```

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Permet d'afficher le sprite voulu à l'écran.

## Happy C64

=====

**\* Cacher un sprite à l'écran \***

=====

**void hide\_sprite(unsigned char id\_sprite)**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Permet de désactiver un sprite à l'écran.

=====

**\* Configurer la position d'un sprite à l'écran \***

=====

**void draw\_sprite(unsigned char id\_sprite, unsigned int position\_x, unsigned char position\_y)**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

<b>unsigned int position_x</b>	Position X du sprite
--------------------------------	----------------------

<b>unsigned char position_y</b>	Position Y du sprite
---------------------------------	----------------------

Permet de position d'un sprite à l'écran.

Attention un sprite à l'écran est visible à partir de 24 pixels horizontalement (X) et 50 pixels verticalement(Y).

La fonction gère tout seul le 9em bits de la position x pour dépasser les 255 pixel de l'écran. (C'est pour cela que la position X est un **unsigned INT** et pas un **unsigned char**).

=====

**\* Doubler la hauteur d'un sprite \***

=====

**void double\_height\_sprite\_on(unsigned char id\_sprite)**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Le sprite est "doublé" sur la hauteur.

=====

**\* Désactiver le Doubler la hauteur d'un sprite \***

=====

**void double\_height\_sprite\_off(unsigned char id\_sprite)**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Le sprite n'est plus doublé sur la hauteur.

=====

**\* Doubler la largeur d'un sprite \***

=====

**void double\_width\_sprite\_on(unsigned char id\_sprite)**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Le sprite est doublé en largeur.

=====

**\* Désactiver le Doubler de largeur d'un sprite \***

=====

**void double\_width\_sprite\_off(unsigned char id\_sprite)**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Le mode "doubler" du sprite est désactivé.

## Happy C64

### **\* Priorité d'un sprite par apport au tile \***

**void sprite\_priority\_on(unsigned char id\_sprite)**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Fait passer le sprite derrière tiles. Seul les pixel invisible du tiles affiche le sprite.

**void sprite\_priority\_off(unsigned char id\_sprite)**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Fait passer le sprite devant le tile. Seul les pixel invisible du sprite fait apparaître les couleurs du tile.

### **\* Détecter les collisions d'un sprite \***

**Unsigned char get\_collision\_sprite()**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Collision Sprite/Sprite

**Unsigned char get\_collision\_character()**

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Collision sprite/tiles

une collision se produit quand le sprite en question se superpose à un pixel non transparent d'un sprite ou d'un tile (character). La fonction renvoie 1 octet. Et en fonction si le bit de l'octet est 0 (pas de collision) ou 1 (collision) on peu connaître qu'elle sprite est en collision !

Bit activé    Sprite en collision	
Bit 7	Sprite 7
Bit 6	Sprite 6
Bit 5	Sprite 5
Bit 4	Sprite 4
Bit 3	Sprite 3
Bit 2	Sprite 2
Bit 1	Sprite 1
Bit 0	Sprite 0

## Happy C64

### **\* Activer le Mode Multicolore pour les sprites \***

```
void set_sprite_multicolore_on(unsigned char id_sprite)
```

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Tous comme les tiles, Chaque sprites peuvent passer en mode multicolore.

Dans ce mode il faut tous comme les tiles, 2 bits pour afficher un point. Ce qui permet 4 possibilités. (La couleur transparente, la couleur individuel du sprite, et deux couleurs choisis généralement pour les sprites.)

### **\* Désactiver le Mode Multicolore pour les sprites \***

```
void set_sprite_multicolore_off(unsigned char id_sprite)
```

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

Désactive le mode multicolore du sprite.

### **\* Choix de la Couleur individuel du sprite \***

```
void set_color_sprite(unsigned char id_sprite,unsigned char color_id)
```

<b>unsigned char id_sprite</b>	Index du sprite à configurer (0 à 7)
--------------------------------	--------------------------------------

<b>unsigned char color_id</b>	Couleur du sprite (0-15)
-------------------------------	--------------------------

Permet de choisir la couleur du sprite.

### **\* Choix de la Couleur 1 des sprites \***

```
void set_sprite_color_1(unsigned char color_id)
```

<b>unsigned char color_id</b>	Index de la couleur (0 à 15)
-------------------------------	------------------------------

Permet de configurer la couleur 1 des sprites.

### **\* Choix de la Couleur 2 des sprites \***

```
void set_sprite_color_2(unsigned char color_id)
```

<b>unsigned char color_id</b>	Index de la couleur (0 à 15)
-------------------------------	------------------------------

Permet de configurer la couleur 2 des sprites.

## Happy C64

### =====

### \* Les commandes \*

### =====

#### =====

#### \* Tester les deux joystick \*

#### =====

##### Unsigned char get\_joystick\_1()

Renvois le résultat du joystick 1

##### Unsigned char get\_joystick\_2()

Renvois le résultat du joystick 2

Voici les defines pour savoir la position du joystick.

DEFINE	REEL	DIRECTION
J_UP	1<<0	HAUT
J_DOWN	1<<1	BAS
J_LEFT	1<<2	GAUCHE
J_RIGHT	1<<3	DROITE
J_FIRE	1<<4	FEU

Exemple de code :

##### Code en C : Exemple Joystick 1

```
if (get_joystick_1() & J_FIRE)
{
    // Votre code si le bouton feu du joystick 1 est utilisé...
}
```

#### Attention :

Sur un vrais commodore 64 (et 128), ne pas brancher une manette de jeu megadrive/genesis car ça renvoie du 5v et le cia n'aime pas ça du tout !

#### =====

#### \* Tester le clavier \*

#### =====

##### Unsigned char get\_keyboard\_key()

Renvois le résultat de la touche testé

cette fonction permet de récupérer une valeur de la touche du clavier utilisé. Voir le tableau de macro à la fin du document pour utiliser les défines approprié au touche.

Numéros des touches en annexe

## Happy C64

### Code en C : Exemple Clavier

```
if (get_keyboard_key()==KEY_A)
{
    // Votre code si la touche A est utilisé.
}
```

=====

**\* Attendre une touche du clavier \***

=====

#### **void wait\_key(unsigned char id\_key)**

<b>unsigned char id_key</b>	Valeur de la touche testé.
-----------------------------	----------------------------

Permet d'attendre l'appuie d'une touche avant de continuer le programme.  
Note : l'utilisation de KEY\_ANY permet d'attendre n'importe qu'elle touche.

=====

**\* Activer le touche shift \***

=====

#### **set\_shift\_on()**

Active la touche shift

=====

**\* Désactiver la touche shift \***

=====

#### **set\_shift\_off()**

Désactive la touche shift et évite le combo Commodore + shift pour changer les graphismes "minuscule/majuscule" en plein jeu donc bug graphique.

=====

**\* Tester les paddles \***

=====

#### **unsigned char get\_paddle(unsigned char port,unsigned char id\_paddle)**

<b>unsigned char port</b>	Port d'entrée (1 ou 2) du paddle sur le commodore 64. (Prise DB9)
<b>unsigned char id_paddle</b>	Numéro du paddle sur le port. (1 ou 2)
Renvois la position d'un paddle. (entre 0 et 255)	

=====

**\* Tester le bouton fire des paddles \***

=====

#### **unsigned char get\_paddle\_fire (unsigned char port,unsigned char id\_paddle)**

<b>unsigned char port</b>	Port d'entrée (1 ou 2) du paddle sur le commodore 64. (Prise DB9)
<b>unsigned char id_paddle</b>	Numéro du paddle sur le port. (1 ou 2)

## Happy C64

### =====

### \* La mémoire \*

### =====

#### =====

#### \* Peek et Poke \*

#### =====

##### *unsigned char* PEEK(addr)

<i>addr</i>	Adresse 16bits
Permet de lire une valeur 1 octet à l'adresse voulue	

##### *unsigned int* PEEKW(addr)

<i>addr</i>	Adresse 16bits
Permet de lire une valeur sur 2 octets (un mot/word) à l'adresse voulue	

##### *POKE(addr, val)*

<i>addr</i>	Adresse 16bits
<i>val</i>	Valeur sur 1 octet (0-255)
Permet écrire 1 octet à l'adresse voulue.	

##### *POKEW(addr, val)*

<i>addr</i>	Adresse 16bits
<i>val</i>	Valeur sur 2 octet (0-65535)
Permet écrire 2 octets à l'adresse voulue.	

**Note :** Pour POKEW() et PEEKW(), la valeur 16 bits est décomposée dans deux octets à l'adresse 1 et adresse 2. Pour connaître la valeur 16 bit, la formule est octet de l'adresse 2 \* 256 + contenu de l'adresse 1.

#### =====

#### \* Désactiver la rom basic \*

#### =====

##### *void* set\_loram\_ram()

Désactive la rom Basic. Récupéré de la ram (8ko à partir de \$A000)
---

Note : La Rom Basic est désactivé par défaut avec happyc64

#### =====

#### \* Active la rom basic \*

#### =====

##### *void* set\_loram\_basic()

Activer la Rom Basic.
-----------------------

Note : Normalement la rom du basic est désactivé nativement avec happyc64.

## Happy C64

=====

**\* Désactiver la rom Kernal \***

=====

**Void set\_hiram\_ram()**

Désactive la rom kernal. Récupéré de la ram (8ko à partir de \$E000)

Attention à bien switcher correctement cette bank de donnée sous peine de bug !  
(Je conseille au débutant de ne pas toucher à cette partie et de garder le kernal activé. HappyC64 l'utilise pour la lecture des fichiers par exemple)

=====

**\* Activer la rom Kernal \***

=====

**Void set\_hiram\_kernal()**

Active la rom kernal.

Note : Avant de désactiver le kernal pour lire les valeurs qui se trouve à cette adresse, il faut désactiver les interruptions.  
Il faut aussi réactiver les interruptions après avoir réactivé le kernal.

=====

**\* Lire un bit dans un octet \***

=====

**unsigned char get\_bit(unsigned char id\_bit,unsigned char value)**

id_bit	Position du bit dans l'octet. (0 à 7) 0: <i>1er bit à droite</i>
value	Valeur à tester comprise entre 0 et 255. (1 octet)

La fonction permet de tester un l'état d'un bit à une position voulue !

=====

**\* Mettre à 1 le Nème bit \***

=====

**unsigned char set\_bit(unsigned char id\_bit,unsigned char value)**

id_bit	Position du bit dans l'octet. (0 à 7) 0 : <i>1er bit à droite</i>
value	Valeurs à modifier comprise entre 0 et 255. (1 octet)

La fonction permet de mettre un bit à 1 à la position voulue

=====

**\* Mettre à 0 le Nème bit \***

=====

**unsigned char unset\_bit(unsigned char id\_bit,unsigned char value)**

id_bit	Position du bit dans l'octet. (0 à 7) 0 : <i>1er bit à droite</i>
value	Valeurs à modifier comprise entre 0 et 255. (1 octet)

La fonction permet de mettre un bit à 0 à la position voulue



## Happy C64

### =====

### \* Le SID \*

### =====

Le sid est le processeur sonore du C64. HappyC64 permet de sortir du son avec le sid.

### =====

### \* Modifier le volume sonore du C64 \*

### =====

***void set\_volume(unsigned char volume)***

<b><i>unsigned char volume</i></b>	Paramètre le volume general. (0-15)
------------------------------------	-------------------------------------

### =====

### \* Jouer un Son \*

### =====

```
void set_sound(
    unsigned char voice,// VOICE_1,VOICE_2,VOICE_3
    unsigned char lb_freq,
    unsigned char hb_freq,
    unsigned char lb_pulse,
    unsigned char hb_pulse,
    unsigned char waveform,// Utiliser TRIANGLE,SAWTOOTH...
    unsigned char attaque_decay,
    unsigned char sustain_release
)
```

<b><i>unsigned char voice</i></b>	utiliser le macro VOICE_1 , VOICE_2 , VOICE_3
-----------------------------------	---

<b><i>unsigned char lb_freq</i></b>	
-------------------------------------	--

<b><i>unsigned char hb_freq,</i></b>	
--------------------------------------	--

<b><i>unsigned char lb_pulse</i></b>	
--------------------------------------	--

<b><i>unsigned char hb_pulse</i></b>	
--------------------------------------	--

<b><i>unsigned char waveform</i></b>	Utiliser le macro TRIANGLE,SAWTOOTH...
--------------------------------------	--

<b><i>unsigned char attaque_decay</i></b>	
---	--

<b><i>unsigned char sustain_release</i></b>	
---	--

### Define pour le numéro de canal

Define	Valeur Réel
--------	-------------

VOICE_1	0
---------	---

VOICE_2	7
---------	---

VOICE_3	14
---------	----

### Define pour la configuration du canal

Define	Valeur Réel
--------	-------------

TRIANGLE	17
----------	----

SAWTOOTH	33
----------	----

PULSE	65
-------	----

NOISE	129
-------	-----

## Happy C64

### =====

### \* Divers \*

### =====

#### =====

#### \* Générateur Pseudo Aléatoire 8 et 16 bits \*

#### =====

<b><i>unsigned char get_rnd(unsigned char nombre_max)</i></b>	
<b><i>unsigned char nombre_max</i></b>	Valeur entre 0 et 255

<b><i>unsigned int get_rnd16(unsigned int nombre_max)</i></b>	
<b><i>unsigned char nombre_max</i></b>	Valeur entre 0 et 65535

Fonction qui retourne une valeur comprise entre 0 et nombre\_max.  
(Utilise le générateur de bruit du SID)

#### =====

#### \* Décompression RLE \*

#### =====

<b><i>void rle_decompression(unsigned int source,unsigned int destination)</i></b>	
<b><i>unsigned int source</i></b>	Adresse des data à décompresser
<b><i>unsigned int destination</i></b>	Destination des data décompresser

HappyC64 possède une fonction simple pour décompresser des "data" au format RLE avec une contrainte ! Vos datas RLE doivent se finir par un octet qui vaut 0 pour mettre fin à la routine de décompression.

Le format RLE est simple : Voici une série d'octet au format RLE

2,4,1,0,7,5,4,4,0

Ce qui fait une fois décompressé !

4,4,0,5,5,5,5,5,4,4,4,4

#### =====

#### \* Compression RLE \*

#### =====

<b><i>void rle_compression(unsigned int source,unsigned int destination, unsigned int size)</i></b>	
<b><i>unsigned int source</i></b>	Adresse des data à décompresser
<b><i>unsigned int destination</i></b>	Destination des data décompresser
<b><i>unsigned int size</i></b>	Nombre d'octet dans le source à compresser.

La fonction : *void rle\_compression(unsigned int source,unsigned int destination,unsigned int size)*  
Permet d'appliquer une simple compression RLE. A la fin de la routine, 1 octet supplémentaire est ajouter avec la valeur 0 qui est une balise pour mettre fin à la routine de decompresion .

## Happy C64

### **\* Cassette et Disquette \***

#### **\* Sauvegarder des datas dans un fichier \***

<b>unsigned char save_file(unsigned char* name, const void* buffer, unsigned int size, unsigned char device)</b>	
<b>unsigned char* name</b>	Le nom du fichier. un , "option" permet de typer le fichier.  d : del s : Sequenciel (seq) u : USR p : PRG (default) l : REL exemple : "data,s"
<b>const void* buffer</b>	Adresse source à sauvegarder
<b>unsigned int size</b>	Nombre d'octet dans le source à sauvegarder (2 octets est ajouté sur la disquette/cassette)
<b>unsigned char device</b>	Id du device. 1 pour cassette, 8 pour lecteur disquette 1, 9 pour lecteur disquette 2 ...

Permet de sauvegarder des datas binaires dans un fichier sur disquette (ou cassette).

2 octets sont ajoutés au début du fichier.

#### **\* Charger des datas dans un fichier \***

<b>unsigned int load_file(const char* name, const void* buffer, unsigned char device)</b>	
<b>unsigned char* name</b>	Le nom du fichier. un , "option" permet de typer le fichier.  d : del s : Sequenciel (seq) u : USR p : PRG (default) l : REL exemple : "data,s"
<b>const void* buffer</b>	Adresse de destination des datas
<b>unsigned char device</b>	Id du device. 1 pour cassette, 8 pour lecteur disquette 1, 9 pour lecteur disquette 2 ...

Cette fonction permet de charger les données binaire d'un fichier dans un tableau (ou à partir d'une adresse mémoire)

Note : Les deux octets du "header" ne sont pas enregistré dans le buffer.

Note 2 : La Fonction renvoie le nombre octet chargé. Si le nombre est égale à 0, alors il y a une erreur de chargement.  
Erreur à récupérer avec get\_error()

## Happy C64

### Code en C : Exemple Ouverture de fichier

```
if (load_file("map1,s",(void*)0x8000,8)!=0)
{
    error = get_error() ;
}
```

=====  
\* Gestion des erreurs \*  
=====

### *unsigned get\_error()*

*Renvois un code erreur avec load\_files()*

=====  
\* Un petit mode d'emplois sur les pointeurs \*  
=====

Les deux fonctions pour manipuler les datas dans un fichier utilise une adresse mémoire de départ pour la lecture, ou la sauvegarde. C'est utile pour charger/sauvegarder des datas à un emplacement voulu, le buffer demande un pointeur. Voici comment les gérer si vous n'avez pas l'habitude de cela.

Le nom d'un tableau est pointeur. Donc un tableau avec le nom buffer[] :  
save\_file("sauvegarde,s",buffer,128,8) ; sauvegardera dans le fichier sauvegarde.seq, 128 octets du tableau buffer sur la disquette...

On peut utiliser un pointeur avec une adresse définie (ou autre...)  
unsigned char **buffer** =(char)0xC000

Dans ce cas là, en utilisant le mot buffer on sauvegarde 128 octets à partir de l'adresse 0xC000.

On peut placer une adresse directe dans la fonction aussi avec un cast !

Voici l'exemple avec 0xC000  
save\_file("sauvegarde,s",(void\*)0xC000,128,8);

## Happy C64

### **\* Les Interruptions \***

#### **\* Configurer la fonction irq \***

```
void init_adr_irq(unsigned int adresse);
```

*Permet de choisir une fonction qui sera appelé à chaque interruption*

Permet de choisir la fonction voulue qui sera appelé à chaque interruption.  
Exemple : `init_adr_irq((int)*compteur) ;`

#### **\* Instruction de fin de fonction \***

```
end_fonction_irq()
```

*A placer à la fin de votre fonction qui sera appelé par les interruptions*

#### **\* Code Exemple \***

##### **Code C : Exemple interruption simple**

```
unsigned char cp ;  
  
void compteur()  
{  
    // A chaque interruption, cp est incrémenté de 1  
    cp++ ;  
    end_fonction_irq() ;  
}  
  
main()  
{  
    init_adr_irq((int)*compteur);  
  
    while(1)  
    {  
  
    }  
  
}
```

## Happy C64

### =====

### \* Text Engine \*

### =====

Le SDK permet d'afficher du texte. Ceci dit une petite préparation est à effectuée. La représentation du texte dans la mémoire de caractère doit avoir une certaine organisation.

- Il suit l'organisation ASCII
- Le premier élément c'est "l'espace".
- Vous n'êtes pas obligé de placer les majuscules et minuscules pour gagner de la place. Vous pouvez représenter des minuscules dans la partie majuscule mais vous devez écrire vos textes en majuscule !
- Vous devez garder le pattern du mode vidéo appliqué. Un A en mode standard n'aura pas la même gueule qu'un A en mode Multicolor !

### =====

### \* Ordre ASCII HappyC64 \*

### =====

Voici l'ordre de vos fonts pour happyC64 et les numéros ASCII,  
(Sp = Espace)  
Il faut 93 tiles pour mémoriser la table complète.

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Sp	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	Égal	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125		
p	q	r	s	t	u	v	y	z	y	z	{		}		

### =====

### \* Modifier le pointeur de texte \*

### =====

```
void set_text_pointer(unsigned char pointeur)
```

*unsigned char pointeur*

Index du pattern pour l'espace (0-255)

Cette fonction vous permet de redéfinir la place du premier caractère (l'espace) dans votre plage mémoire de pointeur de caractère. Exemple à 0, votre jeu de police d'écriture est au début de la plage des caractères. A 5, le départ sera le 6<sup>em</sup> caractère.

## Happy C64

### **\* Afficher un texte à l'écran \***

<b>void draw_text (unsigned char px, unsigned char py, unsigned char* text, unsigned char color, unsigned char slow_wait_letter )</b>	
<b>unsigned char px</b>	Position X du texte. (Case) (0-255)
<b>unsigned char py</b>	Position Y du texte. (Case) (0-255)
<b>unsigned char* text</b>	Pointeur / Chaîne du caractères. (Mode ASCII)
<b>unsigned char color</b>	Index de couleur du texte. (0-15)
<b>unsigned char slow_wait_letter</b>	Permet d'attendre X waitvbl() entre l'affichage d'une lettre. (Permet d'afficher du texte lettre par lettre)

Cette fonction permet d'afficher du texte à l'écran.

Exemple : draw\_text(0,0,"Hello world",5,0);  
Un Hello world jaune s'affiche en 0,0

### **\* Afficher un bloc de text à l'écran \***

<b>void draw_bloc_text (unsigned char px, unsigned char py, unsigned char* text, unsigned char color, unsigned char size_ligne, unsigned char slow_wait_letter)</b>	
<b>unsigned char px</b>	Position X du texte. (Case) (0-255)
<b>unsigned char py</b>	Position Y du texte. (Case) (0-255)
<b>unsigned char* text</b>	Pointeur / Chaîne du caractères. (Mode ASCII)
<b>unsigned char color</b>	Index de couleur du texte. (0-15)
<b>unsigned char size_ligne</b>	Taille d'une ligne.
<b>unsigned char slow_wait_letter</b>	Permet d'attendre X waitvbl() entre l'affichage d'une lettre. (Permet d'afficher du texte lettre par lettre)

Note sur le size\_ligne : Permet de paramétrer un saut ligne automatique. C'est aussi utile quand vous utilisez la fonction @C pour effacer le bloc de texte.

Draw\_bloc\_text permet d'utiliser un code de fonction :

Code	Fonction
@S	Force un saut de ligne.
@W	Demande un appui de n'importe quelle touche pour la suite du texte.
@C	Permet d'effacer le texte. L'effacement se fait en fonction du size_ligne en largeur et de la dernière position Y du texte. Notons que la prochaine lettre se place au coordonné de départ. Le prochaine lettre à afficher est bien la lettre suivant @C

Note : Vous ne pouvez pas utiliser la lettre @ dans votre texte ! Risque de bug.

## Happy C64

=====

**\* Afficher une valeur à l'écran 8 bits et 16 bits \***

=====

```
void draw_text_value_8 (unsigned char px,unsigned char py,unsigned char  
valeur,unsigned char color)
```

<b>unsigned char px</b>	Position X du nombre. (Case) (0-255)
-------------------------	--------------------------------------

<b>unsigned char py</b>	Position Y du nombre. (Case) (0-255)
-------------------------	--------------------------------------

<b>unsigned char valeur</b>	Permet d'afficher une valeur sur (8bits)
-----------------------------	--

<b>unsigned char color</b>	Index de couleur du nombre(0-15)
----------------------------	----------------------------------

```
void draw_text_value_16 (unsigned char px,unsigned char py,unsigned int  
valeur,unsigned char color)
```

<b>unsigned char px</b>	Position X du nombre. (Case) (0-255)
-------------------------	--------------------------------------

<b>unsigned char py</b>	Position Y du nombre. (Case) (0-255)
-------------------------	--------------------------------------

<b>unsigned int valeur</b>	Permet d'afficher une valeur sur (16 bits)
----------------------------	--

<b>unsigned char color</b>	Index de couleur du nombre(0-15)
----------------------------	----------------------------------



## Happy C64

### =====

### \* REU : RAM EXPANSION UNITE \*

### =====

#### =====

#### \* Introduction \*

#### =====

Les extensions REU sont des modules mémoire supplémentaires pour le commodore 64. Trois modules officiels existent. Le 1700 (128ko), 1750 (512ko), et 1764 (256ko). (L'émulateur VICE permet de simuler un module extension. Pour information, C64 forever (qui utilise vice) permet de choisir entre un 256ko et 512ko.

Le module fonctionne par DMA. c.à.d il va copier la partie voulue de la Ram du C64 pour la placer dans l'extension et inversement. (Contrairement au CPC6128 qui fonctionne en mode bank switching pour sa ram supplémentaire).

On ne peut adresser que 64ko dans le REU, pour adresser les 128 à 512ko, il faut aussi désigner une "page" du REU dans les commandes de configurations. Par exemple, si je choisis l'adresse \$400 de la page 0, on va taper à l'adresse physique \$400 du REU, mais si on choisit l'adresse \$400 de la page 1, on tapera à l'adresse physique du reu en \$10400.

Dans les registres du REU, il y a 1 octet qui permet de choisir la page soit au maximum 256 pages. (On peut donc utiliser des extensions REU qui vont jusqu'à 16 Mo. La cartouche ultimate 2+ doit exploiter ça.

#### =====

#### \* Configurer l'adresse du C64 \*

#### =====

void reu_set_adresse_c64(unsigned int adresse)	
unsigned int adresse	Adresse dans le C64 (\$0 à \$FFFF)

Cette fonction permet de configurer l'adresse de départ de travail sur le commodore C64.

*Note pour l'écriture d'une adresse : Ce n'est pas un "pointeur". Utiliser directement une valeur. Par exemple 0x400 ou une variable int qui contient la valeur 0x400.*

#### =====

#### \* Configurer l'adresse du REU \*

#### =====

void reu_set_adresse_reu(unsigned int adresse, unsigned char id_bank)	
unsigned int adresse	Adresse offset dans le reu (\$0 à \$FFFF)
unsigned char id_bank	Numéros de la page dans le reu (0 à 255)

Cette fonction permet de configurer l'adresse de départ de travail du reu. Elle comprend deux valeurs, une adresse entre \$0 et \$FFFF (qui est un offset) et la page du reu entre 0 et 255. (Attention, une extension 256ko comprend moins de page qu'une extension de 512ko). Une page c'est 64ko. Une extension de 256ko comprend donc 4 pages en tout. (Donc entre 0 et 3)

*Note pour l'écriture d'une adresse : Ce n'est pas un "pointeur". Utiliser directement une valeur. Par exemple 0x400 ou une variable int qui contient la valeur 0x400.*

## Happy C64

=====

**\* Plage d'octet à travailler \***

=====

**void reu\_set\_size(unsigned int size)**

**unsigned int size**

Nombre d'octet à travailler.

Nombre d'octet que l'extension va travailler au moment du transfère. Si il faut transférer que 1ko utilisé la valeur 1024 par exemple.

=====

**\* Lancer le transfère \***

=====

**void reu\_start\_dma(unsigned char value)**

**unsigned char value**

valeur qui représente le type de transfère

Cette commande permet de lancer le transfère de Ram. 3 defines à placer dans **value** est préparé.

**Défine pour le reu\_start\_dma**

#Define	Valeur Réel	Effet
MODE_C64_REU	0b10010000	Lance le transfère du C64 => REU
MODE_REU_C64	0b10010001	Lance le transfère du REU => C64
MODE_SWAP	0b10010010	Lance le transfère en mode échange. <i>REU=&gt;C64 et en même temps C64=&gt;REU</i>

Note sur le SWAP : Les datas sont échangés entre le C64 et le REU. La Cible prend les donnés de la source, et la source prend les donnés de la cible...

Pour les deux autres commandes, ce qui se trouve dans la cible est perdu puis qu'il est remplacé par les nouvelles donnés mais les donnés de la source ne sont pas effacé.

## Happy C64

### =====

### \* Index des touches du clavier \*

### =====

Touche Alphabétique		
<u>#DEFINE</u>	<u>RÉEL</u>	<u>TOUCHE</u>
KEY_A	10	A
KEY_B	28	B
KEY_C	20	C
KEY_D	18	D
KEY_E	14	E
KEY_F	21	F
KEY_G	26	G
KEY_H	29	H
KEY_I	33	I
KEY_J	34	J
KEY_K	37	K
KEY_L	42	L
KEY_M	36	M
KEY_N	39	N
KEY_O	38	O
KEY_P	41	P
KEY_Q	62	Q
KEY_R	17	R
KEY_S	13	S
KEY_T	22	T
KEY_U	30	U
KEY_V	31	V
KEY_W	9	W
KEY_X	23	X
KEY_Y	25	Y
KEY_Z	12	Z

## Happy C64

Valeurs Numériques		
#DEFINE	Réel	Touche
KEY_0	35	0
KEY_1	56	1
KEY_2	59	2
KEY_3	8	3
KEY_4	11	4
KEY_5	16	5
KEY_6	19	6
KEY_7	24	7
KEY_8	27	8
KEY_9	32	9

Touche Divers		
Define	Réel	Touche
KEY_L_ARR	57	<-
KEY_CLR	51	
KEY_DEL	0	Del
KEY_RET	1	Retourn
KEY_DN	4	
KEY_RT	2	
KEY_STOP	63	
KEY_SPC	60	Espace
KEY_EMPTY	64	Pas de touche

Touche Arithmétiques		
Define	Réel	Touche
KEY_PLUS	40	+
KEY_MOINS	43	-
KEY_DIVISER	48	/
KEY_MULTIPLIER	49	*

## Happy C64

Touche Fonctions		
Define	Réel	Touche
KEY_F1	4	F1
KEY_F3	5	F2
KEY_F5	6	F3
KEY_F7	3	F4

## Happy C64

### \* La carte mémoire utile du C64 pour HappyC64 \*

Le Commodore 64 possède 64ko (65536 octets) de mémoire ram. (Adresse \$0 à \$FFFF). Le C64 est découpé en 4 plages de 16ko (4096 octets). (Bank).

Cette documentation va tenter de vous apporter des informations utiles pour HappyC64 et la gestion de sa mémoire.

### \* Des variables exploitables avant le memory screen de base \*

Il existe des emplacements libres et exploitable dans la première partie de la ram. Des octets par ici, des octets par là ce qui peut être très pratique pour grappiller des "variable manuel".

Mémoire disponible dans la partie System	
\$0042 , \$0052, \$033B	Trois emplacements d'1 octet de disponible.
\$00FB à \$00FE	Une plage de 89 octets disponible
\$033C à \$03FB	Une plage de 192 octets disponible. <i>Note ceci est le buffer datasette. Si vous re charger des datas par cassette la plage sera effacer.</i>
\$0CFC à \$03FF	Une plage de 4 octets disponible

### \* Le Memory Screen en configuration de base \*

A l'allumage du commodore 64, le memory screen se trouve à l'adresse \$400 et prend fin à l'adresse l'adresse \$7FF. Une plage 1000 octets est réservé pour l'affichage vidéo. Les 8 derniers octets de cette plage sont réservés pour les pointeurs de sprite.

Screen Memory		
Adresse de départ	Offset	Description
\$400	0	Début de la mémoire écran
\$7E7	999	Fin de la mémoire écran
\$7E8 à \$7F7	1000	Zone libre de 15 octets
\$7F8	1016	Pointeur de sprite 0
\$7F9	1017	Pointeur de sprite 1
\$7FA	1018	Pointeur de sprite 2
\$7FB	1019	Pointeur de sprite 3
\$7FC	1020	Pointeur de sprite 4
\$7FD	1021	Pointeur de sprite 5
\$7FE	1022	Pointeur de sprite 6
\$7FF	1023	Pointeur de sprite 7

En déplaçant le VIC2 ou / et le memory screen, il est possible de récupérer cette espace mémoire pour votre programme.

## Happy C64

### =====

#### \* Plage pour votre programme \*

### =====

C'est à partir de l'adresse **\$800** que votre programme est mémorisé avec happyc64. (Cassette ou Disquette) *L'adresse est choisie dans le fichier .cfg de CC65 pour information.*

Il y a 2 octets pour le header de lancement. Et le programme en lui même débute en **\$802**.

Votre programme comprend le HappyC64, (il faut bien mémoriser la librairie) et votre code.

La limite de vos programmes à ne pas franchir est **\$D000** et en sachant que CC65 (et le fichier de configuration de base que nous utilisons) place une pile de **2ko** avant **D000** ce qui revient à ne pas utiliser les adresses au dessus de **C800**.

### =====

#### \* Plage du VIC en Standard \*

### =====

Le vic est configuré en standard à l'adresse **\$0** de la mémoire du C64. Le vic est capable d'adresser **16ko** de mémoire Ram (et l'espace mémoire de la color map).

Configuré de base, le vic peut donc lire que la plage d'adresse comprise entre **\$0** et **\$3FFF** !

Ce qui veut dire que les patterns de sprite + patterns de tiles doivent se trouver à la fin de la taille de votre programme sans dépasser **\$3FFF** dans une configuration standard qui est vivement conseillé de modifier).

### =====

#### \* Plage de 16ko de libre \*

### =====

**16 ko** de ram sont disponibles entre l'adresse **\$4000** et **\$7FFF**.  
(Le vic II peut être déplacé dans ce bloc !)

### =====

#### \* Deux bloc de 8ko sont disponibles \*

### =====

Les deux plages d'adresses comprises entre **\$8000** à **\$9FFF** et **\$A000** à **\$BFFF**.  
**16ko** découpés en 2 Bloc de **8ko**.

Le premier bloc est prévu pour une cartouche de **8ko**. Mais reste disponible pour de la ram.

Le 2nd bloc est prévu pour 8ko aussi de cartouche (cartouche de 16ko) et la rom basic.

Ceci dit la rom basic est désactivé ce qui permet de récupérer **8ko** supplémentaires de ram.

(Le vic II peut être déplacé dans ce bloc qui peut être une place de choix pour nos programme avec happyC64 et en plaçant le memory screen au début du bloc ou pourquoi pas à la fin).

### =====

#### \* Bloc de 4 ko de libre \*

### =====

**4 ko** est libre de l'adresse **\$C000** à **\$CFFF**.

Ceci dit CC65 place **2ko** pour la pile à la fin de ce bloc.

Il ne faut donc pas utiliser la mémoire à partir de **C800**.

Le vicII peut être déplacé en **\$C000** si vous voulez utiliser les 8ko du kernal pour la mémoire video.

## Happy C64

### =====

#### \* IO/Chara et Kernal \*

### =====

Les deux dernières bank de la ram sont partagés par **8ko** dédiés au i/o, et à la rom de caractère :(**\$D000**) et le Kernal (**\$E000**)  
Je suis bien tenté de dire pas touche à ça petit con.  
Il est possible de désactivé le Kernal pour retrouver **8ko de Ram** mais ceci dit, ça plante si vous ne faite pas gaffe. (Il y en a qui ont essayé, et ils ont eu des problèmes)



## Happy C64

### =====

### \* ASTUCES \*

### =====

#### =====

#### \* Le VIC-II dans la bank du Kernal \*

#### =====

Le **kernal** débute à l'adresse **\$E000** jusque à la fin de la ram du c64 **\$FFFF**. Le kernal c'est le system exploitation du commodore 64, c'est une ROM. c.a.d que les informations sont seulement en lecture en **\$E000**, il y a aussi de la RAM ou nous pouvons lire et écrire qui se trouve à notre disposition. Un truc cool c'est que si nous voulons écrire dans un emplacement d'une ROM, ba ça passe directement en RAM. Mais l'inverse non. Si la rom est activé, si on veux la lire, c'est dans la rom que nous allons taper.

Il est possible de désactiver la ROM du kernal pour lire la RAM. Ceci dit, sans faire attention, ça fait buger le programme. Je conseille donc de ne pas faire ça. Ceci dit c'est **8ko** de ram perdu ! Ou pas. Il est possible d'utiliser les 8ko de ram en tant que mémoire video et même sans désactiver le kernal. Le vic-II lui ira chercher les donnés en RAM directement, et comme on peut écrire directement en RAM sans désactiver la rom.

#### -----

#### - Déplacer le vic-2 dans la dernier bank mémoire du vic. -

#### -----

Le vic 2 ne peut adresser que **16ko** de mémoire. A la mise sous tension de la machine, il se trouve au début de la ram. **\$0000**  
Il faut le déplacer à la fin de la ram dans l'une des 4 positions possibles du vic-2 à l'adresse **\$C000**

La commande à utiliser est : *set\_vic\_bank(VIC\_BANK\_3);*

#### -----

#### - Déplacer le screen memory au début de la plage du kernal -

#### -----

Pour le moment le screen memory se trouve en offset **\$400** par apport à l'adresse du vicII soit à l'adresse **\$C400**

Cela ne va pas, il faut le déplacer dans la partie occupé du kernal en **\$E000**  
Une fonction permet de déplacer le screen memory, on va donc le déplacer à l'offset **\$2000** par apport au VIC-II (**\$C000 + \$2000 = \$E000**)

*set\_adresse\_screen\_memory(SM\_2000) ;*

Il est possible de le déplacer à la fin avec un **SM\_3C00**. (**\$FC00**)

#### -----

#### - Déplacer le pointeur de character -

#### -----

C'est la 3em manipulation à faire. Déplacer le pointeur de caractère pour lire les tiles dans la bank du kernal.

Il y a quatre possibilités pour les character.

La fonction pour déplacer le pointeur est *set\_location\_character(id)*  
les 4 possibilité sont :

- 8** : pour placer les tiles en **\$E000** donc en début kernal
- 10** : pour les placer en **\$E800**. (Il y a une plage de 400 octet de libre entre le screen memory et le premier tiles)
- 12** : pour placer les tiles en **\$F000**
- 14** : pour placer les tiles en **\$F800**

## Happy C64

### ----- - Mots sur le pointeur de sprite !!! - -----

Le pointeur de sprite (0) débute au début du VIC2 en \$C000.  
Vous avez 2ko de libre pour placer des pattern à cet endroit avant d'être dans la zone de la pile pour votre programme C. (32 patterns).

Vous pouvez placer vos pattern de sprite dans le kernal. Mais attention à ne pas prendre l'adresse \$E000 comme offset 0 pour calculer votre emplacement des sprites et de bien configurer la machine pour faire cohabiter pattern de tiles et de sprites. (Un pattern de sprite c'est 64 octets).

### ----- - Mots sur le pointeur de tiles !!! - -----

Tout comme les sprites, il est possible de placer les tiles \$C000 (cela fait 256 patterns de tiles).

Ceci dit la zone du kernal fait 8ko, on peut très bien regrouper le screen memory, les patterns de tiles et les patterns de sprites en même temps pour libérer un max de place pour votre programme.

### ----- - Notre nouvelle ram card - -----

Notre nouvelle ram card se décompose donc comme ceci :

ADRESSE	TAILLE	NOTES
\$0000 à \$03FF	1ko	Variable System
\$0400 à \$07FF	1ko	Ram libre
\$0800 à \$3FFF	14ko	pour votre programme ou autre data. (Début du programme (Sauf cartouche))
\$4000 à \$7FFF	16ko	pour votre programme ou autre data.
\$8000 à \$BFFF	16ko	pour votre programme ou autre data. (Zone des cartouches)
\$C000 à \$C7FF	2ko	pour votre programme ou autre data.
\$C800 à \$CFFF	2ko	PILE pour votre programme en C. (Variable Local)
\$D000 à \$DFFF	4ko	Variable I/O (Pas touche à ça petit con!!!)
\$E000 à \$FFFF	8ko	KERNAL et MÉMOIRE VIDÉO (SCREEN MEMORY et PATTERN)

Vous avez un espace de 43 ko pour créer votre jeu en bien vous organisant. Notons que votre programme se loge à l'adresse \$800 ce qui vous laisse une bonne 40en de ko pour écrire votre programme si vous désirez ne pas passer par un système de chargement de fichier data en plein jeu.

# Happy C64

**\* Binaire / Décimale / Hexadécimale \***

**\* Valeur Décimale \***

Nous avons l'habitude de ce système de notation. Nous l'utilisons à longueur de journée. Le système décimale travaille en base de 10. C'est à dire qu'une unité à le choix de 10 valeurs. La valeurs souhaiter est comprise entre 0 et 9. (0,1,2,3,4,5,6,7,8,9) ce qui fait bien 10 possibilités.

Quand on ajoute 1 à 9, notre unité repart à 0, et l'unité de gauche, une valeur de 1 est ajouté à l'unité de gauche.

Exemple :  $9 + 1 = 10$   
 ou pour mieux comprendre :  $09 + 1 = 10$   
 autre exemple :  $99 + 1 = 100$   
 décomposition :  $099 + 1 = 0(9+1)0$   
 $\quad\quad\quad = 100$

Sur happyC64 l'écriture décimale se fait simplement en écrivant nos valeurs directement sans pré fixe.

**\* Valeur Binaire \***

Les valeurs binaires sont la base de l'informatique. Pour comprendre le principe, c'est très simple, soit le courant passe, ou le courant ne passe pas. On peut voir ça aussi en fonction de la lumière. Il y a de la lumière ou pas.

Le système binaire travaille sur une base de 2. Une unité à que deux possibilités. Une valeur comprise entre 0 et 1.  
Tous comme les valeurs décimale. Ajout 1 à la valeur 1 revient à 0 dans notre unité et à ajouté 1 à l'unité de gauche.

Exemple :

1+0	=	1
1+1	=	10
11+1	=	100

Sur HappyC64 (et surtout CC65) nous pouvons utiliser directement des valeurs binaires. Pour cela il faut utiliser le préfixe **0b**

Exemple : 0b10100101

Dans les assembleurs ou représentation, une valeur binaire peut avoir comme préfixe %

Exemple :%10100101

**\* valeur Hexadécimale \***

3em forme très utilisé en programmation. La forme Hexadécimale qui est une base de 16. Donc une unité permet 16 valeurs. Pour son écriture, après le 9, nous utilisons les lettres de l'alphabet. Ce qui fait :  
(0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)

Ajouter 1 à 9, ne donne pas 10 mais A.  
Ajouter 1 à F donne 10. On passe l'unité à 0 et on ajoute 1 à l'unité de gauche.

## Happy C64

Sur HappyC64 nous pouvons aussi utiliser directement des valeurs hexadécimale.  
Pour cela il faut utiliser le préfixe **0x**  
exemple : 0x400

Notons que dans d'autre langage ou "norme", vous pouvez rencontrer le préfixe **\$**  
\$400

voir un **H** au début du nombre. H400

Les valeurs décimale sont très très utilisé pour les adresses dans la ram.  
Sur Commodore 64, l'adresse débute à 0x0 pour finir en 0xFFFF

### =====

#### \* L'astuce des nombres \*

### =====

Voici des petites tips à comprendre.

1 octet c'est une valeurs comprise entre 0 et 255 en décimale.  
C'est aussi une valeur comprise entre 0 et 11111111 en valeurs binaire. (8 bits)  
et aussi une valeurs comprise entre 0 et FF en Hexadécimale.

Si on décompose FF en hexadécimale, C'est bien 4 bits à gauche et 4 bits à droite.

Hexadécimale : F/F  
Binaire : 1111/1111  
Décimale : 15 / 15  
(255)

Si on décompose F1  
Hexadécimale : F/1  
Binaire : 1111/0001  
Décimale : 15 / 1  
(241)

### =====

#### \* Décalage de bit ! \*

### =====

Très utile en programmation. Sachez que votre commodore 64 ne sait pas faire de multiplication et de division. Demander à votre C64 de faire ce genre d'opération revient en gros à faire des additions à gogo. Donc on mange du temps machine et de la ram.

Par contre votre commodore 64 connaît une technique simple qui est de décaler des bits d'un octet à droite ou à gauche.  
Et ça, il fait très très très bien le faire et rapidement surtout.

A vrais dire je dis, le commodore 64, mais c'est le processeur de l'ordi qui fait ça...

Nous allons voir deux / trois trucs avec le décalage de bit.

#### Etude de cas :

%00000001

Cette valeur binaire représente 1 décimale.

Maintenant nous allons décaler le "1" binaire d'une unité à gauche.

%00000010

ce qui représente %10 soit 2

## Happy C64

Nous allons continuer de nouveau en décalant encore d'une unité à gauche.

%00000100

soit%100 qui est égale à 4.

Ce qui nous donne la suite suivant. 1,2,4 puis 8,16,32,64,128,256

Nous pouvons voir que décaler les bits binaires d'une unité revient à multiplier par 2. De déclarer de deux unités, revient à multiplier par 4...

En C le signe de décalage vers la gauche est <<

Et le décalage à droite est >>

1<<1 revient à faire  $1 \times 2$ .

16<<2 revient à faire  $1 \times 4$

32>>4 revient à faire  $32/4$

Notons que le décalage de bit n'est pas une opération prioritaire.

1>>2x2 donnera 1>>4

Penser à mettre des parenthèses dans les décalages de bit.

$(16>>2) \times 2 = (16/4) \times 2 = 4 \times 2 = 8$

Voilà la raison que dans le monde rétro, on aime travailler en base de 2 pour faciliter le décalage de bit et le travaille sur les nombres.

## Happy C64

### **\* Commandes Basic utiles \***

#### **\* Gestion des disquettes \***

Les programmes de HappyC64 sont transformés directement en ASM. Le lancement des disquettes sur émulateur sont souvent automatisées. Sur un vrais C64, ce n'est pas le cas. Voici un petit guide rapide de gestion de disquettes.

Chaque périphérique porte un numéro d'identification. Pour le lecteur de disquette, le numéro id débute à 8 pour le lecteur 1 jusque à 30.

La commande de chargement de donnée en mémoire :  
**load "nom du fichier",numéro du périphérique,adresse**  
Ce qui donne par exemple :  
**load "prog",8,1**

*Voici des exemples :*

*Par défaut j'utilise le périphérique 8*

*\* Charger le premier programme en mémoire \**

**load "\*",8,1**

*\* Charger le fichier "prog" en mémoire \**

**load "prog",8,1**

*\* Lancer le programme \**

**run**

*\* Charger le contenu de la disquette en mémoire \**

**load"\$",8**

*\* Lister le contenu de la disquette après chargement \**

**list**

*\* Formater une disquette \**

**OPEN 15,8,15:PRINT#15,"NEW:NOM,2A":CLOSE 15**

## Happy C64

### **\* Les Cartouches \***

#### **\* Introduction \***

Le Commodore 64 possède un port cartouche à l'arrière de la machine. (Pas très pratique j'en conviens).

Elle permet de brancher des boîtiers qui contiennent des roms et un programme dessus. L'avantage c'est que le contenu de la rom se logue très très mais vraiment très rapidement en mémoire !

Il existe plusieurs types de cartouche.

#### **\* 8k / 16k \***

Ce sont les cartouches de type standard au Commodore 64. Elles permettent d'avoir des programmes de 8ko ou 16ko de donnée.

Les cartouches 8k se logent à l'adresse \$8000-\$9FFF.  
Les cartouches 16k se logent à l'adresse \$8000-\$BFFF.

#### **\* Compilation pour faire des cartouches 8ko/16ko \***

Le dossier **cartridge** contient les fichiers de configuration (*.cfg*) et les fichiers binaires de paramétrage (*.o*) pour créer des programmes sur cartouche standard 8/16 ko.

La compilation de votre programme ne donnera pas un prg ou un d64 mais deux fichiers. Un fichier **bin** et un fichier **crt**.

Le fichier bin est le programme brut qui permet d'être copié sur les puces. (Ou à découper pour les programmes 16ko).

Le fichier CRT possède un header pour être utilisé dans les émulateurs.

Le Makefile permet de compiler pour les cartouches :

- 8ko (make crt8k)
- 16ko (make crt16k)

## Happy C64

### **\* Happy Map \***

#### **\* Introduction \***

Happy Map est une serie de fonction pour afficher simplement une "map" à l'écran. La mise en place est très simple.  
Happy Map fonctionne en Meta Tile, c.a.d qu'il faut configurer les 4 tiles (caractère) et sa couleur, pour 1 meta tile.  
Happy Map permet de mémoriser 64 meta tile au même moment.  
Attention Happy map ne permet pas de faire scrolling ou de map qui dépasse l'écran.(Pour le moment ?)

#### **\* Configurer la taille de la carte \***

Votre carte doivent avoir une dimension qui représente le nombre de meta\_tile qui seront afficher horizontalement et verticalement.

Deux fonctions existe pour gérer la taille de la map.  
Il est vitale de bien configurer la taille de la map avant d'utiliser la fonction pour configurer le tableau de la map et le draw map car sinon vous allez avoir des problèmes d'affichage.

```
void hm_set_map_size_x(unsigned char value)
```

unsigned char value	Taille de la map sur sa largeur (Meta Tile) (0-255)
---------------------	---

```
void hm_set_map_size_y(unsigned char value)
```

unsigned char value	Taille de la map sur sa hauteur (Meta Tile) (0-255)
---------------------	---

#### **\* Configurer la position de départ de la carte \***

Deux fonctions existe pour positionner la map dans l'écran.  
Attention cette fois si la valeur de configuration s'exprime en caractere/tile qui est la grille virtuel du commodore 64 pour placer les caractere(tiles/tuiles...) à l'écran.

```
void hm_set_map_offset_x(unsigned char value)
```

unsigned char value	Position x sur l'écran de la map (case) (0-255)
---------------------	---

```
void hm_set_map_offset_y(unsigned char value)
```

unsigned char value	Position y sur l'écran de la map (case) (0-255)
---------------------	---



## Happy C64

### **\* Configurer des données du meta tile \***

Chaque meta tile doit être configuré pour que la fonction d'affichage de la map puisse récupérer les bonnes données.

Chaque meta tile possède 8 données.

Les 4 premiers données sont l'index des caractères qui représente le meta tile. (Haut-Gauche / Haut-Droite / Bas-Gauche/Bas-droite)

Et les 4 données suivantes sont tous simplement l'index de couleur de l'encre des 4 tiles qui compose le meta tile.

```
void hm_set_tile_data(  
unsigned char id_mt,  
unsigned char id_1,  
unsigned char id_2,  
unsigned char id_3,  
unsigned char id_4,  
unsigned char color_1,  
unsigned char color_2,  
unsigned char color_3,  
unsigned char color_4)
```

unsigned char id_mt	Index du meta tile à paramétrer. (0-63)
unsigned char id_1	Index du tile Haut_Gauche du Meta_tile (0-255)
unsigned char id_2	Index du tile Haut_Droite du Meta_tile (0-255)
unsigned char id_3	Index du tile Bas_Gauche du Meta_tile (0-255)
unsigned char id_4	Index du tile Bas_Droite du Meta_tile (0-255)
unsigned char color_1	Couleur du tile Haut_Gauche
unsigned char color_2	Couleur du tile Haut_Droite
unsigned char color_3	Couleur du tile Bas_Gauche
unsigned char color_4	Couleur du tile Bas_droite

### **\* Configurer la tableau de la carte \***

Très simple d'utilisation. On modifie le tableau en fonction de trois valeurs. Position X, Position Y et l'index du meta-tile, Les positions X et Y sont cette fois si exprimés en "Meta-tile"

```
void hm_set_buffer_map(unsigned char px,unsigned char py,unsigned char value)
```

unsigned char px	Position X du meta_tile dans la map (meta_tile) (0-255)
unsigned char py	Position Y du meta_tile dans la map (meta_tile) (0-255)
unsigned char value	Id du meta tile à cette emplacement de la map.

## Happy C64

### **\* Afficher la carte à l'écran \***

Cette fonction permet tout simplement de déclencher l'affichage de la map à l'écran.

<b>void hm_draw_map()</b>	
<b>unsigned char</b> value	Position X sur l'écran de la map (case) (0-255)

### **\* Information technique de Happy Map \***

2 Tableaux et 4 variables en Global sont déclaré quand on utilise happy map.

<b>Variables et Tableaux de Happy Map</b>	
<b>unsigned char</b> hm_size_x	Taille X de la map
<b>unsigned char</b> hm_size_y	Taille Y de la map
<b>unsigned char</b> hm_offset_x	Position X de la map
<b>unsigned char</b> hm_offset_y	Position Y de la map
<b>unsigned char</b> tiles_buffer[512]	Data des meta_tiles
<b>unsigned char</b> buffer_map[240]	Organisation de la map.

## Happy C64

### =====

### \* Le Langage C \*

### =====

#### =====

#### \* Introduction \*

#### =====

**Note : Le chapitre sur le Langage C est pour le moment incomplet.**

Le langage C est un langage compilé. Vous allez écrire votre programme dans ce langage, et un logiciel va traduire votre code dans des valeurs binaire compréhensible par la machine cible. (Ici le commodore 64 et surtout le processeur 65010). Pour cela, le logiciel utilisé est la suite CC65.

Le basic que vous utilisez sur Commodore 64, est un langage interprété. Le commodore 64 va lire vos instruction Basic en temps réel, et va faire la moulinette ligne par ligne pour transformer ça en langage machine. Cela prend donc du temps, et c'est pour cela que le basic est "lent". Surtout sur les machine 8bits avec des "petits" processeur.

Cette partie du manuel n'a pas vraiment but de vous apprendre le C. C'est tellement vaste, mais cela peut être un bon guide pour débiter.

#### =====

#### \* La Fonction main() \*

#### =====

La fonction main() est obligatoire dans votre code. C'est la porte d'entrée de votre jeu. Ce n'est pas tout à fait exacte, car internement il y a plein de chose qui se passe avant le main() mais pour vous, on débute par un main(). Le code minimum est :

#### La fonction main()

```
void main()
{
    // instruction ;
}
```

**void** est un typage de la fonction. C'est pour savoir qu'elle type de donnée la fonction va renvoyer. Ici c'est void. Cela se traduit par **Rien**. La fonction main() ne renvoie donc rien.

Main est suivie de parenthèse. () Entre les parenthèse on place des arguments. Nous verrons plus tard comment exploiter des arguments. Notre main ne possède pas argument, donc les () sont vide.

La fonction main() est entourée de {}. { c'est l'ouverture d'un bloc et } c'est la fermeture d'un bloc.

// instruction ;  
// c'est le début d'une ligne de commentaire sur une ligne. Le compilateur va ignorer cette ligne.  
Les commentaire peuvent être multi ligne avec cette syntaxe.

```
/*  
Ligne un de commentaire  
ligne deux de commentaire  
*/
```

Et chaque "instruction" doit se finir par un point virgule. ;

## Happy C64

### =====

#### \* Le Header happyc64.h \*

### =====

Nous avons pas encore un code minimum pour happyc64. Il faut aussi dire au programme où se trouve le fichier header de la librairie.

Un fichier header regroupe des signatures de fonction. Pour faire simple, ça dit au compilateur : Et gros dans ce fichier tu peux utiliser la fonction X ou Y ...

Le fichier happyc64.h contient toutes les signatures de la librairie. Pour l'appeler il faut utiliser une fonction **include()** voici le nouveau code minimum

### Le header

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    // instruction ;  
}
```

**#include** ne se fini pas par une point virgule car ce n'est pas une instruction C a proprement parlé.  
C'est une instruction propre du compilateur pour lui demander d'inclure le contenu du fichier xxx.h dans le fichier actuel avant de passer tout ça au compilateur.

### =====

#### \* Les variables \*

### =====

Les variables sont super important dans un langage de programmation. C'est des conteneurs qui permet de mémoriser des valeurs. Par exemple la variable vie\_pj pour garder en mémoire le nombre de point de vie du personnage joueur. Pour PX, PY...

Contrairement au Basic du C64, (et des basic 8 bits), l'écriture du nom d'une variable est sensible à la case. PV est différent de pv par exemple.

Autre truc à savoir, les variables mémorisent seulement des nombres. Pour du texte, il faut regrouper des variables l'une à côté de l'autre et mémoriser les valeurs ASCII correspondant au lettre. On verra ça plus tard.

Le Basic c64 possède trois types de variables.

- Les variables Chaîne qui permet de mémoriser des chaîne de caractère. (Du texte quoi),
- Les variables en nombre entier,
- Les variables en nombre avec virgule. (Valeur réel)

C'est ce qu'on appelle le typage des variables.

En langage C c'est pareil, chaque variable doit être configuré pour un rôle.

## Happy C64

- La variable peut mémoriser 1 octets.
- La variable peut mémoriser 2 octets.
- La variable peut mémoriser 4 octets.
- La variable peut mémoriser des valeurs réel.

En plus de cela une variable peut dit être signé. (On peut travaillé avec des valeurs négative) ou non signé. (Que des valeurs positives.)

Nous allons voir un petit tableau utile pour HappyC64 des typages.

Typage des variables avec le compilateur CC65				
Typage	Signification	(Octets)	Plage	Commentaire
signed char	Caractère	1	-128 à 127	Représente 1 octets dans la mémoire. C'est un caractère.
Unsigned char	Caractère non signé	1	0 à 255	Représente 1 octets dans la mémoire. C'est un caractère
signed int	Entier	2	-32768 à 32767	Sur certain compilateur, c'est comme un long.
Unsigned int	Entier non signé	2	0 à 65535	Sur certain compilateur, c'est comme un long.
Signed long	Long	4	-247483648 à 247483647	
Unsigned long	Long non signé	4	0 à 4294967295	

Ce sont les 6 typages que vous allez souvent utiliser en programmation sur happyc64.

Il existe d'autre typage comme short qui est l'équivalent d'un int. Float et double. Mais trop grouement en place.

Un petit mot sur le int. Le int peut prendre 2 ou 4 octets en fonction des compilateurs (et configuration des compilateurs). Sur le compilateur CC65 utilisé par happy c'est 2 octets. Mais si vous faites du C sur d'autre compilateur, la règle n'est peut être pas vrais.

(En principe sur les machine 16/32 bits le int est comme le long) et faut utiliser short pour du typage à 2 octets.

Deuxième truc à savoir, signed char et unsigned char. On peut aussi typé sans marqué signed ou un unsigned. Donc char. Ceci dit en fonction des compilateur un char simple peut être signé ou non signé de base. Et sur une mise à jour, les règles du jeu peuvent changer. (le compilateur SDCC à déjà opéré ce changement).

Donc je conseille de forcer le signed ou le unsigned quand vous allez déclarer une variable. Déclarer une variable ? Oui prochain chapitre.

### =====

#### \* Déclaration des variables \*

### =====

En basic, si on veut utiliser une variable, on pose un nom et on injecte la valeur dedans. (10 vie = 20) Quand le programme va arriver sur cette ligne, il va réserver de la mémoire si la variable n'existe pas et lui coller l'étiquette vie dans cette espace.

## Happy C64

En C, ça ne marche pas trop comme ça. Il faut dire au compilateur un truc du genre: Bon la tu vois avant que j'utilise ma variable vie, tu vas réserver x octets et lui coller l'étiquette vie. Après je vais utiliser cette espace comme je voudrais...

Et truc cool, le x octets ba vous le maîtriser. C'est tous simplement le typage de la variable. Vous pouvez dire au compilo de réserver 1 octets, 2 octets et 4 octets avec l'étiquette vie !

Pour déclarer une variable, c'est simple : Typage + nom de la variable voulue + point virgule.

### Les variables

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    // + ----- +  
    // * Déclaration des variables *  
    // + ----- +  
    unsigned char vie ;  
}
```

Dans cette exemple, j'ai déclaré une variable "vie" de type char et non signé. Je peux utiliser la variable vie dans mon programme.

Enfin pas partout. Car en C il existe une notion de portée des variables. Une portée c'est la zone ou la variable peut exister, et être utilisé. En gros une variable existe dans la zone {}. On dit qu'elle est locale.

Dans notre exemple elle est dans la zone void main(){}  
Elle est locale à main. Si on fait une autre fonction, vie ne peux pas fonctionner dans l'autre fonction.

### Les variables

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    // + ----- +  
    // * Déclaration des variables *  
    // + ----- +  
    unsigned char vie ;  
}
```

## Happy C64

```
// =====  
// * Fonction exemple() *  
// =====  
void exemple()  
{  
    // + -----+  
    // * Déclaration des variables *  
    // + -----+  
  
    // La variable vie du main ne peut pas fonctionner ici  
}
```

Notons qu'on peut déclarer de nouveau une fonction vie dans la fonction exemple.  
Ceci dit la variable vie du main() n'est pas la même que la variable vie de exemple().

Il y a un moyen de rendre une variable utilisable dans notre fonction main() et exemple().

C'est de la déclarer en dehors des fonctions. (En haut). Donc d'en faire une variable global.

Exemple :

### Les variables

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// + -----+  
// * Déclaration des variables global *  
// + -----+  
unsigned char vie ;  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    // La variable vie est utilisable dans main()  
}  
  
// =====  
// * Fonction exemple() *  
// =====  
void exemple()  
{  
  
    // La variable vie est utilisable dans exemple()  
}
```

- Attention au variable global. Son utilisation peut entraîner la foudre de certain extrémiste du C !

- Les variables (sur cc65) se déclare au début d'une fonction. (Certain compilateur autorise la déclaration de variables en pleine fonction. (Instruction ; déclaration de variable ; instruction...) mais je déconseille

## Happy C64

cette méthode. Déclarer ses variables au début d'une fonction permet de garder une lisibilité.

-A encore un petit mot sur les variables globales. Sa portée n'est valable que sur son fichier. En C on peut faire plusieurs fichiers.c et les regrouper pour faire son programme. La variable globale vie du fichier 1 n'est pas accessible dans le fichier 2, fichier 3... enfin pas nativement...

-On peut aussi déclarer plusieurs variables sur une même ligne si c'est le même type. Exemple : unsigned char vie,xp,hp ;

### =====

#### \* Modifier le contenu d'une variable \*

### =====

Le contenu d'une variable peut (et doit être) modifier sinon cela ne va pas trop servir xd), être modifier. C'est très simple.

"Nom de variable" "opérateur" "valeur ou nom de variable ou autre mais ça on verra plus tard"

Exemple : vie = 5 ;

vie va prendre la valeur 5.

vie = vie + 5 ;

vie va reprendre son contenu et ajouter 5 à cette valeur pour la réinjecter dans vie.

Opérateur général		
Signe	Signification	Commentaire
+	Addition	Permet de faire une addition
-	Soustraction	Permet de faire une soustraction
*	Multiplication	Permet de faire une multiplication
/	Division	Permet de faire une division
=	Affectation	Permet de modifier la variable par une constante.
++	Incrémenter	Ajoute 1 à la variable (variable++)
--	Décrémenter	Retire 1 à la variable.(variable--)
+=	Addition	Additionne directement la valeur à la variable.
-=	Soustraction	Soustrait directement la valeur à la variable.
*=	Multiplication	Multiplie directement la valeur à la variable.
/=	Division	Divise directement la valeur à la variable.



## Happy C64

### =====

#### \* Les conditions \*

### =====

Un programme sans conditions, cela va être compliqué. Nous avons besoin de tester des valeurs pour travailler. En langage C, le mot clef de condition c'est if. Voyons voir un peu une simple construction de condition.

#### Condition simple

```
// =====
// * Fichier include *
// =====
#include <happyc64.h>

// =====
// * Fonction main() *
// =====
void main()
{
    // + ----- +
    // * Déclaration des variables *
    // + ----- +
    unsigned char vie = 5 ;

    // + ----- +
    // * Condition sur vie *
    // + ----- +

    if (vie == 0)
    {
        // Instruction ;
    }
}
```

Dans notre fonction main() on déclare une variable vie, de type unsigned char et on injecte dès le départ la valeur 5 dedans.

Plus bas je fais un if (vie == 0) {}

Le If c'est une condition. Entre parenthèse nous avons vie. On va tester la variable vie. Nous avons un double égale. On va tester une égalité et une valeur 0. Puis les {}

Le programme va entrer dans les {} si vie est égale à 0.

si vie est égale à 0 ; alors le test est vrais. Si vie n'est pas égale à 0, alors le test est faux et on n'entre pas dans les accolades. On passe directement à la fin de l'accolade.

#### Piège à con !

Pour tester une égalité, c'est le double égale "==" et non pas un seul égale "="

En basic le test d'égalité c'est un seul égale. Attention à ne pas vous mélanger les pinceaux...

A oui pas de " ;" à la fin d'une condition. Ce n'est pas une instruction. Voici le tableau des opérandes de conditions.

## Happy C64

Opérateur de comparaison		
Signe	Signification	Commentaire
==	Egalité	Permet de tester une égalité.
<	Infériorité strict	Plus petit que !
<=	Infériorité	Plus petit ou égale que
>	Supériorité strict	Plus grand
>=	Supériorité	Plus grand ou égale
!=	Différent	Différente de

Il est possible de faire du multie teste dans une condition. Par exemple j'ai besoin de tester deux variables pour jouer ce qu'il y a dans les accolades.

Pour cela on va utiliser ce que nous appelons des opérateurs logiques.  
(OU / ET)

Voici un exemple.

### Condition simple

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    // + ----- +  
    // * Déclaration des variables *  
    // + ----- +  
    unsigned char vie = 5 ;  
    unsigned char immortel = 1 ;  
    // + ----- +  
    // * Condition sur vie *  
    // + ----- +  
  
    if ( (vie == 0) && (immortel == 0) )  
    {  
        // Instruction ;  
    }  
}
```

Dans cette exemple le && représente un "ET".  
Pour que le test soit vrais, il faut que la variable vie soit égale à 0 ET que la variable immortel soit aussi égale à 0.

Si une des deux test est faux, la conditions est fausse, donc on n'entre pas dans les accolades.

voici les trois opérateurs logiques du langage C

## Happy C64

Opérateur logique de condition		
Signe	Signification	Commentaire
	OU logique	Besoin d'une seul condition soit vrais
&&	ET logique	Besoin que toutes les conditions soit vrais
!	Non Logique	Inverse l'état

Allons un petit plus loin. Avec le jeu des parenthèse on peut faire des groupes de condition.

```
If ( (Variable a || variable b) && (variable C) )
{
}
```

Alors là pour que la CONDITION soit vrais. Il faut :  
que le groupe variable C soit vrais ET que le groupe variable A/B soit vrais aussi.

Pour que le groupe A/B soit vrais, dans notre exemple il faut seulement qu'une des deux variable (A ET B) soit vrais !(A cause du ou de ce groupe de parenthèse)

Avec un peu d'entraînement vous allez comprendre. C'est juste de la logique.

Nous avons pas fini avec les conditions !!! On va parler du mot clef **else**

oué else placé à la fin permet tous simplement d'activer un groupe d'instruction si la condition renvoie faux !!!

```
else

// =====
// * Fichier include *
// =====
#include <happyc64.h>

// =====
// * Fonction main() *
// =====
void main()
{
    // + ----- +
    // * Déclaration des variables *
    // + ----- +
    unsigned char vie = 5 ;
    unsigned char immortel = 1 ;
    // + ----- +
    // * Condition sur vie *
    // + ----- +

    if ( (vie == 0) && (immortel == 0) )
    {
        // Instruction ;
    }
    else
    {
        // Instruction si la condition n'est pas vérifié.
    }
}
```

## Happy C64

Ce n'est pas fini, il y aussi les **else if** la c'est pour faire une suite de condition les un à coté des autres mais si une condition est vérifié, à la fin de son accolade on test pas les autres. Si la condition n'est pas vérifié, on passe au else if suivant.

Voici sa construction.

### Else if

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    // + ----- +  
    // * Déclaration des variables *  
    // + ----- +  
    unsigned char vie = 5 ;  
    unsigned char immortel = 1 ;  
    // + ----- +  
    // * Condition sur vie *  
    // + ----- +  
  
    if ( (vie == 0) && (immortel == 0) )  
    {  
        // Instruction ;  
    }  
  
    else if ( (vie == 1 && (immortel == 0) )  
    {  
        // Instruction ;  
    }  
  
    else if ( (vie == 2 && (immortel == 0) )  
    {  
        // Instruction ;  
    }  
  
    else  
    {  
        // Instruction si la condition n'est pas vérifié.  
    }  
}
```

Dans cette exemple ça revient à vérifier la variable vie si elle vaut 0,1 ou 2 l'exemple est un peu con, on verra plus tard qu'il y a mieux à faire. Mais l'idée est de comprendre.

Si la variable vie = 0 (et immortel aussi hein) on va entrer dans ses accolades. Une fois à la fin de son accolade on passe directement à la fin de la condition sans tester les autres !

Si vie n'est pas à 0 alors on va tester la prochaine entrée donc vie est égale à 1. Et on continue...

## Happy C64

Les conditions sont donc puissantes et vitales en programmation. C'est une fondamentale de la programmation.

### **\* Les boucles \***

Ah les boucles, les boucles permettent de répéter des actions et vous allez voir c'est super utile. Il existe trois types de boucle en C.

- while
- do while
- for

Les trois boucles ne fonctionnent pas tout à fait de la même manière. Nous allons voir cela tout de suite !

### **\* La boucle while \***

La boucle while va tester au départ une condition. Si la condition est vraie va entrer dans les accolades. Et à la fin de l'accolade va renvoyer le programme au début de la boucle et faire un nouveau test !

Si le test est faux, on saute la boucle et on continue le programme. C'est aussi simple que ça.

La construction d'une boucle while est la suivante :

```
while(condition)
{
    instruction ;
}
```

### Boucle while

```
// =====
// * Fichier include *
// =====
#include <happyc64.h>

// =====
// * Fonction main() *
// =====
void main()
{
    // + ----- +
    // * Déclaration des variables *
    // + ----- +
    unsigned char i = 0 ;

    // + ----- +
    // * Compteur *
    // + ----- +
    while(i<10)
    {
        i++ ;
    }
}
```

Cette exemple permet d'ajouter 1 à i dans chaque boucle. A la sortie de la boucle i est égale 10.

## Happy C64

### =====

#### \* La boucle do while \*

### =====

Alors la boucle do while est un peu différente du while. Le programme va entrer obligatoirement dedans, jouer les instruction et à la fin de boucle va faire le teste de condition. Si la condition est vraie, on retourne au début de la boucle. (donc on entre dedans) , si le test est faux, le programme continue.

La construction d'une boucle while est la suivante :

```
do
{
    instruction ;
} while(condition) ;
```

Attention au **point virgule** à la fin de while !!!

### Boucle do while

```
// =====
// * Fichier include *
// =====
#include <happyc64.h>

// =====
// * Fonction main() *
// =====
void main()
{
    // + ----- +
    // * Déclaration des variables *
    // + ----- +
    unsigned char i = 12 ;

    // + ----- +
    // * Compteur *
    // + ----- +

    do
    {
        i++ ;
    } while(i<10) ;
```

Dans cet exemple je n'ai pas initié i à 0 mais à 12. A la sortie i prendra la valeur de 13. Le test se fait à la fin et on entre obligatoirement dans la boucle.

### =====

#### \* La boucle for \*

### =====

La boucle for permet d'entrer dedans un nombre de fois définie par le programmeur. Personnellement j'utilise beaucoup cette boucle.

La construction est simple :

```
for (initiation de la variable; condition ; modification de la variable)
{
    instruction ;
}
```

Attention entre parenthèse c'est des points virgules pas des virgules)

## Happy C64

voici un petit exemple :

### Boucle for

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    // + -----+  
    // * Déclaration des variables *  
    // + -----+  
    unsigned char i;  
  
    // + -----+  
    // * Compteur *  
    // + -----+  
  
    for (i=0;i<10;i++)  
    {  
  
    }  
}
```

Quand le programme entre pour la première fois dans la boucle, i prendra la valeur de 0. Les instructions sont jouées et à la fin de boucle dans cette exemple i est incrémenté de 1.

On retourne au début de la boucle et on fait le test pour savoir si i est inférieur à 10. Si c'est vrais, on rentre de nouveau dans la boucle (sans initier la variable à 0 dans notre exemple) et on continue le cheminement.

Si la condition est fausse, on ne rentre pas dans la boucle et le programme continue.

### \* La boucle infinie \*

Attention au boucle infinie ! Cela peut être un piège à la con. On ne peut pas sortir d'une boucle infinie, le programme est donc planté. Un exemple de boucle infinie est présenté plus bas. On initie i à 0. On entre dans la boucle. Force i à 0. (Aucune utilité de faire ça en plus car i n'est pas modifié dans cette boucle mais bon) et comme la condition est toujours vrais, on entre toujours dans la boucle ... Et l'espace temps est déchiré xd.

### Boucle infinie

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{
```

## Happy C64

```
// + -----+
// * Déclaration des variables *
// + -----+
unsigned char i = 0 ;

// + -----+
// * Compteur *
// + -----+
while(i<10)
{
    i=0 ;
}
```

Sur Commodore 64 (et autre machine ou console), il faut faire une boucle infini pour rebouclé le programme. Ba oui sur ses machines pour quitter le programme c'est le bouton off ! Donc dans certain cas, ça ne dérange pas.

Un while(1) {} permet de faire une boucle infinie. 1 veut dire "vrais" quand une condition est vrais, c'est 1. Quand c'est faux c'est 0.

```
=====
* switch *
```

```
=====
```

Dans la partie condition, nous avons réalisé plusieurs if/else if pour tester le contenu d'une variable. En langage C, il y a un raccourcie. La commande switch qui va récupérer le contenu d'une variable et en fonction de valeur, on va pouvoir réaliser des instruction.

Sa construction est simple :

```
switch(variable)
{
    case 1 :
        instruction ;
        break ;

    case 2 :
        instruction ;
        break ;

    default :
        instruction ;
}
```

case x : c'est la valeur de la variable testé. Case 1 va tester si la variable = 1. si oui on entre dedans si c'est faux le programme va tester la prochaine case.

Break veux dire : va a la fin de switch pour continué le programme. Donc si on place pas de break, le programme va rester dans le switch et va tester la prochaine case x. (Cela peut être utile!) et default permet d'entrer obligatoirement dans l'instruction si il n'y pas de case testé. (Enfin quand le programme arrive la sans break).

Notons que pour case, c'est pas obligatoirement 1,2,3,4 heins !



## Happy C64

### switch

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
  
    // + ----- +  
    // * Déclaration des variables *  
    // + ----- +  
    unsigned char i = 5 ;  
  
    // + ----- +  
    // * switch *  
    // + ----- +  
    switch(i)  
    {  
        case 0 :  
            i = 50 ;  
            break ;  
  
        case 1:  
            i = 80 ;  
            break ;  
  
        case 8 :  
            i = 255 ;  
            break ;  
  
        default :  
            i=11 ;  
  
    }  
}
```

### =====

#### \* Les tableaux \*

### =====

Les tableaux en C, est une variable qui contient plusieurs case pour mémoriser des valeurs. Les cases sont identifiées par un numéro !

En langage C qui dit variable dit déclaration de variable, et les tableaux ne font pas exception. Il faut déclarer le tableau avec un type et le nombre de case qu'il va contenir.

Par exemple mon tableau tab avec 10 cases en mode unsigned char.  
Chaque case prendra donc 1 octet en mémoire et donc mon tableau 10 octets.

```
unsigned char tab[10] ;
```

Voilà une initiation d'un tableau tab avec 10 cases.

Pour modifier le contenu d'un tableau c'est simple, nom du tableau [index]=contenu ;

Attention il y a un piège ! En langage C, le début d'un index c'est 0 et non 1. (On dit que les tableaux sont indexés sur 0) note : Certain langage de programmation (lua par exemple) indexe ses tableaux sur 1.

## Happy C64

Donc dans notre exemple, un tableau de 10 case aura pour index :

0,1,2,3,4,5,6,7,8,9

Donc pour modifier la case 5 :

```
tab[4]=69 ;
```

Et pour lire un tableau c'est facile aussi. On va mémoriser dans i la valeur de la case 5 :

```
i = tab[4] ;
```

il est aussi possible d'initier des valeurs directement dans un tableau comme les variables.

```
Tab[]=
{ 1,2,3,4,5,6,7} ;
```

cette exemple permet de créer automatiquement un tableau de 7 cases et initier son tableau des valeurs 1 à 7 au case 1 à 7.

Attention au point virgule après la dernière accolade.

### Tableau

```
// =====
// * Fichier include *
// =====
#include <happyc64.h>

// =====
// * Fonction main() *
// =====
void main()
{
    // + ----- +
    // * Déclaration des variables *
    // + ----- +
    unsigned char i = 5 ;
    unsigned char tab[5];

    // + ----- +
    // * Exemple boucle for et tableau *
    // + ----- +
    for(i=0;i<5;i++)
    {
        tab[i]=50 ;
    }
}
```

Dans cette exemple, on déclare un tableau de 5 case. Et dans la boucle on va placer toutes les cases avec la valeur 50 de notre tableau.

On remplace tab[4] par exemple par tab[i] et la case utilisé est tout simplement le contenu de i.  
(De 0 à 4 dans notre exemple)

### **\* Les tableaux en 2 dimensions \***

En langage C, il existe aussi le concept de tableau en 2d. C'est comme la bataille naval, pour retrouver une case en particulier, il faut deux coordonnées. X et Y. La c'est pareil.

## Happy C64

Unsigned char tab[10][5] ; permet de déclarer un tableau de 5 lignes avec 10 colonnes sur chaque ligne.

Tab[5][3]=3 ; permet d'injecter la valeur 3 à la 6em lignes et la 4em colonnes. (n'oubliez pas que les tableaux sont indexer sur 0 en C)

Ce genre de tableau peut être utile pour mémoriser des truc à des coordonné X / Y par exemple même si on peut faire la même chose sur un tableau en 1d. (C'est même beaucoup plus recommandé).

Pour simuler un tableau 2d dans un tableau 1d c'est simple : la formule est la suivante : Position Y \* le nombre de colonnes simulé sur une ligne + position X et vous avez le numéros de la case dans un tableau 1d !

### **\* Les fonctions et procédures \***

Bon, voici un premier gros morceau du C. Les fonctions. (et procédure). Le C ne fait pas de distinction entre fonction et procédure. Mais c'est quoi la différence ? En gros une procédure c'est une fonction qui ne renvoie pas de résultat. Et une fonction, c'est une procédure qui renvoie un résultat !

Oui mais ?

Bon, une fonction (et procédure) c'est un morceau de code qui est appelé, puis à la fin de celle si retourne de là ou il est appelé. (Comme le couple GOSUB/RETURN en BASIC pour mieux comprendre)

Comment ce construit notre fonction/procédure !

```
Typage nom_fonction(paramètre ou rien)
{
    instruction ;
}
```

A oui, comme les variables, il faut déclarer ses fonctions/procédure. Sa déclaration est simple, au début du fichier :  
on reprend :  
typage nom\_fonction(paramètre ou rien) ;  
on ajoute le point virgule à la fin !

Et pour appeler notre fonction/procédure juste :

nom\_fonction(paramètre ou rien) ;

Voici un exemple qui va appeler notre boucle for :

### Procédure

```
// =====
// * Fichier include *
// =====
#include <happyc64.h>

// =====
// * Déclaration des procédure *
// =====
void play_for() ;

// =====
// * Fonction main() *
// =====
void main()
{
    play_for() ;
```

## Happy C64

```
}  
// =====  
// * Procedure play_for() *  
// =====  
void play_for()  
{  
    // + -----+  
    // * Déclaration des variables *  
    // + -----+  
    unsigned char i  
    unsigned char tab[5];  
  
    // + -----+  
    // * Exemple boucle for et tableau *  
    // + -----+  
    for(i=0;i<5;i++)  
    {  
        tab[i]=50 ;  
    }  
}
```

Voilà un exemple. On a typé notre fonction avec un void. Ce qui la transforme en procédure et donc à la fin, la fonction ne renvoie rien.

On a bien déclaré au début du programme notre play\_for()  
en dessous de main, on a créé notre procédure play\_for et effectué des instructions.

Attention, les variables dans les procédures sont dites locales ! Elles ne vivent que dans les procédures. Une fois sorties de la fonction/procédure, elles disparaissent. (Plus de i, plus de tab[...])

Il est possible d'injecter des valeurs externes d'une procédure par son appel.  
Par exemple, je voudrais que la valeur 50 injectée dans le tab(i) soit une valeur que je choisis au moment de l'appel de la procédure !

Pour cela dans le nom de la procédure et de sa déclaration, dans la parenthèse on va déclarer une variable et son type.

Exemple :

Déclaration de la procédure

```
void play_for(unsigned char valeur);
```

procédure

```
void play_for(unsigned char valeur)
```

```
{
```

```
    ...
```

```
}
```

Appel :

```
play_for(30) ;
```

Par exemple, on va injecter 30.

## Happy C64

### Procédure

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Déclaration des procédure *  
// =====  
void play_for(unsigned char valeur) ;  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    play_for(30) ;  
}  
  
// =====  
// * Procédure play_for() *  
// =====  
void play_for(unsigned char valeur)  
{  
    // + ----- +  
    // * Déclaration des variables *  
    // + ----- +  
    unsigned char i  
    unsigned char tab[5];  
  
    // + ----- +  
    // * Exemple boucle for et tableau *  
    // + ----- +  
    for(i=0;i<5;i++)  
    {  
        tab[i]=valeur ;  
    }  
}
```

Dans cette exemple on récupère la valeur envoyer dans les paramètres. (30 ici) elle se stock dans la variable locale valeur et tab[i]=valeur ;

Il est bien sur possible de faire plusieurs argument.  
On va par jouer sur le nombre de boucle à faire. (Attention dans notre exemple à ne pas sortir du tableau mais pour comprendre un peu c'est pas grave !

### Procédure

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Déclaration des procédure *  
// =====  
void play_for(unsigned char valeur,unsigned char boucle_max) ;  
  
// =====  
// * Fonction main() *  
// =====  
void main()
```

## Happy C64

```
{
  play_for(30,3) ;
}

// =====
// * Procedure play_for() *
// =====
void play_for(unsigned char valeur,unsigned char boucle_max)
{
  // + ----- +
  // * Déclaration des variables *
  // + ----- +
  unsigned char i
  unsigned char tab[5];

  // + ----- +
  // * Exemple boucle for et tableau *
  // + ----- +
  for(i=0;i<boucle_max;i++)
  {
    tab[i]=valeur ;
  }
}
```

Voilà, on vient de créer un 2nd argument boucle max.  
Dans l'appelle on a ajouté une virgule et envoyé un 2nd paramètre.

Nous allons voir maintenant les fonctions ! Et donc le renvoi d'une valeur.  
Pour cela il faut typer notre procédure avec autre chose que void ! (Comme les variables tout simplement.)

A la fin, on fait un return value

voici un petit programme pour comprendre !

### Fonction

```
// =====
// * Fichier include *
// =====
#include <happyc64.h>

// =====
// * Déclaration des procédure *
// =====
unsigned char get_tab_value(unsigned char i) ;

// =====
// * Fonction main() *
// =====
void main()
{
  // + ----- +
  // * Déclaration des variables *
  // + ----- +
  unsigned char get_value ;

  get_value = get_tab_value(6) ;
}
```

## Happy C64

```
// =====  
// * Fonction get_tab_value() *  
// =====  
unsigned char get_tab_value(unsigned char i)  
{  
    // + ----- +  
    // * Déclaration des variables *  
    // + ----- +  
  
    unsigned char value ;  
  
    unsigned char tab[]=  
    {  
        23,41,8,6,12,47,69,23,12,14,78,96,36,251  
    };  
  
    value = tab[i] ;  
    return value ;  
}
```

Voici le petit programme. On déclare en unsigned char une fonction get\_tab\_value() avec un paramètre unsigned char i.

Après le main on crée cette fonction.

On déclare un tableau avec des valeurs dedans et une variable value

value prend la valeur lue dans le tableau tab[] à l'index i.  
et on retourne la valeur value.

Dans main, on appelle la fonction avec un paramètre et mémorise la valeur dans get\_value.

Variable = appelle\_de\_fonction() ; permet d'appeler la fonction et de mémoriser la valeur de retour dans la variable.

On peut faire la même chose dans les conditions pour information.

If (nom\_fonction() == 5){...}

Dans cet exemple le nom\_fonction() est appelé, et le résultat du retour est comparé à 5 pour savoir si la condition est vraie !

C'est cool non. Mais le souci en C, une fonction ne renvoie qu'un seul résultat. Pour modifier / renvoyer divers variables, il existe moult méthodes. (Getter/Setter like , Variable Global , pointeur (Celui là vous ne les aimez pas))

### =====

#### \* Interlude du C \*

### =====

Nous avons vu (rapidement je sais) la base du C.  
(Variable, boucle, tableau, condition, et fonction).

Travaillé un peu ça, documenté vous un peu sur ça. Vous avez une base pour créer des petits programmes en C avec HappyC64.

Si vous ne maîtrisez pas ses 5 concepts, ce n'est pas la peine de vous plonger sur la suite. Si vous avez déjà programmé, normalement c'est bouéno.  
Pour les débutants, ne pas hésiter à vous documenter et suivre des vrais tutos.

### =====

#### \* Les structures \*

### =====

Les structures est un concept sympathique en C. Pour faire simple et imagé, cela permet d'attacher des sous variables, à une grande variable. Bon en

## Happy C64

réalité c'est pas vraiment ça. Les structures se rapproche hachement de l'objet on va dire. Vous allez déclarer différentes variables et tout ça vous allez les rattacher à un nom. Et enfin avec ce "nom". Le nom donné est en faite un nouveau typage de variable. Bon c'est pas claire (ou léon) je vais expliquer avec un exemple vous allez vite comprendre.

Je vais créer des caractéristiques pour des personnages.

Point de vie  
Défense  
Attaque

Tous les personnages joueur et non joueur de mon programme possède ses trois caractéristiques.

On va dire que défense et attaque sont typé en unsigned char. Et PV en unsigned int.

En tant normale on devrait faire un truc du genre.

```
unsigned int PJ_PV ;  
unsigned char pj_def ;  
unsigned char pj_att ;
```

et pareil avec chaque personnage du jeu, monstre bla bla bla.  
(ou utiliser des tableaux aussi)

Ba la avec les structures on va faire autrement.  
On va définir un groupe de variable sous une étiquette.

Exemple dans mon exemple str\_caracteristique et dans le bloc on déclare des variables.

```
struct str_caracteristique  
{  
    unsigned int PV ;  
    unsigned char def ;  
    unsigned char att ;  
};
```

Ne pas oublier le point virgule à la fin !!!

et enfin dans votre programme, vous pouvez déclarer une variable dont le typage porte le nombre de votre structure.

Exemple :

```
str_caracteristique pj;  
str_caracteristique monstre_A;  
str_caracteristique monstre_B;
```

Dans cette exemple pj, monstre\_A et monstre\_B sont un conteneur. Elles possèdent tous des "sous variable" PV def et att qui leurs sont propre. Les pv de pj, ne sont pas les pv du monstre\_A.

Pour taper dans la variable PV de pj c'est simple :  
pj.pv = valeur ;

et pour le monstre\_A c'est  
monstre\_a.pv = valeur ;

On peut construire des tableaux de monstre aussi.

```
str_caracteristique monstre[5] ;  
permet de créer 5 monstres dont chaque monstre possède les trois  
caractéristiques cité plus haute.
```



## Happy C64

Monstre[1].pv = 50 ; va modifier les pv du monstre 1. (Attention un tableau débute par 0. (sauf en déclaration) )

### Structures

```
// =====  
// * Fichier include *  
// =====  
#include <happyc64.h>  
  
// =====  
// * Déclaration de la structure *  
// =====  
struct str_caracteristique  
{  
    unsigned int PV ;  
    unsigned char att,def ;  
}  
  
// =====  
// * Fonction main() *  
// =====  
void main()  
{  
    // + ----- +  
    // * Déclaration des variables *  
    // + ----- +  
    struct str_caracteristique PJ ;  
    struct str_caracteristique boss ;  
    struct str_caracteristique monstre[5];  
  
    // + ----- +  
    // * Exemple d'utilisation de structure *  
    // + ----- +  
  
    PJ.PV = 50 ;  
    boss.PV = 300 ;  
    boss.att = 10 ;  
    monstre[3].def = 8 ;  
}
```

Tester les structures, ça va vous changer la vie !

#### =====

#### \* Vos Header \*

#### =====

En programmation C, il est vivement recommandé de créer vos propre header. Jusque ici, on utilise le happyc64.h qui est le header de la librairie. Il contient la déclaration de fonction, et autre joyeuseté de la programmation.

Les headers est pratique pour déclarer des fonctions, des macro aussi (on varra ça au prochain chapitre, et même écrire vos structure...

Le fichier header contient une extension en .h et non en .c il faut protéger chaque header pour éviter la répétition de celui si.

Quand vous créer votre header, il faut faire une condition dedans. Une condition en langage du compilateur qui n'est pas du C.

## Happy C64

```
#ifndef NOM
#define NOM

// votre code

#endif
```

NOM doit être le même avec le ifndef et define  
et NOM ne doit pas être réutilisé dans d'autre define du programme.  
(Plus tard les définit!)

A partir de là dans l'espace votre code vous pouvez déclarer les fonctions de votre programme et du (ou des) fichier que le header sera lié, et des structures par exemple.

Exemple :

```
#ifndef NOM
#define NOM

// votre code
void ma_fonction() ;

struct ma_structure
{
    unsigned char ma_variable ;
    unsigned char ma_variable_bis ;
} ;

#endif
```

Pour lier le fichier .h à votre fichier.c c'est simple on utilise la fonction du compilateur #include  
la deux cas s'annonce soit le nom de fichier entre <,,,> comme happyc64.h  
ou entre deux ".,.,"

le premier cas, faut placer votre h dans un certain dossier dont le compilateur connaît pour les .h  
(bon nativement en utilisant les fichiers de compilation donné, il va chercher des .h dans le dossier hpc,header. Donc pas très pratique pour nous.

Le mieux c'est utiliser les " ... "  
là c'est vous qui indiqué ou chercher le.h

exemple :

```
#include "header/mon_header.h"
```

La ça va chercher le fichier mon\_header.h dans le dossier header présent dans le source si vous respecté les conseils de happyc64 dans arborescence de votre projet.

Il est possible de placer des #include dans des fichier includes. C'est ce que je fais.

Un exemple avec deux fichier, main.c et main.h

### Fichier main.c

```
// =====
// * Fichier include *
// =====
#include <happyc64.h>
```

## Happy C64

```
#include <header/main.h>

// =====
// * Fonction main() *
// =====
void main()
{
    // + ----- +
    // * Déclaration des variables *
    // + ----- +
    unsigned char get_value ;

    get_value = get_tab_value(6) ;
}

// =====
// * Fonction get_tab_value() *
// =====
unsigned char get_tab_value(unsigned char i)
{
    // + ----- +
    // * Déclaration des variables *
    // + ----- +

    unsigned char value ;

    unsigned char tab[]=
    {
        23,41,8,6,12,47,69,23,12,14,78,96,36,251
    };

    value = tab[i] ;
    return value ;
}
```

### Fichier main.h

```
#ifndef MAIN_H
#define MAIN_H

// =====
// * Déclaration des procédure *
// =====
unsigned char get_tab_value(unsigned char i) ;

#endif
```

```
=====
* Les defines *
=====
```

Nous allons parler des Constantes ! Très très très très utile. Les Defines. Une Define c'est tout simplement une étiquette sera remplacé par ce que nous voulons au moment de la compilation.

L'initiation d'une define se fait par #define NOM ce\_que\_nous\_voulon

## Happy C64

Exemple :

```
#define VIE_MAX 50
```

Si on reprend notre structure d'avant sur PJ.

```
PJ.PV = VIE_MAX ;
```

le compilateur va passer sur notre code source et va remplacer (provisoirement)  

```
PJ.PV = 50 ;
```

Et ensuite faire son job.

C'est vachement utile pour modifier une valeur pour que cela se répète dans tous le code source.  
Souvent une valeur qui ne doit pas bouger en cours de programme, il est utile de faire des défines.

On peut aussi définir des fonctions.

```
#define EFFACER_ECRAN cls(0) ;
```

Quand le programme va trouver EFFACER\_ECRAN il va remplacer ça par `cls(0)`.

Attention, la déclaration des defines doivent se trouver hors fonctions, ou dans les headers ! Et avant d'utiliser les defines dans le programme !!!