Numéro 1
Avril 93

Passerelle Développeurs

EDITO

Le Falcon est enfin sorti et les ventes ont démarré en flèche dès les premiers jours, aidées par une couverture "presse" exceptionnelle (unanimité élogieuse par toutes les revues informatiques, y compris Décision et le Monde Informatique, généralement peu enclin à parler d'Atari, mais aussi dans la presse généraliste comme Libération, Le Figaro, etc.). Maintenant la balle est essentiellement dans votre camp...

Il faut rapidement au Falcon030 des logiciels originaux tirant pleinement parti de ses caractéristiques. Les développeurs français sont bien partis (comme en témoigne la présence d'Eurosoft, Silmarils, Brainstorm et Lankhor au CeBIT) avec quelques logiciels clés pour le Falcon030 (Photo Studio, AFM, D2M, Crazy Music Machine, Toki...) Il faut poursuivre cet effort. Nous attendons de votre part des logiciels d'animation, de vidéo, de création musicale exploitant le DSP, des éducatifs d'une nouvelle génération, des jeux spectaculaires et autres applications Multimédia. Pensez Grand Public! Même ceux d'entre vous dont les développements sont verticaux, se doivent d'envisager des applications plus grand publics de leur travaux. Mais avant tout, soyez originaux! Le marché informatique a besoin d'être ébranlé et le public veut être étonné. Le Falcon est là pour çà !

Daniel Hammaoui
Loïc Duval & Jean Marie Cocheteau

EN BREF

Bon, Okay, on sait! On a pris pas mal de retard sur un certain nombre de services développeurs, à commencer par cette lettre aux développeurs dont vous tenez entre les mains le premier exemplaire. On ne peut pas tout faire en même temps, et, je pense que nul ne nous tiendra rigueur si nous avons préféré donner la priorité à la sortie du Falcon030.

Le service télématique 3614 souffre lui aussi de retard dans sa mise en place. Cependant depuis la fin de Février, les développeurs ont accès à un **BBS** dont le numéro est le **(1).44.67.08.44.**
La formule BBS est à notre avis plus pratique (et pas nécessairement plus onéreuse) que le 3614.
En effet, télécharger 400Ko d'un nouveau TOS, avec un Minitel, relève du sado-masochisme.
Reste que ce projet de 3614 n'est pas jeté aux oubliettes. Nous attendons là aussi votre avis...

Et pour nous faire pardonner, puisque ce numéro 1 a pris du retard, vous aurez droit au numéro 2 dans trois semaines, histoire de rattraper le temps perdu...

ATARI FRANCE 79, Avenue Louis Roche – 92238 Gennevilliers Cedex Tél: (1) 40.85.31.00 Fax: (1) 40.86.29.78

CD PHOTO KODAK et FALCON030

par Loïc Duval

Tout le monde a entendu parler (du moins je l'espère) du CD-Photo Kodak qui connaît un succès croissant. Le transfert des photos sous forme numérique sur CD constitue une véritable révolution de l'univers photographique. Les applications grand public et professionnelles du Photo-CD sont nombreuses.

Toutefois l'ensemble de la technologie CD-Photo est rigoureusement copyrightée et surveillée par Kodak. Notamment il est impossible de développer un logiciel accédant aux CD-Photos sans autorisation de Kodak (qui dit autorisation dit "royalties" élevées)...

Atari et Kodak ont donc signé fin 1992 un accord de partenariat faisant de Kodak un développeur Atari, et d'Atari un développeur Kodak. Grâce à cet accord, vous pouvez bénéficier du kit de développement Photo-CD Atari. Sachez que KODAK interdit à tout constructeur, éditeur ou développeur l'utilisation des Photo-CD en dehors du kit de développement.
Ce qui signifie:

* Qu'Atari ne pouvait commercialiser de produits Photo-CD sans avoir signé l'accord pré-cité.
* Qu'aucun développeur Falcon030 n'a le droit de commercialiser une application exploitant le Photo-CD sans avoir préalablement acquis le kit de développement Atari/Kodak.
* Qu'aucun logiciel exploitant le Photo-CD ne peut être commercialisé sans l'approbation préalable d'Atari France (conformément aux accords Kodak Corp/Atari Corp).

Plus concrètement, si vous désirez développer un programme exploitant les Photo-CD, il vous faudra acquérir le kit de développement Atari/Kodak auprès d'Atari France et payer pour chaque exemplaire fabriqué de votre logiciel des royalties à Kodak (via Atari France). Pour plus d'informations sur le kit et le montant des royalties, veuillez contacter Loïc Duval chez Atari France au 40.85.31.54.

Atari commercialisera courant Mai un package "Photo-CD" permettant aux utilisateurs d'accéder aux Photo-CD, d'afficher les images du PCD et de les sauver dans un autre format sur disque dur, de rejouer les "PCD-Portfolios" de Kodak comme "From Alice to Ocean".

En effet le kit Atari devrait offrir un accès complet à tous les types de CD-Photo:

– **Le Photo-CD grand public** (une centaine de photo sur 1 CD)
– **Le Photo-CD MasterPRO** (pour films 135, 120 et 4x5 inches)
– **Le Photo-CD Medical** (dédié à l'imagerie médicale très haute définition)
– **Le Photo-CD Portfolio** (800 photos à accès programmé avec textes et sons)
– **Le Photo-CD Catalog** (intègre sous forme de pages photo/texte/son et peut contenir jusqu'à 6000 images).

Les objets 3D

par Brainstorm

D'abord sous MultiTos puis sous le TOS 4.01 est apparue une nouvelle race d'objets: les objets 3D (tridimensionnels).

Il s'agit en réalité d'objets **AES** classiques (BUTTON, BOXTEXT, ...) dessinés en simili-relief grâce à l'utilisation de teintes de gris. Il est d'ailleurs depuis devenu interdit de changer les 16 premières couleurs de la palette. Les fenêtres, les boîtes d'alerte, le sélecteur de fichiers, les boîtes du bureau et même les CPX systèmes sont devenus **3D.**

Il est donc devenu obligatoire d'être en look 3D dorénavant.

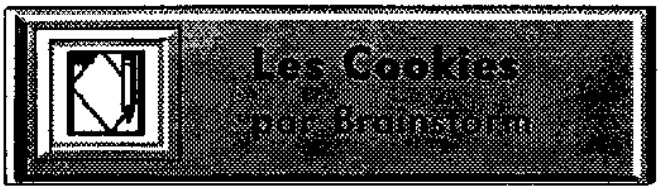
Sous réserve donc que les 16 premières couleurs de la palette ne changent pas, créer un **objet 3D** demande peu d'effort au programmeur. Sous un éditeur de ressource classique (par exemple Interface ou K-Ressource) il suffit de mettre le **bit 9 d'ob_flag à 1** dans le cas d'un **"indicator"** ou les **bits 9 et 10** dans le cas d'un **"activator"**.
Un **"indicator"**, comme son nom le laisse sous-entendre, est en fait réservé aux objets indiquant un état ou un choix. C'est le cas des objets radio-button ou des boutons ON/OFF par exemple. Un **"activator"** désigne un ascenseur, un curseur, etc...

La première implémentation de la **3D** se faisait en mettant une valeur particulière de l'**"Extended Object Type"** (type étendu d'objet, c'est à dire l'**octet haut d'ob_type**). La valeur **0** était réservée à l'**indicator**, et la valeur **1** à l'**activator**. De nombreux problèmes ont alors surgis, car cet octet était déjà utilisé par certains logiciels et certaines librairies (par ex NeXT GEM) pour gérer de nouveaux types d'objets. De plus, le numéro des sous-menus récemment introduits était aussi stocké à cet endroit, ce qui rendait la chose assez perturbante...
Cela a donc été changé.

Au passage, un mode supplémentaire méritait d'être créé. Les objets inactivables (fonds de boîtes, champs d'édition, ...) devaient pouvoir se fondre dans le décor **3D**. Un bit **"Background"** fut donc ajouté (le **bit 10**), permettant tout simplement à l'objet d'hériter de la couleur de fond de la **3D**.

Ainsi, dorénavant, les significations des combinaisons de bits sont les suivantes:

bit 9 à 1, 10 à 0: Indicator.
bit 9 à 1, 10 à 1: Activator.
bit 9 à 0, 10 à 1: Background.
bit 11 à 1: ext_ob_type contient un numéro de sous-menu.



Les "cookies" ("gâteaux") servent à pouvoir laisser des informations en mémoire mais de manière dynamique.

Le système ainsi que certains TSR mettent en place des cookies, afin que des programmes "classiques" reconnaissent leur présence et utilisent les informations qu'ils donnent.
La **"cookie jar"** (boîte à gâteaux) contient l'ensemble des cookies. Il s'agit en réalité d'un tableau de structures cookie dont le pointeur se situe en $5A0. Cette adresse étant en mémoire superviseur, il convient de prendre son contenu en étant en mode superviseur, par un **Supexec()** ou après un **Super().** Le tableau lui-même se situe en mémoire utilisateur. Sur les TOS antérieurs au TOS 1.6, **$5A0** contient 0.

Chaque cookie est composé de 2 longs mots:

```
typedef struct {
  long magic; /* identifie le cookie */
  longdata;   /* valeur ou pointeur spécifique à ce cookie */}
COOKIE;
```

"magic" est une séquence ASCII de 4 caractères, en théorie unique. Les magics commençant par _(underscore) sont réservés au système.
"data" est variable suivant les cookies. Il s'agit normalement d'un long mot constituant une information en soi, ou d'un pointeur sur une structure quelque part en mémoire, généralement allouée ou située dans le **TSR** l'ayant mise en place.

Les cookies du système sont:

_CPU: data est un mot long pouvant être:
- 0: 68000 (ST, STe, Mega STe)
- 10: 68010 (rare)
- 20: 68020 (aussi rare)
- 30: 68030 (TT, Falcon)
- 40: 68040 (?)

_VDO: data contient deux mots, celui de poids fort étant le numéro principal et celui de poids faible le sous-numéro:
- 0,0: ST
- 1,0: STe (scrolling hardware)
- 1,1: STBook
- 2,0: TT
- 3,0: Falcon

_SND: les premiers bits de data ont la signification suivante:
- 0: PSG (Yamaha) (Tous modèles).
- 1: 8-bit DMA (STe, TT, Falcon)
- 2: 16-bit codec (Falcon)
- 3: DSP (Falcon)
- 4: Matrix (Falcon)

_MCH: donne le type de machine. Comme ci-dessus, le long mot contient deux mots (poids fort, poids faible):
- 0,0: ST (en fait TOS 2.06).
- 1,0: STe (et Mega STe), donc TOS 1.06, 1.62, 2.05.
- 1,1: STBook.
- 1,$100: Falcon rev 1, rev 2.
- 2,0: TT, donc TOS 3.06.
- 3,0: Falcon à partir de la rev 3, donc TOS 4.00 à 4.04.

_SWI: donne la configuration des "switches" (interrupteurs) présents suivant le type de machine.
- $00: ST.
- $BF: Falcon.
- $FF: TT.
- $xx: autres.

_FRB: signifie Fast Ram Buffer. data est l'adresse d'un tampon de 64K servant au driver ACSI/DMA. N'est présent que sur TT, seule machine possédant 2 types de RAM.

_FDC: indique la présence d'AJAX et les capacités de gestion des drives Haute Densité. L'octet de poids le plus fort prend les valeurs suivantes:
- 00 Double ou simple densité -720/360 K
- 01 Haute densité 1.44 Mo (HD)
- 02 Extra Haute Densité 2.88 Mo (ED).

Si le **_FDC** est présent avec les valeurs pour HD ou ED vous pouvez être certain que le système est configuré pour la gestion de ces mécanismes et qu'il est donc possible d'effectuer un **Flopfmt()** en 18 secteurs ou un **Protobt** avec l'argument 4 (pour HD).
Les 3 octets les plus faibles identifient le système HD utilisé:
- $000000 Pas d'information spécifiée sur l'origine du système
- $415443 "ATC" signifie que AJAX est installé
- $445031 "DP1" signifie que l'update DreamPark est installé sur ce ST.

_FPU: data contient deux mots dont le plus fort signifie:
bit 0: 68881 par entrée-sortie (ex: SFP004).
bits 1 à 3:
- 0: rien.
- 1: 68881 ou 68882.
- 2: 68881.
- 3: 68882.
- 4: 68040.

_IDT:
- $FF: Falcon. (Cf doc développeur sur ce Cookie)

_PWR: sur ST-BOOK, data est le pointeur sur la structure de paramètres du "Power Management" (gestion de la consommation).

_NET: n'est pas à proprement parler un cookie du système. Il s'agit du cookie à installer par tout réseau. Sa donnée est un pointeur vers une structure commençant obligatoirement par son numéro d'identification (en long mot) donné par Atari suivi par son numéro de version (toujours en long).

MiNT: ne commence pas par _, mais est maintenant un cookie du système. Il indique la présence de MiNT, et donc de MultiTOS.

DEUX NOUVEAUX COOKIES VONT FAIRE LEUR APPARITION DANS DEUX SEMAINES. Ils seront détaillés dans le prochain numéro. Ces deux cookies sont:
_AFM: Pour l'Audio Fun Machine
_JPD: Pour signaler la présence du décodeur JPEG.

Utilisation du SSI

par Vincent Habchi

Le SSI (Synchronous Serial Interface) est le canal série qui permet au DSP de communiquer avec le monde extérieur. Dans l'absolu cette liaison série est très complète, et donc assez difficile à programmer correctement. Dans le cas qui nous intéresse ici, nous ne verrons que la programmation nécessaire pour communiquer avec le CODEC interne, de manière à réaliser des effets.

Le SSI se programme par les registres suivants :

CRA, situé à l'adresse : X:$FFEC
CRB, situé à l'adresse : X:$FFED
SSR, situé à l'adresse : X:$FFEE
SRX et **STX** à l'adresse : X:$FFEF

Pour pouvoir utiliser le SSI en tant que tel, il est tout d'abord nécessaire de programmer le registre PCC (X:$FFE1) à la valeur $01F8. Ceci aiguille les entrées/sorties du port C du DSP vers le SSI.

Lorsqu'il reçoit des échantillons du CODEC, le SSI fonctionne en mode réseau, c'est à dire que chaque échantillon accepté est accompagné par un signal qui le délimite dans le temps. Les échantillons sont groupés par paires puis envoyés dans des paquets au DSP.

Ceci nous conduit au mode de programmation suivant des registres du DSP:#
CRA : $4100
CRB : $F800

Le CRB mis à $F800 autorise les interruptions en réception et en émission, et met en marche les transmetteurs et les récepteurs série. Il sélectionne le mode réseau, avec une horloge fonctionnant en continu, un protocole asynchrone (c'est à dire que les échantillons sont reçus et émis à des instants différents), une horloge RX correspondant aux limites des paquets d'échantillon, des horloges RX et TX semblables, une direction de décalage normale (droite à gauche) et une horloge externe (Cf la documentation DSP pour plus de précisions).

Pareillement, le CRA à $FFEC sélectionne des échantillons de 16 bits, une horloge non divisée (utilisée telle quelle) et deux échantillons par paquets.

L'écriture de ces trois registres suffit à configurer correctement le SSI pour le faire fonctionner avec le CODEC interne. Théoriquement, si un équipement externe était relié au port DSP, il suffirait seulement de rajouter un certain nombre d'échantillons à la taille de paquet (Par exemple, si on rajoute un deuxième CODEC, il faudrait faire passer la taille de chaque paquet de 2 à 4 échantillons).

Différences entre les TOS 4.0x

par Brainstorm

Changements intervenus entre le TOS 4.01 et le TOS 4.04:

Sur le TT a été ajoutée une zone de RAM non volatile, c'est à dire ne se vidant pas à l'arrêt de la machine. Elle sert entre autre à spécifier le pays de la machine et l'unité disque de démarrage. Si par extraordinaire cette zone appellée **NVRAM** était écrasée, il existe sur **TOS 4.03** un moyen de remettre une configuration standard (pays:USA et unité de boot:18, cad IDE). Il suffit d'appuyer pendant la vérification de la mémoire intervenant juste après un redémarrage à froid sur les touches **Control – Alternate – Undo**.

Avec les **TOS 4.00 à 4.02** pour se sortir des plantages entraînés par la destruction de la NVRAM, il suffit de connecter un **moniteur monochrome et de relancer le programme CONFIG** présent sur les disquettes du kit.

Le nouvel appel du driver video Vsetmode() faisait une partie du travail qu'aurait dû faire Setscreen(). A partir du TOS 4.04, Vsetmode ne change plus que les registres video hardware.

Pour information, à l'heure actuelle (15 Avril) les Falcon030 sortant des chaînes de production sont en TOS 4.02.

Dialogue

par Atari France

– Pensez à nous envoyer les versions (Bêtas & finales) de vos logiciels. Nous en avons besoin pour les présentations à la presse ou à l'internationale. De plus, cela peut vous être profitable; sachez qu'en 6 mois, nous avons effectué près de 120 démonstrations des produits ! Et n'ayez crainte, ces versions ne sortent pas de chez nous... (Très tôt dans vos développements, pensez aux versions de démos)

– Prévenez-nous de tout changement dans vos coordonnées, des dates de sorties de vos produits, du nom de votre éditeur etc...

– Si vous êtes à la recherche d'un éditeur ou d'un distributeur pour vos produits, nous pouvons intervenir.

– Les colonnes de ce journal vous sont ouvertes, alors profitez-en pour nous faire parvenir articles, suggestions et remarques.

Doc Développeur:
Mises à jour des corrections
par Alan Corp.

On page HARDWARE.7
I have uploaded a diagram of the internal expansion port to the library. It is intended to replace the diagram.

Falcon030 Memory Connector

J6. 30 pin, dual row, upright male header

Pin#	Signal	Pin#	Signal
1	GND	2	GND
3	VCC	4	MADDR 8
5	MADDR 7	6	MADDR 6
7	MADDR 5	8	MADDR 4
9	MADDR 3	10	MADDR 2
11	MADDR 1	12	MADDR 0
13	GND	14	VCC
15	VCC	16	GND
17	GND	18	VCC
19	MADDR 10	20	MADDR 9
21	WE	22	RAS 0
23	RAS 1	24	CAS0 H
25	CAS0 L	26	CAS1 H
27	CAS1 L	28	GND
29	VCC	30	VCC

J17. 50 pin, dual row, upright male header

Pin#	Signal	Pin#	Signal
1	GND	2	VCC
3	MDATA 15	4	MDATA 14
5	MDATA 13	6	MDATA 12
7	MDATA 11	8	MDATA 10
9	GND	10	VCC
11	MDATA 9	12	MDATA 8
13	MDATA 7	14	MDATA 6
15	MDATA 5	16	GND
17	VCC	18	MDATA 4
19	MDATA 3	20	MDATA 2
21	MDATA 1	22	MDATA 0
23	DRAM 0	24	GND
25	VCC	26	GND
27	VCC	28	MDATA 16
29	MDATA 17	30	MDATA 18
31	MDATA 19	32	MDATA 20
33	MDATA 21	34	GND
35	VCC	36	MDATA 22
37	MDATA 23	38	MDATA 24
39	MDATA 25	40	MDATA 26
41	GND	42	VCC
43	MDATA 27	44	MDATA 28
45	MDATA 29	46	MDATA 30
47	MDATA 31	48	DRAM 1
49	GND	50	VCC

DRAM 0	DRAM 1		
0	0	1MB	(256k x 4)
1	0	4MB	(1M x 4)
0	1	14MB	(4M x 1)
1	1	Reserved	

On page HARDWARE.29 at the top...
It should read: *In Continuous mode there are 128 clock cycles per sample period. XO_SYNC will go high for the first 16 bits of a sample period and then low for the remaining 112 bits.*
(This replaces the text which states that the number of remaining bits is 96. That number was incorrect.)

On page VIDEO.2
The 1st paragraph of the description of OPCODE 5 (WORD SetScreen) reads as follows:
"If you pass a 3 in the rez word and a modecode in the mode word, SetScreen will set that mode and relloc the screen RAM to match that mode."
This should be changed to:
If you pass a 3 in the rez word and a modecode in the mode word, SetScreen will set that mode.
If you pass values of 0 for log and phys, then SetScreen will realloc screen RAM to match that mode.
Otherwise it will set them to the passed values or do nothing if passed minus 1 (-1).

On page VIDEO.6
The description of the VsetMask() call should be changed as follows:

OPCODE 150
VsetMask(ormask, andmask, overlay)
LONG ormask, andmask;
WORD overlay;

The VsetMask() function is used to set the mask values used by VDI to modify the color values computed for vs_color().
The vs_color() function converts its input to a 16-bit RGB value which is bitwise OR'ed with 'ormask' and then bitwise AND'ed with 'andmask'. This allows the application to set any color to be transparent (or not) in the 15-bit per pixel true color modes with genlock and overlay.

The default mask values are:
'ormask' = 0x0L, 'andmask' =(LONG)0xFFFFFFFF.
This combination of mask values has no effect.
To set the overlay bit, use:
'ormask' = 0x0020L, 'andmask' =(LONG)0xFFFFFFFF.
Now any color set with vs_color() will have the overlay bit set.
To clear the overlay bit, use:
'ormask' = 0x0L, 'andmask' = 0xFFFFFFDF.
Now any color set with:
vs_color() will have the overlay bit cleared.
If the 'overlay' parameter is non-zero, then the system will be put into overlay mode.
If the 'overlay' parameter is zero, then the system will be taken out of overlay mode.

On page E-4 of the "Falcon Owner's Manual"
The pinout diagram is numbered incorrectly. It shows the pin numbers wrapping around as a chip's pin numbers do. It should start with pin 1 being on the top left and pin 10 designated on the top right (the opposite of what it shows now) The bottom row is correct.

La Gestion du GEM en True Color

par Mike Fulton

In palette-based video modes, a pixel value represents a hardware color register. Each hardware color register contains an RGB color value which determines the actual color used to display the pixel. If you change the RGB values contained in a color register, all pixels in the display which contain the value corresponding to that register will change color. The built-in video of the ST/STe/TT series is palette-based.

With true color video, the value of a pixel directly represents the actual RGB value that will be displayed. Each pixel's color is set independently of all other pixels and changing its color is done by changing the pixel value itself. The Atari Falcon030 features both palette-based and true color video capabilities.

Detecting True Color Video

If you request the extended inquire values from the *vq_extnd()* function for a screen workstation, the value returned in *intout[5]* specifies if a color lookup table is supported. (See the documentation correction in this newsletter.) If it's zero, it means the device does not support a color lookup table, and that the pixel values themselves directly represent the displayed color. This means you are either in a true color video mode, or that you are in a monochrome mode. To determine which, check the number of bitplanes (*intout[4]* from *vq_extnd()*), if greater than one, then you're in a true color video mode, and if it's equal to one, then you're in a monochrome video mode.

What's Different?

From a programmer's viewpoint, for the most part, GEM VDI works with true color video modes in the same way that it does in palette-based video modes. However, there are some inevitable differences which cannot be avoided. Some GEM VDI functions take different parameters or behave a bit differently in true color video modes. The information below details these differences, and should be used in conjunction with your GEM VDI manual.

The *intout[13]* value from *v_opnwk()* or *v_opnvwk()* returns the number of pre-defined colors, or pens, that are available. For palette-based video, this indicates how many colors can be shown at once (without resorting to doing things like using interrupts to change the color registers). If you change the RGB value of a pen using *vs_color()*, then any pixels on screen which were drawn with that pen will change to the new color.

For true color video modes, *intout[13]* indicates how many pre-defined pens there are, but not how many colors can be displayed on screen at once. The pens contain RGB values that will be used to draw whenever you use VDI functions like *v_pline()* or *v_bar()*, but they don't necessarily have a direct relationship to what's already on screen like they do in palette-based video modes.

In other words, changing a pen's color with *vs_color()* does not affect anything you've already drawn with that pen. For example, if you draw a circle with pen 12, then change pen 12's RGB value with *vs_color()*, and then draw a square with pen 12, you will end up with the circle in pen 12's original RGB color and the square in pen 12's new RGB color. One result of this behavior is that you cannot do color cycling by simply changing the pen's color values with *vs_color()*.

vro_cpyfm()

```
vro_cpyfm( handle, wr_mode, pxyarray, psrcMFDB, pdesMFDB );
WORD handle;
WORD wr_mode;
WORD pxyarray[8];
MFDB *psrcMFDB;
MFDB *pdesMFDB;
```

The *wr_mode* parameter indicates the write mode to use in copying a rectangular area from one raster form to another. In both true color and palette-based modes, this is a bit-level operation. Since the bits for a pixel value mean something different in true color modes, the onscreen results of the same logic operation can be different. They are easily predictable, however.

Some programs use *vro_cpyfm()* with write mode 0 to clear areas of the screen or to clear an off-screen buffer. This sets all bits in the destination rectangle to zero, regardless of the original value. In palette-based modes, this has the effect of causing the pixels within that rectangle to be displayed using Pen 0, which is the background color, and which can be any RGB value. However, in true color video modes, setting all the bits to zero causes the pixels to be displayed as black, no matter what RGB value pen 0 is set to, since the values for red, green, and blue are all zero. In cases where the background color isn't supposed to be black, say for example in an animation playback, the results aren't going to be what was intended.

So how do you get around this? You can use *vrt_cpyfm()* and a dummy raster form instead of *vro_cpyfm()*.

The *vrt_cpyfm()* function blits from a single-plane (1 bit per pixel) raster to a multi-plane raster. You specify two VDI pens to be used for drawing pixels in the destination raster: one for the pixels with 1's in the source raster, and one for the pixels with 0's. If you specify the same pen for both, then the whole rectangle will be drawn with the same pen, and it doesn't even really matter what the source MFDB points to (the easiest thing to do is point it at the same memory as the destination raster, but set the MFDB to one plane). If you wanted to set a rectangular area of the raster to the background color, then you could just use *vrt_cpyfm()* with zero for both pens. Unlike other VDI functions that use pens, like *v_bar()* or *vr_recfl()*, the *vrt_cpyfm()* function can work with offscreen buffers with no significant extra effort.

Another thing to watch out for is the way other logic operations react. With palette based modes, if one did an OR blit operation with a source raster containing pixel values of 1 and a destination raster containing pixel values of 4, you'd end up with the destination rectangle containing pixel values of 5 (1 OR 4 = 5) which could be the same RGB color as either the pixel value 1 or pixel value 4, or even something completely different and apparently unrelated as far as RGB colors are concerned. In true color modes, if you did an OR blit operation with a source raster containing red pixels and a destination raster containing blue pixels, you'd end up with the destination raster containing magenta (red + blue = magenta) pixels. (And if you think about it a bit, you'll see that this sort of thing could be very useful.)

What *vro_cpyfm()* does to the raster memory hasn't really changed; the bits are affected in the same way as before, but now the results may not always mean the same thing they used to. While the replace write mode hasn't changed, other write modes with *vro_cpyfm()* may also give different results from palette-based modes. Some experimentation may be required to obtain the desired results.

vswr_mode()
vrt_cpyfm()

```
vswr_mode( handle, mode )
WORD handle;
WORD mode;

vrt_cpyfm( handle, wr_mode, pxyarray, psrcMFDB, pdesMFDB,
        color_index )
WORD handle;
WORD wr_mode;
WORD pxyarray[8];
MFDB *psrcMFDB;
MFDB *pdesMFDB;
WORD color_index[2];
```

The *vswr_mode()* function sets the write mode logic that GEM VDI will use in drawing pixels to the screen for all calls except *vro_cpyfm()* and *vrt_cpyfm()*, both of which use their own write mode parameters. Since the write modes for *vrt_cpyfm()* are the same as for *vswr_mode()*, the information below refers to both *vswr_mode()* and *vrt_cpyfm()*.

```
mode =  0 Replace mode
        1 Transparent mode
        2 eXclusive OR (XOR) mode
        3 Reverse Transparent
```

As with *vro_cpyfm()*, the differences with *vswr_mode()* arise from the fact that the logic of the write mode works at the bit level.

In palette-based modes, the write modes work as follows: Replace mode simply copies bits from the new pixel value into the destination pixel, completely replacing the old value with the new value. XOR mode does an exclusive-OR logic operation between the new pixel value and the destination pixel's existing value, and sets the pixel to the resulting value. Transparent mode affects pixels only when they are not being drawn with pen zero. Reverse-Transparent mode only affects pixels that are supposed to be drawn using pen zero, except it uses the drawing color to draw those pixels instead.

Figure #1 shows an example of each type of write mode. The circle was drawn first using replace mode, then the box was drawn on top, using a different write mode each time.

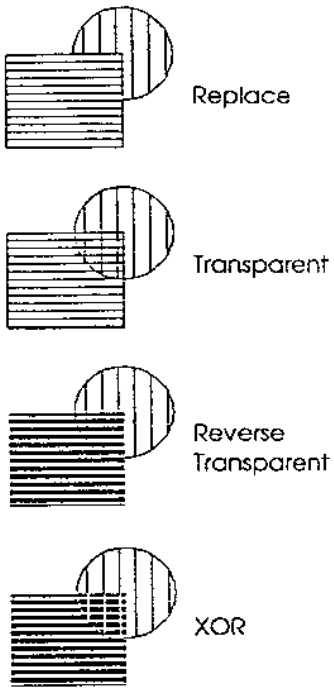


Figure #1

If the fill color is set to pen four, then in replace mode, the box outline and hatch pattern will be drawn using pen four, and the rest of the inside of the box will be drawn with pen zero so that whatever was previously underneath will be completely replaced. In transparent mode the rest of the inside of the box won't be drawn at all, and whatever was previously underneath will still be there. In reverse transparent mode, the outline and hatch pattern won't be drawn at all, but the rest of the inside of the box will be drawn, using pen four (the fill color) instead of pen zero. In XOR mode, the box outline and hatch pattern will be drawn in whatever is the result of pen 4 XOR the existing pixel values, and the rest of the inside of the box will be drawn in the result of pen 0 XOR the existing pixel values.

Using XOR mode can be a little different in true color video modes. A thing sometimes done in palette-based modes is to write some object to the screen against the background color, then write it again using XOR mode to erase it. In true color modes there are two ways this can fail unless the background is pure-black.

If XOR mode is used to draw the object both times, then the it will be drawn in the wrong colors if the background isn't pure black, but will be erased as expected when drawn the second time. If XOR mode isn't used to draw the object in the first place, but replace mode is used instead, then the object will be drawn in the correct colors, but it will not be erased properly when drawn the second time. Instead of the background color reappearing, you'll get black instead because any number XOR'ed with itself is zero, and in true color, a pixel value of zero means black.

Another difference of XOR mode is that when you XOR a pixel in a palette-based mode, the result is a different pixel value, which will have its own RGB value that may not have any meaningful relationship to the original pixel's RGB value and the XOR operation. In true color mode, if you XOR a pixel, the resulting color is both easily predictable and meaningful. For example, if you XOR any RGB value with a pure white

RGB value (0xffff for 16-bit or 0x00ffffff for 24/32 bit) then you'll get the opposite color. For example, if you XOR a red pixel with 0xffff and you get a cyan (blue+green) pixel, which is the opposite color on a color wheel. If you XOR a yellow (red+green) pixel with 0xffff, you get blue.

v_get_pixel()

```
v_get_pixel( handle, x, y, pel, index )
WORD handle;
WORD x, y;
WORD *pel, *index;
```

In palette-based video modes, *pel* is the hardware-based value of the pixel and *index* is the GEM VDI pen value for the pixel. In true color video modes, the original GEM pen used to draw a pixel cannot be determined with any certainty, and the pixel value represents an RGB value, so the meanings of the values returned in *pel* and *index* change.

In 16-bit true color modes:

```
index = 0
pel = 16-bit RGB pixel value, where:

        Bit 15          Bit 0
pel:    (RRRR RGGG GGGB BBBB)

        Red = 0-31
        Green = 0-63
        Blue = 0-31
```

In 32-bit true color modes:

```
index = high word of 32-bit RGB pixel value
pel = low word of 32-bit RGB pixel value

        Bit 15           Bit 0
index:  (AAAA AAAA RRRR RRRR)
pel:    (GGGG GGGG BBBB BBBB)


        A = Alpha Channel (non-RGB information)
        R = Red (0-255)
        G = Green (0-255)
        B = Blue (0-255)
```

vsf_udpat()

```
vsf_udpat( handle, pfill_pat, planes )
WORD handle;
long *pfill_pat;
WORD planes;
```

In palette-based modes, *pfill_pat* is a pointer to a 16x16 raster form in device-specific format. The raster may be either monochrome or color. In true color modes, for color fill patterns, the *pfill_pat* parameter should be a pointer to 256 (16 rows of 16 pixels each) 32-bit values which contain 24 bits of RGB information for each pixel of the fill pattern (using the format shown below). The *planes* value should be set to 32. This is true even for true color video modes with less than 24 bits per pixel. (GEM VDI will translate the RGB information to the correct values automatically.)

```
RGB format for user defined fill pattern data:

Bit 32                                  Bit 0
(0000 0000 RRRR RRRR GGGG GGGG BBBB BBBB)
```

Single-plane user-defined fill patterns work the same way in true color modes as in palette-based modes.

v_contourfill()

```
v_contourfill( handle, x, y, index )
WORD handle;
WORD x, y;
WORD index;
```

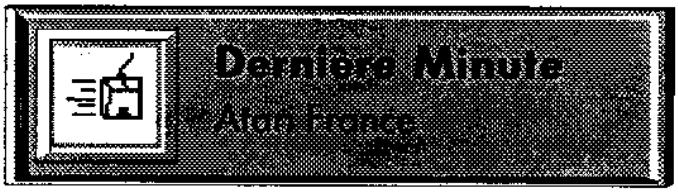
In true color video modes, the results of *v_contourfill()* may be different from palette-based video modes, although the function is still called in the same way.

In palette-based video modes, if the *index* parameter is a valid pen value, then the fill expands outward from (*x,y*) until it reaches pixels drawn with that pen. If *index* is negative, then the fill expands outward from (x,y) until it reaches pixels drawn with a different pen from the pixel at (*x,y*).

For example, let's say VDI pen 12 and VDI pen 13 have both been set to RGB values of [1000,0,0] (pure red) with *vs_color()*. If you have a big circle drawn with pen 12, and a smaller circle drawn with pen 13 inside that, and you fill at the center point of both circles with *index* equal to 12, then the fill will expand outside of the little circle until it reaches the big circle. It doesn't matter that both circle are shown in the same RGB color on-screen, because the VDI is able to determine that they have been drawn with different pens, because the values in the screen raster memory represent the pen values, not the RGB values.

In true color video modes, VDI will look up the current RGB values used for the pen indicated by *index* and the fill will stop when it reaches any pixel with the same RGB values. It doesn't matter what pen was used to draw the pixel; if a pixel's RGB values matches the current RGB value for the *index* pen, the fill will stop at that pixel. In the above example, that means the fill will stop when it reaches the smaller inside circle.

If *index* is negative in true color video modes, then the fill will expand outward from (*x,y*) until it finds pixels with a different RGB value from the pixel at (*x,y*). It doesn't matter if the pixels were drawn with different pens, if they have the same RGB value as the pixel at (*x,y*) they will be filled.



Voici les dernières nouveautés que nous avons reçues, et qui seront disponibles sur le BBS Support Développeur:

- **TOS** en version française 4.04
- **MultiTOS** en version française 1.01
- **AFM** version 1.01
- **SAM** version finale