

AMIGA NEWS-TECH

EN VENTE UNIQUEMENT PAR ABONNEMENT

ExecBase : Le C ANSI (III)	3	Denis Jarril
AMOS : AMOSTel (I)	4	François Lionet
ARexx : Les Bibliothèques	8	François Gueugnon
Pascal : Présentations (III)	10	Olivier Garrigues
GFA : 1 000 000 de Caractères à la Minute !	12	Pierre-Philippe Launay
Algos : Les Files (I)	14	Lucien Turski et Bruno Bailluet
Transactor : Le RGB Plasma	16	Stéphane Rubinstein
Algos : La Compression de Données (I)	18	Emmanuel Hocdet
Hardware : Les Interruptions (I)	20	Philippe Rivaillon
Hardware : 68030, Motorola nous gâte ! (I)	22	Emmanuel Hocdet
ToolBox : Yacc (II)	24	Herr Doktor von Truc
SubWay : C du Son (I)	26	Herr Doktor von Bidule
Demos : Effet de Zoom	29	Jérôme Etienne
Requester	32	Ok et Cancel



*Félix le Chat
s'ennuie tout seul.*

Amiga NewsTech est une publication de Commodore Revue SARL 5, rue de la Fidélité, 75010 PARIS.

EDITO

Vous l'aurez déjà certainement remarqué, depuis quelques temps, votre ANT mensuel n'arrive plus aux environs du 15, mais plus vers la fin du mois. C'est normal, il n'y a aucune raison de vous inquiéter : nous avons tout simplement décidé de ne pas rattraper le retard pris par le numéro 27.

Autre chose : les abonnés avec disquette recevront dorénavant leur magazine et la disquette dans deux envois séparés. Nous ne vous cachons pas que cela nous permet quelques économies substantielles (envoyer un gros paquet de journaux coûte en effet moins cher que deux petits). Mais d'un autre côté, cela supprimera également le décalage d'envoi entre les deux formules. Tout le monde y gagne.

Tiens, en parlant de gagner... Le formulaire de participation à Transactor a disparu de nos pages. Ce qui ne signifie pas que la rubrique n'existe plus, bien au contraire (allez faire un tour pages 16 et 17 pour vous en convaincre !). Rappelons qu'il y a 1 500 F à gagner pour tout article que nous publierons.

Et enfin, puisqu'on parle d'écrire, je vous rappelle que nous avons besoin de votre avis sur les nouvelles rubriques Langages pour décider de leur avenir. Alors, à vous de jouer !

Stéphane Schreiber

Abonnement à L'ANT et Anciens Numéros

Bon de commande à découper ou photocopier (pas de recopie manuelle SVP) et à retourner correctement rempli et accompagné de votre règlement à **MCM-ANT, 16, Quai Jean-Baptiste Clément, 94140 Alfortville, France**. Merci de libeller vos chèques à l'ordre de *Commodore Revue SARL*. Etranger : paiement par mandat international uniquement.

• *OUI, je m'abonne à Amiga NewsTech. Je choisis la formule :*

- ☐ 1 an (11 numéros) **SANS** disquette 350 FF (385 FF hors France)
- ☐ 1 an (11 numéros) **AVEC** disquette 600 FF (650 FF hors France)
- ☐ (cochez cette case pour un renouvellement d'abonnement)

• *Je commande les anciens numéros suivants :*

- | | | |
|--|--|--|
| ☐ n°20 (☐ avec disquette) | ☐ n°25 (☐ avec disquette) | ☐ n°30 (☐ avec disquette) |
| ☐ n°21 (☐ avec disquette) | ☐ n°26 (☐ avec disquette) | |
| ☐ n°22 (☐ avec disquette) | ☐ n°27 (☐ avec disquette) | |
| ☐ n°23 (☐ avec disquette) | ☐ n°28 (☐ avec disquette) | |
| ☐ n°24 (☐ avec disquette) | ☐ n°29 (☐ avec disquette) | |
| au tarif unitaire de 45 FF (60 FF avec disquette), pour un total de | | FF. |

Nom : Prénom :

Adresse :

Code Postal : Ville : Pays :

Suite de notre survol contemplatif de la bibliothèque standard du C version ANSI. Au menu, la fin des fonctions dédiées à la gestion des fichiers.

Rappelons rapidement que toutes les fonctions qui suivent sont déclarées dans le fichier d'en-tête *stdio.h*. Plusieurs types particuliers, généralement définis par un *typedef* bien placé et quelques fois par un *#define* de derrière les fagots, y figurent également. C'est notamment le cas de *FILE*, *size_t* et *fpos_t*, entrevus le mois dernier. D'autres types seront rencontrés, que nous expliciterons le moment venu.

```
int ferror(FILE *stream)
```

ferror() retourne une valeur non nulle si l'indicateur d'erreur est positionné pour le flux *stream*.

```
void perror(const char *s)
```

perror(s) imprime la chaîne *s* suivie d'un message d'erreur (dépendant du système) qui correspond à l'entier *errno*. C'est l'équivalent de *fprintf(stderr, "%s: %s\n", s, sys_errlist[errno]);*

```
int fgetc(FILE *stream)
```

fgetc() retourne le caractère suivant du flux *stream*, converti en *int*. *fgetc()* retourne EOF si la fin de fichier est atteinte.

```
char *fgets(char *s, int n, FILE *stream)
```

fgets() lit au plus *n-1* caractères dans le flux *stream* et les place dans la chaîne *s*, en s'arrêtant si elle rencontre un caractère de fin de ligne '\n'. La chaîne est ensuite terminée par le caractère nul '\0'. *fgets()* retourne *s* si tout s'est bien passé, et NULL en cas d'erreur.

```
int fputc(int c, FILE *stream)
```

fputc() écrit le caractère *c* dans le flux *stream*. Elle retourne le caractère écrit, ou EOF en cas d'erreur.

```
int fputs(const char *s, FILE *stream)
```

fputs() écrit la chaîne *s* dans le flux *stream*. Elle retourne EOF en cas d'erreur.

```
int getc(FILE *stream)
```

getc() équivaut à *fgetc()*, mis à part que c'est une macro (qui peut donc évaluer *stream* plusieurs fois).

```
int getchar(void)
```

getchar() est une macro équivalant à *getc(stdin)*.

```
char *gets(char *s)
```

gets() agit comme *fgets()*, si ce n'est qu'elle utilise automatiquement *stdin* et qu'aucune vérification de longueur de la chaîne en entrée n'est effectuée. Elle retourne *s* si tout va bien, ou NULL en cas d'erreur.

```
int putc(int c, FILE *stream)
```

putc() équivaut à *fputc()*, mis à part que c'est une macro.

```
int putchar(int c)
```

putchar() est une macro équivalant à *putc(c, stdout)*.

```
int puts(const char *s)
```

puts() écrit la chaîne *s* et le caractère de fin de ligne '\n' sur *stdout*. Elle retourne EOF en cas d'erreur.

```
int ungetc(int c, FILE *stream)
```

ungetc() remet *c* dans le flux *stream*, où il sera retrouvé à la prochaine lecture. Elle retourne le caractère remis, ou EOF en cas d'erreur.

```
int fprintf(FILE *stream, const char *format, ...)
```

fprint() convertit ses arguments (en nombre variable) d'après le format spécifié par la chaîne *format*, et écrit le résultat dans le flux *stream*. Elle retourne le nombre de caractères écrits, ou une valeur négative en cas d'erreur.

Note : la chaîne de formatage est décrite sous printf().

```
int fscanf(FILE *stream, const char *format, ...)
```

fscanf() lit les données depuis le flux *stream*, d'après le format défini par la chaîne *format*, et affecte les valeurs converties à chacun de ses arguments (en nombre variable), *chacun de ceux-ci devant être un pointeur*. Elle retourne le nombre d'objets convertis et affectés si tout va bien, et EOF en cas d'erreur.

Note : la chaîne de formatage est décrite sous scanf().

A suivre...

Les Listes Variables d'Arguments

La bibliothèque standard C met à la disposition du programmeur des fonctions dont le nombre et le type des arguments peut être variable. L'exemple le plus frappant en est printf() et ses consœurs.

Il n'y a bien entendu là-dedans aucun truc particulier (seulement une bonne gestion de la pile par le compilateur !), et le programmeur peut lui-même créer de telles fonctions. Bien que cela relève du fichier d'en-tête *stdarg.h*, il nous a paru opportun de faire un aparté sur le sujet dès maintenant.

On déclare une telle fonction en spécifiant trois points de suspension comme son dernier argument :

```
int fonction(char *arg1, int arg2, ...);
```

Dans cet exemple, notre fonction accepte au minimum deux arguments, appelés *arg1* et *arg2*, respectivement de type *char ** et *int*. D'autres peuvent suivre, de n'importe quel type. Pour implémenter cette fonction, on définit une liste d'arguments de type *va_list*, que l'on initialise (une seule fois !) au moyen de la macro *va_start*, à laquelle on fournit le nom du dernier argument obligatoire :

```
va_list va;
va_start(va, arg2);
```

Par la suite, chaque appel à la macro *va_arg* retournera un objet ayant le type et la valeur de l'argument suivant non nommé, et modifiera la liste *va* de telle sorte qu'elle pointe sur le prochain argument. Quand tous les arguments transmis auront été traités, il faut impérativement appeler la macro *va_end* avant de sortir de la fonction.

Pour illustrer tout ceci, la fonction suivante concatène toutes les chaînes qui lui sont transmises en une seule, dans le buffer *buf*. Elle retourne le nombre total de caractères placés dans *buf*.

```
int concat(char *buf, ...)
{
    va_list va;
    char *s;
    int i, j;

    va_start(va, buf);          /* Initialise la liste va */

    *buf = '\0';                /* "Vide" le buffer de réception */
    i = 0;                      /* Compteur de caractères à 0 */

    while (s = va_arg(va, char*)) /* Lit un "char *" */
    {
        if (j = strlen(s))
        {
            strcat(buf, s);      /* Ajoute la chaîne à buf */
            i += j;
        }
    }
    va_end(va);                 /* Remet la pile en état */
    return(i);                  /* Renvoie le nombre de caractères */
}
```


Chose promise, chose due : Daisy et moi nous sommes réconciliés ! Main dans la patte, nous allons vous proposer dans cette rubrique, de réaliser un émulateur Minitel complet en AMOS.

Il s'agit d'un travail de longue haleine, dont la publication sera étalée sur plusieurs articles. Nous n'allons pas trop nous étendre sur l'émulation elle-même, ce qui prendrait tout l'ANT pendant plusieurs mois, mais plutôt sur l'implémentation en AMOS. Ainsi aujourd'hui, vous trouverez un truc permettant de changer le jeu de caractères dans toute fenêtre ouverte. A propos de la documentation : le Minitel est un domaine assez professionnel, à cheval sur les réseaux de communication et la télématique. Les documentations que l'on trouve sont en général assez ardues. Nous avons cependant trouvé quelques livres assez abordables, comme *Guide Pratique du Vidéotex* aux éditions Eyrolles.

LES JEUX DE CARACTERES

Première chose à faire pour émuler le Minitel, reproduire ses jeux de caractères. Le Minitel ne contient pas moins de quatre jeux de cent vingt-huit caractères différents chacun.

- *Jeu G0* : les caractères ASCII - presque - standards. Les lettres de l'alphabet, les chiffres et ponctuations. Neuf caractères sont spécifiques, de \$5b à \$5e, et de \$7b à \$7f.
- *Jeu G1* : les mosaïques contigües. C'est grâce à ces caractères semi-graphiques que les serveurs peuvent afficher des images. Chaque caractère est divisé en six (deux par trois), et toutes les combinaisons sont explorées.
- *Jeu G2* : les mosaïques séparées, semblables aux précédentes avec un trait noir entre chaque pavé. Ces mosaïques sont souvent utilisées par les serveurs. L'annuaire électronique affiche le cadran de téléphone en haut à gauche de l'image avec les caractères du jeu G2.
- *Jeu G3* : les caractères spéciaux, à savoir les accents et trémas.

Comment implémenter ceci en AMOS ? Il est en effet impossible (théoriquement) de changer le jeu de caractères courant d'un écran. Impossible, jusqu'à maintenant ! La fonction `=SCREEN BASE` retourne l'adresse de base de la structure interne de définition d'un écran AMOS. En position 170 de cette structure, se trouve l'adresse de la structure de la fenêtre courante, dans laquelle l'adresse du jeu de caractères se trouve en position 8.

La petite procédure suivante fait pointer le jeu de caractères courant sur une adresse quelconque.

```
Procédure _SET_WINDOW_FONT[A]
  AFONT=Leek(Screen Base+170)+8
  Loke AFONT,A
End Proc
```

Pour aller plus rapidement lors de l'affichage, les caractères sont codés très simplement : l'un après l'autre, se trouvent les huit octets de définition de chaque caractère. Le jeu de caractères comprend les trente-deux premiers caractères, même si ceux-ci ne sont pas accessibles.

Quelques applications :

- Notre programme d'émulation, bien sûr. Nous allons fabriquer une banque mémoire contenant les quatre jeux à la suite.
- Définir de nouveaux symboles graphiques.
- Coder un listing, en faisant pointer les caractères sur n'importe quoi. Il n'y a aucun risque. Le nec plus ultra étant de faire pointer sur des données modifiées, par exemple `_SET_WINDOW_FONT[Log Base(0)]`.

```
*****
' Création des fontes minitel
' Daisy & François Lionet
' Paru dans l'ANT de Mars 1992
*****
Dim GR(6,8),C(8)
Global GR(),C()
'
' Lis les datas
Restore _GRAPH
For C=0 To 6
  For B=0 To 7
```

```
    Read A$ : GR(C,B)=Val(A$)
  Next
Next
'
' Ouvre un écran simple
Screen Open 0,320,200,2,0 : Curs Off : Palette 0,$FFF
'
' Reserve la banque de travail
Erase 15 : Reserve As Data 15,128*8*5
'
' Cree le jeu G0
Cls 0
For C=32 To 255 : _CPRINT[C,Chr$(C)] : Next
_CPRINT[$5F,Chr$(86)]
_CPRINT[$60,Chr$(89)]
_CPRINT[$7B,Chr$(83)]
_CPRINT[$7C,Chr$(8B)]
_CPRINT[$7D,Chr$(84)]
_CPRINT[$7E,Chr$(81)]
Inverse On : _CPRINT[$7F," "] : Inverse Off
_GET_FONT[Start(15)+128*8*0,256]
'
' Cree le jeu G1 mosaïques contigües
_CREE_GRAPH[0] : _GET_FONT[Start(15)+128*8*2,128]
'
' Cree le jeu G1 mosaïques séparées
_CREE_GRAPH[1] : _GET_FONT[Start(15)+128*8*3,128]
'
' Cree le jeu G2 spécial
Cls 0
For C=32 To 127 : _CPRINT[C,Chr$(5F)] : Next
_CPRINT[$23,"€"]
_CPRINT[$24,"$"]
_CPRINT[$26,"#"]
_CPRINT[$2C,"<"]
_CPRINT[$2D,"^"]
_CPRINT[$2E,">"]
_CPRINT[$2F,"|"]
_CPRINT[$30,"°"]
_CPRINT[$31,"+"]
_CPRINT[$38,"%"]
_CPRINT[$3C," "]
_CPRINT[$3D," "]
_CPRINT[$3E," "]
_CPRINT[$6A," "]
_CPRINT[$7A," "]
_GET_FONT[Start(15)+128*8*4,128]
'
' Sauve la banque
Save "Ram:Minitel_Fonte.Abk",15
Edit
'
Procedure _CREE_GRAPH[MASK]
'
' Cree le jeu graphique
Cls 0 : P=$20
For C=0 To 63
'
  For B=0 To 7 : C(B)=0 : Next
  For B=0 To 7
    If Bst(B,C)
      For BB=0 To 7
        C(BB)=C(BB) or GR(B,BB)
      Next
    End If
  Next
'
  ' Pour le jeu de mosaïques
  ' séparées
  If MASK
    For BB=0 To 7
      C(BB)=C(BB) and GR(6,BB)
    Next
  End If
'
  _PLOT[P]
'
  Inc P
  If P=$40 : P=$60 : End If
'
Next
'
' Encore un caractère
For C=0 To 7
  C(C)=-1 : If MASK : C(C)=C(C) and GR(6,C) : End If
Next : _PLOT[$5F]
'
End Proc
'
Procedure _PLOT[P]
'
' Poke le caractere dans l'écran
Y=P/16 : X=P mod 16
AEC=Logbase(0)+Y*8*40+X
For B=0 To 7
  Poke AEC+B*40,C(B)
Next
```


Quelques mots sur le fonctionnement de ce programme.

La procédure `_GET_FONT[]` se charge de transférer la fonte de l'écran dans la banque. Les jeux graphiques sont quant à eux fabriqués à partir des datas, en explorant toutes les différentes possibilités.

Vous devez vous procurer un cable Minitel-Amiga et relier les deux machines. Pour le moment, toutes les manipulations doivent se faire sur le Minitel.

Le programme est composé de deux procédures principales :
DISPLAY_INIT : initialise l'écran et lit toutes les données des codes de contrôle.

DISPLAY[A] : réceptionne un caractère, et émule l'écran du Minitel. Le caractère peut provenir de l'interface série, ou d'une banque mémoire dans laquelle vous aurez enregistré des données.

Ces deux procédures peuvent facilement être récupérées dans d'autres programmes. Vous devez pour cela :

- copier les deux procédures et la définition des variables globales ;
- ouvrir un écran 16 couleurs Lowres, d'au minimum 320x200 pixels ;
- appeler la procédure *DISPLAY INIT.*

Bien que l'Amiga soit infiniment supérieur au niveau graphique au Minitel, l'émulation de l'affichage est assez complexe à réaliser, du fait de la présence de codes de contrôles de couleur, appelés attributs graphiques.

Qu'est-ce donc qu'un attribut graphique ? Il s'agit d'un caractère affiché par le Minitel comme un espace, et définissant les caractéristiques de ce qui vient à sa droite sur l'écran. Peuvent ainsi être définies la couleur de fond et le soulignement.

Lorsque l'on affiche un caractère à l'écran, il faut donc pour connaître sa couleur de fond, explorer la ligne de la gauche vers la droite. Une émulation parfaite n'est donc possible qu'en langage-machine.

En AMOS, il faut ruser. Nous gardons en mémoire (dans la banque 11) une grille représentant l'écran. Cette grille contiendra les attributs graphiques. Nous avons aussi volontairement simplifié l'émulation. Tout fonctionne, sauf le soulignement, utilisé extrêmement finement par les serveurs, et par conséquent très difficile à reproduire, et le scrolling. Pour les initiés, le niveau de l'émulation se situe entre *Amigatel* et *FLammitel*.

AMOS permet d'adresser des labels contenus dans des chaînes de caractères. C'est extrêmement pratique et surtout, rapide. Notre émulateur doit en effet fonctionner assez rapidement et ne pas aller moins vite que le débit de la prise série !

Nous usons (et abusons) de cette facilité dans le programme :

- *CMODE\$* contient la fonction actuelle. La première instruction de la procédure *_DISPLAY* branche directement à la routine, sans perdre de temps en tests. Les routines de traitements spécifiques peuvent, par une simple égalité, modifier le branchement lors du *prochain* appel de la procédure.

- les labels des routines de gestion des codes de contrôle se trouvent dans les tableaux *CMODES()* et *ESC()*. Le numéro du code pointe directement dans le tableau. En un coup de cuillère à pot, on connaît la routine à appeler.

Nous utilisons une méthode toute simple pour afficher les caractères en double largeur (ou hauteur) : on affiche le caractère en taille normale, puis on le zoome à même l'écran.

Le mois prochain, Daisy et moi allons nous occuper du clavier. Le programme se comportera alors comme un véritable Minitel.

```

*****
' Emulateur minitel en AMOS
' Par Daisy & François Lionet
' Partie 1 : l'affichage
*****
' Tableaux des codes de controle
Dim CMODE$(32),ESC$(128)
' Variables globales
Global CMODE$,CMODE$(),ESC$(),OCMODE$
Global GMODE,FLSH,G2$,GOPAP,CPAP,CPEN,CINV,PRNT$,ALAST
Global XCU,YCU,OXCU,OYCU
Global PAR1,PAR2,PAR3
Global AATR
' Faire fonctionner Minitel_Fonte.AMOS
' puis ce programme. Vous pouvez ensuite
' enlever la ligne suivante...
'Load "ram:minitel_fonte.abk",15
' Ouverture de l'écran d'exemple
Screen Open 0,320,200,16,0 : Flash Off : Cls 0
_DISPLAY_INIT
' Ouvre l'interface série
Serial Open 0,1
' Boucle d'exemple
Repeat
' _SERIAL_GET
A=Param : If A>=0 : _DISPLAY[A,1] : End If
Until A<0
' Fermeture de la série
Serial Close
Procedure _DISPLAY_INIT
' Préparation de l'écran
Paper 0 : Cls
Wind Open 1,0,8,40,24
' La palette du minitel ne comprend que des couleurs
' extrêmement vives!
Palette 0,$F00,$F0,$FF0,$F,$F0F,$FF,$FFF
' Fait flasher les couleurs 8-15
For C=0 To 7
V=Colour(C)
Flash C+8,"("+Mid$(Hex$(V,3),2)+",20)(000,20)"
Next
' Initialisation des fenetres
Window 1 : Paper 0 : Pen 7 : Clw
Scroll Off : Curs Off
Window 0 : Paper 0 : Pen 7 : Clw
Scroll Off : Curs On : _SET_FONT[0]
' Initialisation des attributs
CMODE$="G0"
GMODE=0 : FLSH=0 : PRNT$="_NNPRINT"
GOPAP=-1 : CPAP=0 : CPEN=7 : CINV=0
' Lis les codes de controle
For C=0 To 31 : Read CMODE$(C) : Next
' Lis les code d'échappement
Repeat : Read C : Read ESC$(C) : Until ESC$(C)=""
' Chaine de définition des caracteres spéciaux
G2$=Chr$(41)+"eèaàù"+Chr$(42)
G2$=G2$+"eé"+Chr$(43)+"eêaâiîuôo"
G2$=G2$+Chr$(48)+"eëiîä"
G2$=G2$+Chr$(4B)+"ç"
' Initialise la carte de couleurs
Erase 11 : Reserve As Work 11,40*26 : AATR=Start(11)
Fill AATR To AATR+25*40,0
' Jeu de commandes du mode C0
Data "_Nul","_Nul","_Nul","_Nul"
Data "_Nul","_Nul","_Nul","_Nul"
Data "_ARR","_AVT","_BAS","_HAU"
Data "_CLS","_DLI","_GG1","_GG0"
Data "_Nul","_CON","_RPT","_CLA"
Data "_COF","_Nul","_GG2","_Nul"
Data "_CAN","_GG2","_SKP","_ESC"
Data "_Nul","_Nul","_HOM","_LOC"

```

```

' Jeu de commandes ESCAPE
Data $39,"_dld",$3A,"_lkk",$3B,"_dlk"
Data $40,"_PEN",$41,"_PEN",$42,"_PEN",$43,"_PEN"
Data $44,"_PEN",$45,"_PEN",$46,"_PEN",$47,"_PEN"
Data $48,"_GFL",$49,"_NFL",$4A,"_Nul",$4B,"_Nul"
Data $4C,"_GNN",$4D,"_GDY",$4E,"_GDX",$4F,"_GDD"
Data $50,"_PAP",$51,"_PAP",$52,"_PAP",$53,"_PAP"
Data $54,"_PAP",$55,"_PAP",$56,"_PAP",$57,"_PAP"
Data $58,"_Nul",$59,"_G1N",$5A,"_G1L",$5B,"_CSI"
Data $5C,"_NOR",$5D,"_INV",$5E,"_Nul",$5F,"_Nul"
Data 0,""
End Proc
'
Procedure _DISPLAY[A,F]
'
Goto CMODE$
'
' Entree impression normale
_G0:
If A<32 Then OCMODE$=CMODE$ : Goto CMODE$(A)
ALAST=A : Gosub PRNT$ : Pop Proc
'
' Impression normale
_NNPRINT:
If GMODE=0
If A<>$20
Gosub _ATTRIB
Print At(XCU,YCU)+Chr$(A);
Gosub _CRIGHT
Else
Gosub _ATTRIB : Gosub _LATCH : Print At(XCU,YCU)+" ";
Gosub _DELIM : Gosub _CRIGHT
End If
Else
Gosub _DELIM
Print At(XCU,YCU)+Chr$(A);
Gosub _CRIGHT
End If
Return
'
' Impression double largeur
_DXPRINT:
If XCU>38 Then Goto _NNPRINT
Gosub _CRIGHT : Gosub _NNPRINT
X=X Graphic(XCU-1) : Y=Y Graphic(YCU)
Zoom 0,X,Y,X+8,Y+8 To 0,X-8,Y,X+8,Y+8
Return
'
' Impression double hauteur
_DYPRINT:
If YCU<2 Then Goto _NNPRINT
X=X Graphic(XCU) : Y=Y Graphic(YCU)
Gosub _NNPRINT
Zoom 0,X,Y,X+8,Y+8 To 0,X,Y-8,X+8,Y+8
If XCU=0 : Gosub _CDOWN : End If
Return
'
' Impression double
_DDPRINT:
If XCU>38 Then Goto _DXPRINT
If YCU<1 Then Goto _DYPRINT
X=X Graphic(XCU+1) : Y=Y Graphic(YCU)
Gosub _CRIGHT : Gosub _NNPRINT
Zoom 0,X,Y,X+8,Y+8 To 0,X-8,Y-8,X+8,Y+8
If XCU<=1 : Gosub _CDOWN : End If
Return
'
' Trouve la couleur de papier
_ATTRIB:
AA=Peek(AATR+YCU*40+XCU) : P=AA and $7
If P<>CPAP : Paper P : CPAP=P : Gosub _SINV : End If
Return
'
' Met un délimiteur dans les couleurs
_DELIM:
Gosub _LATCH : AA=CPAP+$80
If XCU<39
For XA=XCU To 39
O=Peek(AATR+YCU*40+XA) : Poke AATR+YCU*40+XA,AA
Exit If AA=0 or Btst(7,0)
Bclr 7,AA
Next
End If
Return
'
' Faut-il changer la couleur de papier
_LATCH:
If GOPAP>=0
CPAP=GOPAP : GOPAP=-1 : Paper CPAP : Gosub _SINV
End If
Return
'
' Passe en inverse s'il faut
_SINV: If CINV : Inverse On Else Inverse Off : End If
Return
'
' Vers la droite

```



```

_CRIGHT: Inc XCU
If XCU>=40
  XCU=0 : If YCU>0 : Gosub _CDOWN : End If
End If
Locate XCU,YCU
Return
'
' Vers la gauche
_CLEFT: Dec XCU
If XCU<0
  XCU=39 : If YCU>0 : Gosub _CUP : End If
End If
Locate XCU,YCU
Return
'
' Vers le bas
_CDOWN:
If YCU=0
  XCU=OXCU : YCU=OYCU : Gosub _NLINE
Else
  Inc YCU : Gosub _NLINE
  If YCU=25 : YCU=1 : End If
End If
Locate XCU,YCU
Return
'
' Vers le haut
_CUP:
If YCU
  Dec YCU : Gosub _NLINE
  If YCU=0 : YCU=24 : End If
End If
Locate XCU,YCU
Return
'
' Code non utilisé
_NUL: Pop Proc
'
' Codes de mouvement du curseur
_ARR: Gosub _CLEFT : Pop Proc
_AVT: Gosub _CRIGHT : Pop Proc
_HAU: Gosub _CUP : Pop Proc
_BAS: Gosub _CDOWN : Pop Proc
_DLI: XCU=0 : Locate XCU,YCU
Gosub _NLINE : Gosub _DELIM : Pop Proc
'
' Efface jusqu'à la fin de la ligne
_CAN: Paper CPAP : Gosub _SINV
ACAN=CPAP+$80
If XCU<40
  Locate XCU,YCU
  For X=XCU To 39
    Print " ";
    Poke AATR+YCU*40+X,ACAN : Bclr 7,ACAN
  Next
  Locate XCU,YCU
End If
Pop Proc
'
' Curseur
_CON: Curs On : Pop Proc
_COF: Curs Off : Pop Proc
'
' Home / Clear
_HOM: XCU=0 : YCU=1
Locate XCU,YCU : Gosub _DEF : Pop Proc
_CLS: Gosub _DEF
Window 1 : Clw : Window 0 : XCU=0 : YCU=1
Fill AATR+40 To AATR+25*40,0 : Pop Proc
'
' Mode mosaïque
_GG1: _SET_FONT[1] : GMODE=1
CINV=0 : Gosub _SINV : Gosub _DEF2 : Pop Proc
_GG0: _SET_FONT[0] : GMODE=0 : Gosub _DEF2 : Pop Proc
'
' Caracteres spéciaux du jeu G2
_GG2: CMODE$="G22" : Pop Proc
_G22: PAR1=Instr(G2$,Chr$(A))
If PAR1 : CMODE$="G23" : Pop Proc : End If
_SET_FONT[3] : Gosub PRNT$ : _SET_FONT[GMODE]
CMODE$=OCMODE$ : Pop Proc
_G23: B=Instr(G2$,Chr$(A),PAR1)
If B : A=Asc(Mid$(G2$,B+1,1)) : Gosub PRNT$ : End If
CMODE$=OCMODE$ : Pop Proc
'
' Locate
_LOC: CMODE$="LO2" : Pop Proc
_LO2:
If A>=$40
  ' Locate direct, en Y
  Y=A-$40
  If Y=0 and YCU<>0 : OYCU=YCU : OXCU=XCU : End If
  If Y>=0 and Y<25 : YCU=Y : End If
Else
  ' En début de ligne, dizaines
  PAR1=A-$30
End If

```

```

CMODE$="LO3" : Pop Proc
_LO3:
If A>=$40
  ' Direct, en X
  X=A-$41 : If X>=0 and X<40 : XCU=X : End If
Else
  ' En début de ligne, unités
  Y=PAR1*10+A-$30
  If Y>=0 and Y<25 : YCU=Y : XCU=0 : End If
End If
CMODE$=OCMODE$ : Gosub _DEF : Locate XCU,YCU
Pop Proc
'
' Repetition
_RPT: CMODE$="RP2" : Pop Proc
_RP2: PAR1=A-$40 : A=ALAST : CMODE$=OCMODE$
For REP=1 To PAR1 : Gosub PRNT$ : Next : Pop Proc
'
' Saute le prochain code
' (par exemple, appui sur une touche du minitel)
_SKP:
_CLA: CMODE$="CL2" : Pop Proc
_SK2:
_CL2: CMODE$=OCMODE$ : Pop Proc
'
' Passage en mode ESCape
_ESC: CMODE$="ES2" : Pop Proc
_ES2: CMODE$=OCMODE$
If(A>=$40) and A<=$60 Then Goto ESC$(A)
If A>=$30 Then CMODE$="SK2" : Pop Proc
If A>=$20 Then CMODE$="SKP" : Pop Proc
Pop Proc
'
' Codes ESC : Paper / Pen
_PAP: If GMODE=0 : GOPAP=A-$50 : Pop Proc : End If
CPAP=A-$50 : Paper CPAP : Gosub _SINV : GOPAP=-1
Pop Proc
_PEN: Pen A-$40+FLSH : CPEN=A-$40 : Gosub _SINV
Pop Proc
' Codes ESC : Mosaïques séparées ou non
_G1N: If GMODE=2 : _SET_FONT[1] : GMODE=1 : End If
Pop Proc
_G1L: If GMODE=1 : _SET_FONT[2] : GMODE=2 : End If
Pop Proc
' Codes ESC : Inverse on / off
_INV: CINV=-1 : Inverse On : Pop Proc
_NOR: CINV=0 : Inverse Off : Pop Proc
' Codes ESC : Taille des caracteres
_GNN: PRNT$="NNPRINT" : Pop Proc
_GDX: If GMODE=0 : PRNT$="DXPRINT" : End If : Pop Proc
_GDY: If GMODE=0 : PRNT$="DYPRINT" : End If : Pop Proc
_GDD: If GMODE=0 : PRNT$="DDPRINT" : End If : Pop Proc
' Codes ESC : Flash On / Off
_GFL: FLSH=8 : Pen CPEN+FLSH : Pop Proc
_NFL: FLSH=0 : Pen CPEN : Pop Proc
'
' Reinitialisations lors de LOCATE
_DEF: GMODE=0 : _SET_FONT[GMODE]
Pen 7 : CPEN=7 : FLSH=0 : Paper 0 : CPAP=0 : GOPAP=-1
CINV=0 : Gosub _SINV
_DEF2: PRNT$="NNPRINT"
Return
'
_NLINE:
If GMODE=0
  Paper 0 : CPAP=0 : GOPAP=-1 : Gosub _SINV
End If
PRNT$="NNPRINT" : Return
'
End Proc
'
Procedure _SERIAL_GET
' Attend un caractere de la série
' ou du clavier
Repeat
  A=Serial Get(0)
  A$=Inkey$
  If A$<>"" : A=-Asc(A$) : Exit : End If
Until A>=0
End Proc[A]
'
Procedure _SET_FONT[N]
' Adresse du jeu de caractere dans la banque
F=Start(15)+128*8*N+Sgn(N)*128*8
' Adresse dans la structure fenetre
A=Leek(Screen Base+170)+8
'
Loke A,F
End Proc

```

Par François et Daisy Lionet

AREXX : LES BIBLIOTHEQUES

Après avoir vu le traitement des erreurs et défauts d'un programme ARExx, nous pouvons nous lancer dans l'utilisation de deux bibliothèques, la rexxarp.library et la rexxmathlib.library.

Bien que ce ne soit pas la vocation d'ARExx, il a été muni par W. Langeveld, de ces deux librairies qui permettent d'une part de profiter des avantages d'Intuition (écrans, fenêtres, menus et gadgets de toutes sortes) et d'autre part, de calculer l'inverse de la tangente hyperbolique sans trop se casser la tête. La librairie graphique - ou du moins visuelle - est moins riche qu'Intuition, mais elle permet de réaliser des écrans passablement conviviaux, c'est-à-dire ergonomiques, même si la variété, par exemple des gadgets, est assez limitée. Quant à la librairie mathématique, elle est, elle, plutôt riche.

Trois librairies spécifiques doivent être présentes dans le répertoire Libs: pour que l'on puisse espérer modeler la visualisation à son gré : rexxsupport.library, rexxmathlib.library et rexxarplib.library. Il est recommandé de tester leur présence dès le début du programme :

```
IF ~SHOW('1', "rexxarplib.library") THEN DO
  test=ADDLIB('rexxarplib.library', 0, -30, 0)
  IF TEST ~1 THEN DO
    SAY 'rexxarplib est introuvable'
    CALL DELAY 100
    EXIT
  END
END
END
```

0, -30 et 0 correspondent respectivement à la priorité, l'offset (ou décalage) et à la version ; on les laisse traditionnellement à cette valeur, surtout l'offset !

Cet exemple doit être étendu à chacune des librairies précitées. Dans le cas présent, le test regarde si la librairie a été effectivement ajoutée au traitement résident, sinon il annonce le défaut pendant *DELAY 100* avant de quitter le programme. Dès que tout est en ordre on peut se lancer.

ECRANS PUBLICS

Ces écrans sont réputés publics parce qu'on peut les adresser et agir sur eux par leur nom, qui est connu. Ce pourra être l'écran que l'on va définir ou un autre dont on connaît le nom (l'adresse, en fait !). Cette définition d'écran n'est pas utile si l'on veut se servir de l'écran courant uniquement. On rappelle ici qu'un écran ne définit que le mode graphique : nombre de bitplanes (correspondant au nombre maximal de couleurs) et résolution en lignes et colonnes. Un écran doit suivre des règles précises, en particulier pour ce qui est de la superposition des écrans de résolution différentes. Dans le cas présent, la seule liberté qui nous est laissée est la hauteur à laquelle commence l'écran, à partir du haut.

Pour ouvrir un écran, on utilise la séquence suivante :

```
resultat=OpenScreen(haut, profond, mode, titre, nom)
```

- *haut* : hauteur d'ouverture à partir du sommet de l'écran matériel.
- *profond* : nombre de bitplanes. C'est la puissance de 2 donnant le nombre de couleurs, en accord avec le mode. Si profond = 3, le nombre de couleurs possibles est 2 puissance 3 = 8.
- *mode* : outre le mode normal (mode=0), on peut choisir *HIRES*, *LACE*, *HAM*, *SCREENBEHIND* et *TRUNCATE*. *HIRES* donne 640 points horizontaux et 16 couleurs maximum, *LACE* est l'entrelacement (donc en PAL, donne 512 lignes) et *HAM* donne accès au mode 4096 couleurs. Quant à *TRUNCATE*, il permet de limiter la hauteur de l'écran créé à sa partie visible si l'on a utilisé haut non égal à 0. Sinon, l'écran aurait une hauteur normale déplacée de haut à partir du sommet et consommant de la mémoire pour rien.
- *titre* : nom donné à l'écran dans la ligne de titre, et qui est purement arbitraire ou décoratif.
- *nom* : c'est le nom public, c'est à dire l'adresse, qui permettra de s'adresser à cet écran.

Tous les arguments doivent être pris en compte. Donc, s'il y en a qui sont laissés à la valeur par défaut, il faut marquer leur position par les virgules de la séquence d'appel. Certains peuvent co-exister avec d'au-

tres (*LACE* en particulier), ou peuvent être incompatibles (par exemple 6 bitplanes et *HIRES*). Si incompatibilité il y a, l'appel ne réussit pas. Lorsque l'appel a réussi, resultat doit être égal à 1, sinon il vaut 0.

Enfin, un écran doit être fermé après emploi :

```
resultat=CloseScreen(nom).
```

En cas de défaut, il se peut que le programme se termine sans passer par la fermeture des écrans publics qu'il aura ouverts. Or, il n'est plus possible de réouvrir cet écran après rectification du ou des défauts ; il faut d'abord le fermer. Une solution est de se rappeler le nom de l'écran et à partir de la console taper : `rx 'closescreen(nom)'`

OpenScreen et *CloseScreen* s'appliquent indifféremment à tous les écrans publics ouverts par vous ou par une autre application. La seule connaissance nécessaire est celle du nom de l'écran considéré.

PARAMETRES D'UN ECRAN

Il peut être intéressant ou utile de connaître les caractéristiques d'un écran public que l'on veut utiliser et que l'on n'a pas défini soi-même. Quatre fonctions spéciales effectuent ce travail :

- *lignes* = *ScreenRows(nom)* donne le nombre de lignes de l'écran nom ;
- *colonnes* = *ScreenCols(nom)* donne le nombre de colonnes ;
- *lace* = *ScreenLace(nom)* donne 1 si l'écran est en mode entrelacé, et 0 sinon ;
- *couleur* = *ScreenColors(nom, numero)* donne pour le numéro de couleur choisi, une chaîne RVB valant de 0 à 15 par couleur Rouge, Verte et Bleue.

Pour les trois premières fonctions, le nom est optionnel et s'il n'est pas donné c'est l'écran Workbench qui est pris par défaut. Le nom est nécessaire pour la dernière.

Si aucune valeur n'est disponible, par exemple par mauvaise appellation de l'écran, ou parce qu'il n'existe pas, alors -1 est retourné.

Position relative d'un écran

Lorsqu'un écran est créé, on peut le placer à volonté devant tous les autres au moyen de *ScreenToFront(nom)*, ou derrière avec *ScreenToBack(nom)*. Ces deux fonctions retournent 1 en cas de succès et 0 dans l'autre cas.

Barre d'écran

La fonction *ShowTitle(nom, ouinon)* permet de faire apparaître le titre de l'écran ou de le cacher à volonté. Si ouinon vaut 1, la barre de titre montre le titre de l'écran nom, sinon elle le cache.

C'est sur cette instruction, dont l'importance n'échappera pas aux plus attentifs de nos lecteurs, que nous fermerons le chapitre écran !

FENETRES

C'est la partie visible de la création d'un environnement. Pour pouvoir interagir avec la fenêtre, il faut créer des moyens d'accès nommés Ports selon l'usage en vigueur dans le système de l'Amiga. Il y a deux types d'accès à une fenêtre : l'écriture ou la lecture de quelque chose se passant dans cette fenêtre. Il va donc falloir créer deux canaux de communication, l'un allant vers la fenêtre, l'autre venant de la fenêtre. Il n'est pas nécessaire que ces deux canaux (Ports) existent simultanément : on ne crée que ceux dont on a besoin. Enfin, on ne peut créer de telles relations qu'avec un écran connu, ou le Workbench par défaut.

Création des ports

Il y a plusieurs façons de créer les ports.

```
Address AREXX "'CALL CreateHost(ECRIT,LIT,nom)'"
```

ce qui, traduit en quelque chose de supposé plus intelligible, veut dire : ordre à AREXX (port permanent du système) de créer un port *ECRIT* permettant d'écrire et un port *LIT* permettant de recevoir des informations de la fenêtre qui va être créée dans l'écran : nom. De la même façon, on pourrait écrire :

```
RunWsh 'CALL CreateHost(ECRIT,LIT,nom)'
```

Il est alors utile de patienter quelque peu, le temps de la création. Voici différentes façons de faire.



```
DO FOR 50 WHILE ~Show('Ports','ECRIT')
CALL DELAY 10
END
```

Ceci teste régulièrement la présence du port en construction ; le *FOR 50* étant là pour limiter la durée si, par hasard, il n'y avait pas d'ouverture possible. Une autre possibilité est d'utiliser la fonction :

```
retour = WaitForPort ECRIT
```

ou encore :

```
WaitForPort -immediate ECRIT
```

qui donne dans retour et au bout de 10 secondes maximum, 0 si le port existe ou 5 dans le cas contraire.

Il faut faire attention à respecter les majuscules et minuscules dans l'orthographe du nom du port. En règle générale, simplifier au plus le nom du port est une heureuse initiative !

Notons enfin qu'il faut ouvrir le port :

```
CALL OpenPort(ECRIT)
```

La fenêtre proprement dit

Une fenêtre est caractérisée par plusieurs éléments :

- le nom de son port d'écriture, ici *ECRIT* ;
- ses dimensions et sa position, qui sont données sous la forme position à gauche du début de la fenêtre, position à partir du haut de l'écran, largeur et hauteur ;
- son nom ;
- les gadgets standards des fenêtres : tirage, fermeture, dimensions...
- les messages IDCMP qu'elle est peut émettre le cas échéant - en supposant qu'elle ait été construite en ce sens.

Pour simplifier l'écriture d'une instruction passablement longue, on utilise traditionnellement deux chaînes : *IDCMP* et *FLAGS* (ou le nom que l'on veut !).

```
FLAGS='WINDOCLOSE+WINDOWDRAG+WINDOWDEPTH+WINDOWSIZING'
```

On peut encore y ajouter *BORDERLESS*, *NOCAREREFRESH*, *ACTIVATE* ou *BACKFILL*, chacun de ces termes étant présent ou absent selon le besoin.

IDCMP représente les messages qui seront émis par la fenêtre. Ils sont nombreux et concernent les gadgets, les menus, la position de la souris, le lecteur de disquettes (pour savoir si l'on a changé la disquette), etc. On aurait par exemple :

```
IDCMP='CLOSEWINDOW+GADGETDOWN+DISKINSERTED+MOUSEMOVE'
```

et bien d'autres (*NDLR* : on ne va tout de même pas vous refaire un cours complet sur les fenêtres, non ?).

L'ouverture de la fenêtre s'effectue selon le cérémonial suivant :

```
FLAGS='...'
IDCMP='...'
CALL OpenWindow(ECRIT,haut,larg,hauteur,IDCMP,FLAGS,'titre')
```

- *haut* est la distance à partir du haut de l'écran ;
- *largeur* est... la largeur de la fenêtre ;
- *hauteur* représente... sa hauteur ;
- *titre* est une chaîne de caractères, spécifiant le titre voulu.

Comme d'habitude en ARexx, les termes entre guillemets ou apostrophes sont pris littéralement. Les autres sont des variables et hauteur, par exemple, peut être le résultat d'une opération tout-à-fait quelconque.

De la même façon que pour les écrans, une fenêtre peut être placée devant ou derrière toutes les autres, dans son écran.

```
CALL WindowToFront(nomduport)
CALL WindowToBack(nomduport)
```

où *nomduport* est le nom du port d'écriture dans la fenêtre.

On peut également la rendre active par *CALL ActivateWindow(nomduport)*.

Enfin, il faut pouvoir la fermer : *CloseWindow(nomduport)*, *Exit(nomduport)*, *Quit(nomduport)* et *Stop(nomduport)* sont synonymes.

Ces commandes, dont la multiplicité n'a pas de raison ouvertement avouée, ferment la fenêtre considérée, désallouent les ressources et provoquent la sortie, à l'exception de *CloseWindow* qui peut supporter une option *CloseWindow(nomduport, 'CONTINUE')* permettant à l'hôte de rester actif tout en fermant la fenêtre.

Comme dans le cas de l'écran, en cas de sortie intempestive du programme, sur défaut par exemple, la ou les fenêtres en cours ne sont pas automatiquement fermées. On peut les fermer à partir de la console par *rx 'Closewindow(nomport)'*. On devra fermer d'abord toutes les fenêtres d'un écran avant de lancer la fermeture de l'écran lui-même. On voit l'intérêt de se rappeler le nom des fenêtres ou celui des écrans pendant la mise au point et de ne leur donner que des noms simples. Sinon c'est le "re-bootage" assuré.

Ce que raconte une fenêtre

Nous verrons par la suite comment extraire un message issu d'une fenêtre. Pour le moment, on admet avoir reçu un message. Que contient-il ? Si rien n'a été précisé (autre que les IDCMP), le message contient par exemple : *CLOSEWINDOW* si *CLOSEWINDOW* a été sollicité, *DISKREMOVED* si la disquette a été retirée et l'IDCMP correspondant initialisé... A part la chaîne contenue dans le gadget et qui est retournée par les IDCMP *GADGETUP* et *GADGETDOWN*, on ne reçoit que le nom ou la classe de l'IDCMP. En particulier avec *MOUSEMOVE*, on ne sait qu'une seule chose : la souris a été déplacée. Cela peut suffire mais c'est plutôt la quantité dont elle a été déplacée qui est intéressante.

Pour les menus c'est la même chose. Il faut donc préciser ce que l'on veut recevoir. C'est l'objet de la commande *ModifyHost*.

Le message envoyé peut être organisé en arguments consécutifs, contenant chacun ce que l'on souhaite. Par défaut, on obtenait sans avoir rien précisé l'*Arg(0)*, qui contenait les noms des IDCMP. Nous allons pouvoir préciser ce que l'on veut et dans quel argument (de 0 à 15), on veut le trouver. On invoque le modificateur de la façon suivante :

```
CALL ModifyHost(nomport,IDCMP,'xxxxxxxxxxxx')
```

IDCMP est le message que l'on veut traiter. Imaginons pour l'exemple que ce soit *MOUSEMOVE*. La chaîne d'*x* s'écrit :

```
%3 %x,%y
```

ce qui se traduit : dans l'argument 3, mettre la position *x* puis la position *y* de la souris.

Donc, *CALL ModifyHost(nomport, MOUSEMOVE, %3%x, %y)* permettra ultérieurement (après extraction de l'argument 3) de trouver la position de la souris dans une chaîne du genre : 145 23. La précision de l'argument ne fait qu'effectuer un pré-tri pour le cas où beaucoup de messages sont possibles.

On trouve de nombreuses possibilités. Par exemple, *%m* donne le numéro du menu sollicité ; *%s* permet d'obtenir le numéro de sous-menu ; *%h* permet d'obtenir la hauteur de la fenêtre alors que *%t* donne le nombre de secondes écoulées depuis le 1 Janvier 1980, à l'instant où l'évènement est survenu.

Il y a 16 types d'IDCMP concernant les fenêtres, la souris et ses boutons, les gadgets, les menus et sous-menus, les touches du clavier, les préférences et les disquettes.

Il y a 18 modifications possibles, et 16 arguments. La souplesse s'obtient en augmentant la complexité.

Alors que la réception de messages met en jeu un mécanisme compliqué, mais tout-à-fait conforme à celui que l'on met en oeuvre avec Intuition, l'écriture dans la fenêtre est beaucoup plus simple et très facilement manipulable. C'est peut-être une consolation.

Avant de passer à des exemples plus concrets, notons encore les commandes *SetNotify(nomport, IDCMP, nouveauport)* qui permet de réaffecter le message émis à un nouveau port et *ReadHost(nomport, portreçois, texte)* qui permet de déclencher l'envoi de texte au portreçois après les modifications éventuellement effectuées par *ModifyHost*.

Nous verrons comment mettre en oeuvre toutes ces considérations théoriques dans de petits programmes, où toutes les complexités précédemment entrevues seront explicitées en détail.

Par François Gueugnon

PASCAL : PRESENTATIONS (III)

Troisième épisode de notre saga sur le Pascal. Ce mois-ci, nous aborderons les types de base, avec un exemple d'implémentation propre à PCQ : les chaînes de caractères.

En informatique, il existe toujours des concepts pré-définis, par exemple l'arithmétique sur des octets signés ou non, le système d'exploitation... qui permettent de ne pas refaire le travail (très difficile pour l'arithmétique au niveau de l'électronique). Pour chaque couche (logicielle pour la plupart), on parle d'une nouvelle machine virtuelle. Elle a porté de nouveaux outils, masqué d'autres plus difficiles d'approche. Le Pascal est une machine virtuelle et est un langage typé : nous avons donc des types prédéfinis. Ils sont nécessaires pour que le compilateur puisse appliquer les ordres suivant les propriétés propres à chacun des types. Le plus simple est de vous les présenter :

- **INTEGER**, auquel correspond la représentation des entiers signés codés sur 4 octets (calculs sur 32 bits). Le nombre maximum positif représentable est 2 147 483 647. Ce nombre étant très facile à retenir, vous pouvez utiliser, en cas d'un oubli momentané, la constante pré-définie **MAXINT**.

- **SHORT** est une autre représentation des entiers signés. Les variables de ce type sont codées sur deux octets. Théoriquement plus rapide que les variables de type Integer sur notre machine, PCQ les étend à quatre octets au niveau des registres. Le seul avantage de ce type est donc de prendre deux fois moins de place qu'un Integer. L'intervalle des nombres pour le type **SHORT** est compris entre -32768 et 32767. La constante pré-définie **MAXSHORT** contient cette valeur.

- **BYTE** est, j'espère ne pas vous surprendre, une représentation des entiers non signés sur un octet. L'intervalle est [0..255] (la notation utilisée ici pour l'intervalle est celle de Pascal). Il n'y a pas de constante comme pour les deux premiers types.

Les trois types présentés ci-dessus sont considérés comme compatibles entre eux : une variable d'un de ces types pourra être affectée à une autre sans prise en compte de son type si celui-ci est un cité plus haut. Nous sommes donc en présence d'un irrespect caractérisé du typage, dans un but d'occupation de zone mémoire et de compatibilité avec les standards de notre bécane (octet, mot, mot long). Hélas, aucun test de dépassement n'est fait. Donc, si vous tapez le programme suivant :

```
Program Octet_Mot_LongMot;
VAR
  i:integer; { 4 octets }
  s:short;   { 2 octets }
  b:byte;    { 1 octets }
BEGIN
  i:=40000; { Nombre plus grand qu'une valeur Short }
  s:=i;     { Une vilaine affectation }
  b:=i;     { Plus méchante qu'un Goto }
  writeln('i=',i,'s=',s,'b=',b) { Pour voir les dégâts }
END.
```

Vous obtenez le résultat suivant :

i= 40000 s= -25536 b= 64

Un joli effet de bord, plus difficile à trouver dans de nombreux calculs. Alors un conseil : pour des valeurs difficiles à quantifier, utilisez des entiers (**INTEGER**).

- **CHAR** est le type permettant de stocker un caractère et d'interpréter sa valeur comme tel et non comme un nombre (vous savez que tout, dans un ordinateur, est représenté par des nombres). Il prend 1 octet en mémoire.

- **BOOLEAN** est le type qui dit "vrai" ou "faux", plus exactement "TRUE" ou "FALSE". Ainsi, nous avons la possibilité de calculer des expressions de la logique de Boole (Boole : 1815-1864).

- Les **Enumérations** sont une catégorie de type qui permet une correspondance entre des nombres et une représentation formelle de différents états d'un objet. Un exemple : le feu tricolore à trois états (vert, orange et rouge) ; en Pascal, on écrit :

```
VAR feu:(vert, orange, rouge);
```

'vert' vaudra ici 0, orange, 1 et rouge, 2. Ainsi, nous pouvons comparer des variables d'une même énumération. Remarquez que l'ordre indique le sens de comparaison. A noter qu'il ne faut pas déclarer deux fois la même énumération sans créer un type propre (ce que nous aborderons au moment de la déclaration de types).

```
Program Essai_Sur_Enumeration;
```

```
VAR
  i,j,l:(vert,orange,rouge);
  { déclaration avec le fameux type feu tricolore }

BEGIN
  i:=vert; { Affectation directe }
  j:=orange; { Idem }

  l:=succ(j);
  { Utilisation d'une fonction particulière au Scalaire,
    qui donne le successeur de la valeur passée en argument }

  writeln('l\'affirmation : \"i est inférieur à l\" est ',i<l);
  { remarquez l'utilisation de la comparaison \"inférieur à\",
    représentée par le signe \"<\", comme valeur d'argument
    booléen. Ce qui est entre les (') sera expliqué plus tard }
END.
```

- **REAL** est le type qui permet d'utiliser l'implémentation des réels au format FFP (Fast Floating Point). Cela nous ouvre les portes des applications scientifiques. La méthode de stockage prend autant de mémoire qu'un entier de type **INTEGER** (4 octets).

Les sept types ci-dessus ont la propriété d'être scalaires, c'est-à-dire qu'ils possèdent toujours un successeur (même s'il peut sembler illogique). De plus, il existe un ordre total : on peut toujours comparer deux valeurs de même type. Toutefois, puisque la mémoire allouée à un type est finie, les valeurs sont des modulus (reste d'une division entière) de la valeur réellement indiquée. Un exemple pour illustrer ceci :

```
Program Valeurs;
{ les valeurs indiquées ne sont pas celles réellement stockées }

VAR
  i:integer;
  s:short;
  b:byte;
  { les réels ayant une implémentation particulière, un exemple ne
    serait pas très explicite (méfiez-vous des réels ! ) }

BEGIN
  { remarquez que tous les nombres affectés dépassent largement la
    borne supérieure des intervalles respectifs de chaque type }

  i:=3_000_000_000;

  { j'utilise une facilité de PCQ qui permet de séparer les chiffres
    d'un nombre par le signe \"_\", merci encore Patrick }

  s:=50_000;
  b:=300;
  writeln('i = ',i,' s = ',s,' b = ',b);

  { si vous avez les mêmes nombre que ceux ci-dessus, votre implémentation
    est supérieure à PCQ. }

END.

{ Et aucune erreur de compilation n'est fournie par PCQ ! }
```

Encore un effet de bord qui, lui, provient de l'implémentation du compilateur.

Pour tous les types arithmétiques (**INTEGER**, **SHORT**, **BYTE**, **REAL**), il existe des opérations mathématiques standards, comme l'addition, la soustraction, la multiplication et la division, et pour les entiers seulement, le modulo (**MOD**) et la division entière (**DIV**). La priorité des opérateurs est normalement respectée (au moins pour les 4 premiers opérateurs). Pour plus de sûreté, vous pouvez utiliser les parenthèses. Voici un petit exemple bête pour se clarifier les idées.



```

Program Arithmetique;
{ utilisation des opérateurs classiques sur plusieurs types }

VAR
  i,j : integer;
  s,t : short;
  b,c : byte;
  x,y : real;

BEGIN
  writeln('Travail sur les entiers : ');
  { multiplication : attention au dépassement de capacité }
  i:= 110 * 44;
  writeln (' i = ',i);
  { addition }
  j:= 115 + 45;
  writeln (' j = ',j);
  { division entière. Remarquez que l'opération se fait direc-
temment sans utiliser une variable intermédiaire }
  writeln (' i divisé par j = ', i div j);
  { reste }
  writeln (' i modulo j      = ', i mod j);
  { division : ici le résultat sera toujours réel }
  writeln (' s divisé par t = ', i / j);
  writeln;

  writeln ('Travail sur les entiers courts : ');
  { utilisation du moins unaire }
  s:= 10 * (-4);
  writeln (' s = ',s);
  t:= 15 + 40;
  writeln (' t = ',t);
  writeln (' s divisé par t = ', s div t);
  writeln (' s modulo t      = ', s mod t);
  writeln (' s divisé par t = ', s / t);
  writeln;

  writeln ('Travail sur les octets : ');
  b:= 10 * 25;
  writeln (' b = ',b);
  { soustraction }
  c:= 15 - 10;
  writeln (' c = ',c);
  writeln (' b divisé par c = ', b div c);
  writeln (' b modulo c      = ', b mod c);
  writeln (' b divisé par c = ', b / c);
  writeln;

  writeln ('Travail sur les réels : ');
  x:= 10.0 + 4.3; { remarquez que le point-zéro ".0" est néces-
saire pour que le compilateur interprète les nombres des réels }
  writeln (' x = ',x);
  y:= 15.0 * 40.7;
  writeln (' y = ',y);
  writeln (' x divisé par c = ', x / y);
END.

```

Si vous avez regardé la documentation de PCQ, il reste trois types de base :

- **Address** : c'est aussi un type scalaire comme INTEGER. Il permet d'indiquer une adresse-mémoire. Sa valeur est l'adresse d'un octet, d'un mot ou d'un long mot. Ce type est le type pointeur le plus général (les pointeurs seront étudiés au moment des constructeurs de types). Un exemple direct, pas très propre, mais facile à implémenter, est l'accès aux registres de couleurs du hardware.

```

Program Couleurs_Pas_Proprie;
{ permet de modifier le registre COUL0 pendant un très court mo-
ment }

VAR
  Coulo : address;
  i:short;

BEGIN
  { adresse de la couleur 0. J'utilise un cast (un changement de
type) car $DFF180 est de type INTEGER et Coulo de type ADDRESS. }
  Coulo :=address($DFF180);

  for i:=1 to 1000 do
    { "For" indique une répétition (une boucle), ici 1000 fois
l'affectation suivante :}

    Coulo^:=i;
    { écriture d'un mot dans le registre, remarquez le "^" indi-
quant que c'est à l'adresse indiqué par la valeur de coulo que la
valeur de i doit être rangée }

  END.

```

Cette méthode n'est toutefois pas du tout recommandable. Je tiens à préciser qu'il faut respecter le système d'exploitation !

- **Text** : avec ce type, on touche au système de gestion de fichiers, chose que nous ne ferons pas avant longtemps car, bien que simplifié en Pascal, il nous faut d'autres connaissances. Il sert pour la gestion de fichiers au format ASCII (comme, par exemple, une fenêtre Shell).

Ces deux types ne sont pas des types pré-définis standards du Pascal, mais ils existent et permettent de simplifier certaines approches de la machine.

Le dernier type a une implémentation différente en PCQ de celle des autres compilateurs Pascal. Je vais donc vous décrire le fameux mais néanmoins folklorique type *STRING*.

L'un des problèmes de Pascal est justement son imprécision sur l'implémentation du type chaîne de caractères. Patrick Quaid utilise deux sortes de chaînes de caractères : la première est en fait un tableau, la seconde est ce que l'on nomme l'indirection. Elle correspond au type *STRING* de PCQ.

On ne peut pas utiliser les opérateurs classiques que l'on rencontre dans d'autres Pascals : concaténation, reproductions multiples... Mais l'implémentation est faite sur la base des chaînes de caractères dans le langage C. Une variable de type *STRING* n'est autre qu'un pointeur sur une zone de mémoire (en fait un tableau de caractères). Celle-ci est particulière. Elle contient tout les caractères et se termine par un caractère spécial dont la valeur équivaut à 0.

Plusieurs points sont à éclaircir :

- Un pointeur est une variable qui désigne exclusivement un seul type d'objet, c'est-à-dire qu'il a comme valeur l'adresse en mémoire de l'objet sur lequel il pointe. D'autres explications, avec des exemples, viendront plus tard dans le chapitre sur les constructeurs de types. Dans le cas présent, nous avons un pointeur sur des variables de type *CHAR*.
- Un tableau est une zone de mémoire continue dont la longueur est fixe et qui ne contient que le même type d'éléments. Il peut être considéré comme une application d'un sous-ensemble fini de $\mathbb{N}$, vers les éléments contenus dans cette zone mémoire. On déclare un tableau avec le constructeur de type *ARRAY*.
- Comme je l'ai précisé pour les variables de type *CHAR*, ils sont codés sous forme numérique sur un octet (convention ASCII étendue). le caractère NUL (dont la représentation est 0) est utilisé comme marque de fin de chaîne.
- Il faut tout de suite bien différencier la longueur d'une chaîne et la longueur du tableau qui contient la chaîne ; ou plus généralement, différencier le contenu du contenant. La séquence de caractères qui représente la chaîne, doit être plus petite que le tableau contenant celle-ci, sinon la chaîne "mangerait" de l'espace qui pourrait être important et, donc, provoquer une erreur (Guru !!!).

Comment distinguer les constantes chaînes de caractères et les tableaux de caractères ? Par l'utilisation des délimiteurs : si l'on utilise les guillemets ("), nous aurons une chaîne de type *STRING*, si nous utilisons les apostrophes ('), nous aurons un tableau de caractères. Cette distinction n'est utile que dans le cas d'appel de procédures implémentées uniquement que pour l'un de ces deux types. Par exemple, la procédure *WRITE* accepte les deux types mais pas la fonction *STRLEN*, qui a besoin d'une variable de type *STRING*. Un petit exemple :

```

Program Les_Chaines;
{ Comment déclarer des constantes chaines sans utilisation de va-
riables }

BEGIN
  Writeln ('Cette chaine de caractères est un tableau');
  Writeln ("aCelle-ci est une STRING\navec des caractères d'é-
chappement");
  { remarquez les caractères d'échappement introduit par "\" }
END. { un peu court non ?}

```

Nous finirons ce chapitre sur les chaînes de caractères la prochaine fois. Bye et à Plus.

Par Olivier Garrigues

Aujourd'hui, nous allons revisiter le célèbre PRINT "Hello" de nos débuts. Mais à la vitesse grand V, soit 16 700 caractères par seconde !

Afficher un million de caractère à la minute est irréalisable avec la plupart des compilateurs. GFA est cependant si optimisé qu'il prend ici une incroyable avance sur la plupart des compilateurs C. Mais il est possible d'aller encore plus vite. Comment se définit une fonction PRINT ? On prend une chaîne de caractères, on lui colle une police de caractères, puis on affiche chaque symbole de la chaîne. Eventuellement, on définira certains codes spéciaux tels que la tabulation, le style ou la couleur de la lettre. Puis vient la gestion de l'affichage.

Pour simplifier, je n'utilise ici que la police Topaz 8, sinon gare au Gourou (non, ce n'est pas vrai... Ou alors, mettez votre version 3.03 à jour), et l'affichage se fera sur un seul plan de bits (mode 2 couleurs).

Nous nous retrouvons donc avec un affichage ligne à ligne de 79 caractères. Le petit test de chronométrage montre alors l'intérêt de la frappe au kilomètre (pas de touche RETURN) : dans ces conditions, on obtiendra des temps de 50 à 51 microsecondes pour l'impression de chaque lettre sur un A3000. Sinon, avec un PRINT toutes les 40 lignes, cela deviendra 190 sur A500. Ces temps s'entendent bien sûr avec l'utilitaire FF activé dans la Startup-Sequence.

Et on peut encore accélérer, mais il faudra alors utiliser un assembleur (Devpac s'il fallait le nommer). En effet, est-ce malice de Franck Ostrowski ou tout simplement une subtilité qui m'aurait échappé, mais la plupart des fonctions C du GFA donnent quelque chose qui ressemble (en source compilé avec l'option table des symboles) à un code comme :

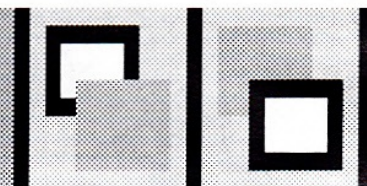
```
MACHIN move $c498a0(a5),d0
        move $c496a2(a5),d1
        bsr Machin_G
        ....
Machin_G  ext.l d0
        ext.l d1
        move.l a6,d7
        movea.l $65e(a6),a6
        jsr _LVOMACHIN(a6)
        move.l d7,a6
        rts
```

Ainsi, on pourrait se demander à quoi sert la ligne *exg*. Et malgré tout, cela va vite. Très, très vite.

```
' *****
' * P R I N T   A S C I I
' * Gestion fin de ligne et bas écran
' *
' * CODAGE      Pierre Philippe Launay
' * RECHERCHE   Cathy Lardy
' * COPYRIGHT   Décembre 1991
' *****
```

```
vite
PROCEDURE vite ! Plan du codage
met.ecran      ! Il est possible d'
met.variable   ! Aller plus vite
affichage      ! En optimisant d' encore
resultat       ! Ce code
salut          ! A vos crayon...
RETURN
PROCEDURE met.variable ! Tout
```

```
IF fichier$=""
    nom$=_dosCmd$
    DIM ecran$(1)
ELSE
    nom$=""
ENDIF
return%=0
essais&=0
ligne%=8
longueur%=79 ! Nombre de lettres/ligne
vide$=SPACE$(80)
vide%=V:vide$
REPEAT
REPEAT
    IF nom$=""
        FRONTS 1
        FILESELECT "SELECTIONNEZ LE FICHIER
ASCII A LIRE"," LIRE ","DF0:",nom$
        IF nom$=""
            fins
        ENDIF
    ELSE
        commentaire%=INSTR(nom$,";")
        IF commentaire%
            nom$=LEFT$(nom$,commentaire%-1)
        ENDIF
        nom$=TRIM$(nom$) ! Ote les espaces
        guil$=CHR$(34)
        guil1%=INSTR(nom$,guil$)
        guil2%=INSTR(guil1%+1,nom$,guil$)
        IF guil1%
            ! Si y en a un
            IF guil2%=0 OR LEN(nom$)<2
                ALERT 0,"Gasp ! Il faudrait 2 "
+guil$+" bien placés pour encadrer le
nom du fichier :|C ' est foutu.",1,
" Clinorhombique ? ",bof%
            nom$=""
            ELSE
                ! C' est bon
                long%=guil2%-guil1%-1
                nom$=MID$(nom$,guil1%+1,long%)
            ENDIF
        ENDIF
        nom$=TRIM$(nom$) ! Ote les espaces
    ENDIF
    IF NOT EXIST(nom$)
        nom$=""
    ENDIF
    INC essais&
    UNTIL nom$<>"" OR essais&=3
    IF essais&=3 OR
INSTR(nom$,";")=LEN(nom$)
    IF nom$="" OR (NOT EXIST(nom$)
OR INSTR(nom$,";")=LEN(nom$)
        ALERT 0,"L'objet ne convient pas"
,1,"Eustatique, n' est-il pas ?",bof%
        nom$=""
    ENDIF
    ENDIF
    IF nom$<>""
        OPEN "i",#0,nom$
        IF LOF(#0)>32767
            CLOSE #0
            ALERT 0,"Fichier trop long|Maximum
32767 caractères",1,"Choliambe ?",bof%
            nom$=""
        ENDIF
    ENDIF
    UNTIL nom$<>""
    fichier$=INPUT$(LOF(#0),#0)
    fichier%=V:fichier$
    CLOSE #0
    long%=LEN(fichier$)
    debut%=fichier%
    CLS
    TEXT 12,25,"Attention : ça déménage"
    TEXT 12,45,"GAUCHE DROIT"
    TEXT 12,55,"Visible Invisible"
    clic
    pos%=1
    seconde%=0
    tick%=0
    compteur%=0
    departsec%=0
    departtick%=0
    vdepartsec%=V:departsec%
```

```

vdeparttick%=V:departtick%
chronosec%=0
chronotick%=0
vchronosec%=V:chronosec%
vchronotick%=V:chronotick%
RETURN
PROCEDURE affichage      ! Le test
IF sourisk%=2           ! Invisible ?
    DISPLAY OFF         ! Alors on cache
ENDIF
~CurrentTime(V:seconde%,V:tick%)
INC seconde%
REPEAT
    ~CurrentTime(vdepartsec%,vdeparttick%)
UNTIL departsec%=seconde%
FOR x%=1 TO long%
    SELECT BYTE{fichier%+x%}
    CASE 10
        DEC pos%
        @return
        ~Text(rastport%,vide%,78-pos%)
        pos%=1
    DEFAULT      ! Ou symboles spéciaux
        SELECT pos%
        CASE 78
            @return
            pos%=1
        DEFAULT
            INC pos%
        ENDSELECT
    ENDSELECT
NEXT x%
~CurrentTime(vchronosec%,vchronotick%)
DISPLAY ON      ! Visible
RETURN
PROCEDURE return
INC return%
SELECT ligne%
CASE 0 TO 177
    ADD ligne%,8
DEFAULT
    ' ~OwnBlitter()
    ~WaitBlit()
    DPOKE &HFFF096,&HC010      ! DMACONR
    LPOKE &HFFF044,&HFFFFFFF ! Masque
    DPOKE &HFFF040,&H9F0      ! A et D
    LPOKE &HFFF050,source%
    LPOKE &HFFF054,cible%
    LPOKE &HFFF064,0          ! BLT%MOD
    DPOKE &HFFF058,taille%
    ' ~DisOwnBlitter()
    ~WaitBlit()
ENDSELECT
~Move(rastport%,2,ligne%)
~Text(rastport%,debut%,pos%)
debut%=x%+fichier%+1
RETURN
PROCEDURE resultat      ! Go Garry, Go
compteur%=long%
durée%=chronosec%-departsec%
delais%=ADD(MUL(durée%,1000000),
SUB(chronotick%,departtick%)) ! En s
$%0      ! Division EXPO#
ici=delais%/compteur% ! En nombre réel
$%3      ! Division LONG%
clic
SGET ecran$()
~RectFill(rastport%,5,12,400,145)
~SetDrMd(rastport%,2) ! Moins rapide
PBOX 10,70,360,85      ! Plus rapide
~SetOpen(rastport%,0) ! Moins rapide
GRAPHMODE 1
TEXT 10,20,"Seconde Système"
TEXT 10,30,"Tick Système  "
TEXT 10,50,"Durée totale  "
TEXT 10,60,"Pagination   "
TEXT 10,80,"IMPRESSION    "
~SetOpen(rastport%,1)
TEXT 137,20,STR$(chronosec%,10)+" S -"+
STR$(departsec%,11)+" S "
TEXT 137,30,STR$(chronotick%,10)+" -"
+STR$(departtick%,11)+" "
TEXT 137,50,STR$(delais%/1000000,18,6)+
" Secondes "
TEXT 137,60,STR$(return%,6)+

```

```

" Lignes de"+STR$(longueur%,3)+
" lettres "
TEXT 137,80,STR$(compteur%,19)+
" lettres "
~SetDrMd(rastport%,2)
TEXT 10,110,"L' affichage d' une lettre
GFA prend ici "
TEXT 10,120,STR$(ici)+
" microsecondes. "
TEXT 10,140," Veuillez utiliser le
bouton de la souris "
~SetDrMd(rastport%,1)
RETURN
PROCEDURE salut      ! A bientôt
clic      ! Laché de souris
SPUT ecran$()
ALERT 0," SOUHAITEZ VOUS | LIRE UN
AUTRE | FICHER ASCII ?|",2,
" Oui | Non ",bof%
SELECT bof%
CASE 2
    PBOX 5,12,400,145
    TITLEW #0,"PASSEZ UNE BONNE ET
AGREABLE JOURNEE",SPACE$(70)
FOR x%=0 TO 99
    COLOR x% MOD 2
    PBOX 5,12,400,145
NEXT x%
TEXT 10,20,"GFA 3.52F"
TEXT 152,20,"Pierre Philippe Launay"
DELAY 2.5      ! A bientôt
fins      ! sniff
DEFAULT
    ferme.ecran
    vite
ENDSELECT
RETURN
PROCEDURE clic      ! Clac
WHILE MOUSEK
WEND
REPEAT
    sourisk%=MOUSEK
UNTIL sourisk% OR STRIG(1)
OR INKEY$<>"
RETURN
PROCEDURE met.ecran      ! Oh Voui
OPENS 1,0,0,640,200,1,&H8000
TITLES #1,"Code PPL (Ant 1992)"
titres$="GFA 3.52F est vraiment très
rapide : qui en doute encore ?"
SETCOLOR 0,&H378      ! bleuâtre
SETCOLOR 1,&HFD8      ! jaune
planbit%=LONG{SCREEN(1)+&HC0} ! Plan 0
source%=planbit%+0+28*640/8
cible%=planbit%+0+20*640/8
taille%=168*64+640/16
,
OPENW #0,0,10,640,189,0,&H1000
titrew$="Bonjour et bienvenue dans
notre monde, le monde fantastique du
développement"
TITLEW #0,titrew$,titres$
rastport%=LONG{ADD(WINDOW(0),50)}
RETURN
PROCEDURE ferme.ecran ! Déjà
CLOSEW #0
CLOSES 1
RETURN
PROCEDURE fins ! Notre ami le système
ferme.ecran
SYSTEM
RETURN

```

Par Pierre-Philippe Launay

ALGOS : LES FILES

Petite définition : une file est une structure de données représentée par un vecteur, où tout ajout d'élément se fait en queue de file et tout retrait, en tête de file.

Nous avons donc à faire à une structure de données de type FIFO (First In, First Out), contrairement à la pile qui, comme nous l'avons vu précédemment, est de type LIFO. La file peut être représentée soit par un tableau, soit par une liste chaînée. Sa gestion n'est pas aussi simple que celle de la pile, pour la bonne et simple raison qu'il existe plusieurs méthodes d'implantation : la gestion contigüe, la gestion contigüe avec décalage et la gestion circulaire.

Nous allons donc étudier successivement ces trois méthodes, en décrivant pour chacune d'elles ses primitives : initialisation de la file, ajout d'un élément (enfiler), retrait d'un élément (défiler), ainsi que les grandes classiques booléennes que nous ne présentons plus : file_vide et file_pleine. Toutes les primitives sont données pour les tableaux, afin que nous puissions effectuer des comparaisons valables entre les différentes méthodes.

Commençons par la plus simple : la gestion contigüe de la file. Partant d'une file vide nous devons déterminer une valeur pour chacune de nos bornes (TETE et QUEUE). Ces valeurs ne peuvent pas être choisies au hasard, puisqu'elles doivent vérifier le test file_vide. Il est hors de question de les initialiser n'importe comment et d'utiliser une variable booléenne qui nous indiquerait l'état de la file. Dans ce premier cas, TETE vaut 1 et QUEUE vaut 0 : la file est vide si TETE = QUEUE + 1, ou plus simplement si TETE > QUEUE.

Procédure Init_File

```
Début
  QUEUE <- 0 ;
  TETE <- 1 ;
Fin ;
```

Fonction File_Pleine : Booléen

```
Début
  Si QUEUE = Borne_Max_Tableau Alors
    File_Pleine <- Vrai ;
  Sinon
    File_Pleine <- Faux ;
  Fsi ;
Fin ;
```

Fonction File_Vide : Booléen

```
Début
  Si TETE > QUEUE Alors
    File_Vide <- Vrai ;
  Sinon
    File_Vide <- Faux ;
  Fsi ;
Fin ;
```

Procédure Enfiler(élément : Type)

```
Début
  Si non(File_Pleine) Alors
    QUEUE <- QUEUE + 1 ;
    File [QUEUE] <- élément ;
  Sinon
    Afficher ("File Pleine") ;
    Quitter () ;
  Fsi ;
Fin ;
```

Fonction Défiler : Type

```
Début
  Si non(File_Vide) Alors
    Défiler <- File [TETE] ;
    TETE <- TETE + 1 ;
  Sinon
    Afficher ("File Vide") ;
    Quitter () ;
  Fsi ;
Fin ;
```

Si nous exécutons manuellement les différentes procédures et fonctions, nous constatons que les deux bornes sont constamment incrémentées. Et je vous assure qu'il n'y a pas d'erreur : en effet, lorsque l'on enfiler un nouvel élément, il est tout-à-fait légitime de voir la QUEUE s'éloigner de la TETE. Il en est de même lorsque l'on défiler un élément : la TETE doit pointer sur la case suivant l'élément défilé. En fait, ce phénomène est tout-à-fait normal. Lorsque nous posons un livre sur une pile, il ne bouge pas. Maintenant, si nous retirons celui qui se

trouve tout en dessous, les autres ne vont pas rester en suspension, ils vont descendre. Le problème ne vient pas de la file mais du tableau. De ce fait si nous n'opérons pas le décalage nous-mêmes, nous risquons d'atteindre très rapidement les limites que nous nous étions fixées pour le tableau. Cette méthode a le seul mérite d'être pédagogique, car elle permet d'aborder les files de manière simple et naturelle, et de soulever quelques problèmes d'implantation. En pratique, elle est à proscrire !

Algorithme Recherche_Occurrence ;
(* version pour les tableaux *)

Constante

```
Tete <- 1 ;
Max_Tableau <- 3 ;
```

Declaration

```
mafile : Tableau [1..3] De Caractere ;
motif : Chaîne [3] ;
Trouve : Booléen ;
Phrase : Tableau [1..100] De Caractere ;
Sert_A_Rien : Caractere ;
Queue, Indice, Defil_NbFois, Defil_Cpt, Position : Entier ;
```

Fonction Cherche : Booléen

```
Début
  Si (mafile = motif) Alors
    Cherche <- Vrai ;
  Sinon
    Cherche <- Faux ;
  Fsi ;
Fin ;
```

Fonction File_Pleine : Booléen

```
Début
  Si (Queue = Max_Tableau) Alors
    File_Pleine <- Vrai ;
  Sinon
    File_Pleine <- Faux ;
  Fsi ;
Fin ;
```

Fonction File_Vide : Booléen

```
Début
  Si (Tete = (Queue + 1)) Alors
    File_Vide <- Vrai ;
  Sinon
    File_Vide <- Faux ;
  Fsi ;
Fin ;
```

Procédure Enfiler (lettre : Caractere)

```
Début
  Si non (File_Pleine) Alors
    Queue <- Queue + 1 ;
    mafile [Queue] <- lettre ;
  Sinon
    Afficher ("File Pleine") ;
    Quitter () ;
  Fsi ;
Fin ;
```

Fonction Défiler : Caractere

Declaration

```
compteur : Entier ;
```

Début

```
Si non (File_Vide) Alors
  Défiler <- mafile [Queue] ;
  compteur <- 1 ;
  Tant Que (compteur < Queue) Faire
    mafile [compteur] <- mafile [compteur + 1] ;
    compteur <- compteur + 1 ;
  Ftq ;
  Queue <- Queue - 1 ;
Sinon
  Afficher ("File Vide") ;
  Quitter () ;
Fsi ;
Fin ;
```

Début

```
(* nous aurions pu aussi saisir cette phrase ainsi que *)
(* le motif à rechercher. Encore un bon petit exercice *)
Phrase = "Only AMIGA makes it possible !!!" ;
motif = "le " ;
```

```
(* ces 2 lignes font office de Procédure Init *)
```

```
Queue = 0 ;
Indice = 1 ;
Tant Que non(File_Pleine) Faire
  Enfiler (Phrase [Indice]) ;
  Indice <- Indice + 1 ;
Ftq ;
Trouve <- Cherche() ;
```

```
Tant Que non(Trouve) Et (Indice < Longueur(Phrase)) Faire
  (* Cette fonction Emplacement a pour but de renvoyer *)
  (* la position de motif[1] dans la file en ignorant *)
  (* le premier élément de celle-ci. Si cet élément *)
  (* n'est pas trouvé la fonction renvoie 0. *)
  Position <- Emplacement (mafile + 1, motif[1]) ;
  Si (Position = 0) Alors
```




```

    Defil_NbFois <- Max_Tableau ;
Sinon
    Defile_NbFois <- Position - 1 ;
Fsi ;

(* pour une fois nous n'utilisons pas les elements que *)
(* nous retourne la fonction defile car nous n'avons nul *)
(* besoin des caracteres qui n'appartiennent pas au motif *)
Defil_Cpt <- 0 ;
Tant_Que (Defil_Cpt < Defil_NbFois) Faire
    Sert_A_Rien <- Defiler () ;
    Defile_Cpt <- Defile_Cpt + 1 ;
Ftq ;

(* et l'on recommence en enfilant de nouveau autant de *)
(* caracteres qu'il en a ete precedemment defiles *)
Tant_Que non(File_Pleine) Faire
    Enfiler (Phrase [Indice]) ;
    Indice <- Indice + 1 ;
Ftq ;
Trouve <- Cherche() ;
Ftq ;
Si (Trouve = Vrai) Alors
    Afficher ("Trouve\n") ;
Sinon
    Afficher ("Non trouve\n") ;
Fsi ;
Fin ;

```

Listing 2 : l'implémentation en C

```

/* Compiler avec LC -L -O */
/* Quelques mots sur le programme qui suit. */
/* Le but est de rechercher une sous-chaine dans une chaine
plus grande. Ici, la longueur de la sous-chaine est 3
et elle contient "le ".
Vous pouvez modifier la sous-chaine a travers la variable
motif (N'OUBLIEZ PAS DE MODIFIER LA CONSTANTE MAX QUI DOIT
ETRE EGALE A LA LONGUEUR DE motif).

L'algorithme est extremement simple comme vous allez pouvoir
le constater et ceci pour deux raisons :
1. Le but de l'article n'etant pas axe sur l'apparition
de motif mais sur la file, il fallait que l'exemple
soit a la portee de chacun (tout en n'etant pas
totalement inutile)
2. La seconde raison est que cet algorithme sort tout
droit de nos cerveaux embrumes.

Pour nous simplifier la tache nous avons pris une file de meme
taille que notre motif.
Algo :
On commence par remplir la file puis tant que la sous-chaine
n'est pas trouvee on regarde dans la file si on trouve la
premiere lettre de la sous-chaine recherchee.
Si cette lettre n'est pas dans la file, on n'a aucune chance
de trouver notre sous-chaine donc on vide la file et on la
remplit de nouveau puis on recommence au debut.
Si cette lettre existe dans notre file, elle est susceptible
d'etre suivie par les autres lettres de notre motif. Pour le
savoir, il suffit de defiler jusqu'a ce que cette lettre se
trouve en premiere position dans notre file et de remplir la
file.
Apres cela, il nous reste a tester si la file contient la
sous-chaine.
Si c'est le cas, c'est GAGNE !!!
Sinon on recommence au debut.
Bon assez parle, voila qui devrait regaler votre compilateur
prefere. Si vous avez des questions, n'hesitez pas nous ferons
de notre mieux pour vous repondre.
*/
#include <stdio.h>
#include <string.h>

#define TETE 0
/* L'indice TETE est une constante initialisee a 0
car en C les tableaux commencent a l'indice 0 */
#define MAX 3 /* On indique ici la taille de notre file */

int cherche (void);
void enfiler (char);
char defiler (void);
int file_pleine (void);
int file_vide (void);

char *mafile = NULL, *motif = "le ";
/* La variable motif contient la sous-chaine a rechercher
Le pointeur mafile contient l'adresse de base de notre file */
int trouve, queue; /* Declaration d'une sentinelle (trouve) nous
indiquant si la sous-chaine a ete trouvee
ou pas, et de la variable queue nous
permettant de savoir ou se trouve la fin
de notre file dans notre tableau */

void main()
{
    char *chaine = "Only AMIGA makes it possible !!!", *courant;
    /* Le pointeur chaine contient l'adresse de base de la chaine
dans laquelle doit s'effectuer la recherche */
    char *adr_a;
    int j, compteur;

    courant = chaine;
    queue = -1; /* Initialisation de notre queue de file */

    /* Allocation de MAX elements pour notre file */
    mafile = (char *) malloc (MAX);

```

```

if (mafile == NULL)
{
    puts ("Probleme d'allocation memoire !!!");
    exit (5);
}

/* Remplissage de notre file */
while (file_pleine() == 0)
{
    enfiler (*courant);
    courant++;
}

trouve = cherche(); /* Test si le contenu de notre file est
le meme que celui de la sous-chaine */
/* Tant que la sous-chaine n'est pas trouvee et que l'on n'a
pas atteint la fin de la chaine */
while ((trouve == 0) && ((courant-chaine) < strlen(chaine)))
{
    /* Recherche dans la file (en ignorant le premier element
sinon si l'occurrence existe bien en premiere position
dans notre file mais que la sous-chaine n'est pas
trouvee ON BOUCLE)
si une occurrence du premier caractere de notre
sous-chaine existe et si c'est le cas, on stocke son
adresse dans adr_a */
    adr_a = strchr (mafile+1,*motif);
    if (adr_a == NULL)
        /* Si adr_a = NULL alors notre file ne contient pas
le premier caractere de notre sous-chaine.
Par consequent, nous pouvons vider la file
completement */
        compteur = MAX;
    else
        /* Si, au contraire, le premier caractere de notre
sous-chaine est trouve dans notre file, il faut
defiler jusqu'a ce caractere */
        compteur = adr_a - mafile;
    for (j = 0; j < compteur; j++)
        defiler();

    /* Remplissage de la file afin de pouvoir tester a
nouveau si la sous-chaine a ete trouvee */
    while (file_pleine() == 0)
    {
        enfiler (*courant);
        courant++;
    }
    trouve = cherche();
}

/* Ce test parle de lui-meme */
printf(trouve == 1 ? "Trouvé\n" : "Non trouvé\n");
free (mafile); /* Soyons fair-play, liberons la memoire */
}

int cherche (void)
{
    /* Si la file contient la sous-chaine recherchee
renvoie 1 Sinon renvoie 0 */
    return((strncmp (mafile,motif,MAX) == 0) ? 1 : 0);
}

int file_pleine (void)
{
    /* Le test doit porter sur MAX - 1
puisque les tableaux commencent en 0 */
    return((queue == (MAX - 1)) ? 1 : 0);
}

/* Si vous desirez plus de details quant aux fonctions ci-dessous
referez vous aux primitives donnees plus haut */
int file_vide (void)
{
    return((TETE == (queue + 1)) ? 1 : 0);
}

void enfiler (char carac)
{
    if (file_pleine() == 0)
    {
        queue++;
        mafile[queue] = carac;
    }
    else
        puts ("File PLEINE");
}

char defiler (void)
{
    char val = '\0';
    int i;

    if (file_vide() == 0)
    {
        val = mafile[queue];
        i = 0;
        while (i < queue)
        {
            mafile[i] = mafile[i + 1];
            i++;
        }
        queue--;
    }
    else
        puts ("File VIDE");

    return (val);
}

```

Par Lucien Turski-Marcon et Bruno Bailluet



TRANSACTION : LE RGB PLASMA

*Eh oui, encore du plasma !
Vous complétez ainsi votre
collection avec ce source qui utilise
pleinement les deux Hercule de
l'Amiga : MM. Copper et Blitter.
Alors au boulot...*

Bien sûr, vous connaissez le plasma, si fréquemment employé dans les démos. Quelques-unes se différencient pourtant par l'utilisation de trois sources de couleurs, notamment le rouge, le vert et le bleu (d'où le terme RGB, d'ailleurs). Ce système permet d'obtenir, en effectuant des opérations logiques entre les sources comme l'opération 'or', des dégradés de couleurs en mouvement. Il est même possible, grâce à un affichage sinusoïdal des différentes sources associées à une déformation verticale, d'afficher les 4096 couleurs de l'Amiga.

LE COPPER

Pour réaliser ce plasma, il suffit d'utiliser un seul registre de couleur, mais il est alors nécessaire de changer cette couleur horizontalement. C'est là qu'intervient le Copper et donc l'initialisation de la fameuse CopperList. Pour chaque ligne de raster, on doit réaliser un 'wait' puis faire suivre cette instruction par une série de 'move' copiant une valeur dans le registre de couleur précédemment choisi (généralement la couleur 0, mais on peut réaliser des effets sympathiques dans des logos, par exemple, en choisissant une autre couleur). Entre chacune de ces copies, l'électron aura parcouru 8 pixels, ce qui donne la précision du plasma. De plus, en variant le masque de la position horizontale du 'wait', c'est-à-dire les 8 bits de poids faible de l'instruction (habituellement \$fe), on peut facilement réaliser une déformation améliorant le rendu final.

Le nombre de données à traiter dans la CopperList, vous l'imaginez, est très important et est même triplé par l'utilisation des trois sources. Voici donc le moment de s'attaquer au second coprocesseur qui se fera une joie de libérer le 68000 d'une tâche aussi ardue.

LE BLITTER

Le Blitter dispose de trois sources (A, B et C) et d'une destination (D) nous permettant, grâce aux Minterms du registre *BLTCON0*, de réaliser différentes opérations logiques entre les trois sources. Il suffit alors que la source A pointe, par exemple, sur un dégradé rouge, B sur un dégradé vert et C sur un dégradé bleu : chacun de ces dégradés utilisant 4 bits indépendants, il est alors possible, en réalisant l'opération 'A or B or C', d'obtenir de superbes variations de couleurs dans l'éventail des 4096 disponibles. Il est nécessaire, pour cela, de recopier verticalement, selon une sinusoïde pré-calculée, les trois sources dans la CopperList. Il y aura donc autant d'accès Blitter que de 'move' par ligne dans la CopperList. La rapidité de ce procédé peut être utilisée pour le changement du masque du 'wait' : il suffit de créer une table de 'wait' que le Blitter utilisera comme source.

Le déplacement des sources de couleurs selon une sinusoïde couplé à la déformation horizontale donnera au plasma une impression de vagues déferlantes. Que ceux sujet au mal de mer s'abstiennent !

Le Blitter permet, il est vrai, de déplacer très rapidement des zones de mémoire, mais il a ses limites. L'écran de travail sera donc restreint à 320 par 256, ce qui obligera quand même le Blitter à déplacer la bagatelle de plus d'une quarantaine de blocs de 600*3 mots. Il reste juste le temps pour une musique et l'affichage de quelques sprites. Il est alors conseillé d'utiliser deux CopperLists pour réaliser l'inévitable double-buffering.

Une option sympa a été ajoutée au source : elle permet, en appuyant sur le bouton droit de la souris, de modifier le Minterm du Blitter, et donc de varier les effets de couleurs.

Listing 1 : RGB_Plasma.s

```
D_init=$8200
D_PriB=$400
D_Btpl=$100
D_Copp=$80
D_Blitt=$40
D_Spts=$20
Dcon =D_init!D_Copp!D_Blitt!D_PriB!D_Btpl

SECTION datas,code_c
debut bsr SaveAll
      lea $dff000,a6
      move #$7fff,$9a(a6)
      move #$7fff,$96(a6)
      clr.l $102(a6)
      clr $180(a6)
      move #0,$10a(a6)

      move.l #ITcopper,$6c.w
      move #$c030,$9a(a6)
      move #Dcon,$96(a6)
      bra Plasma
fPlasma lea $dff000,a6
        bsr RestoreAll
        rts

;-----
SaveAll move.l 4.w,a6
        jsr -132(a6)
        move.l $6c.w,IrqVBL
        move.w $dff01c,INTENA
        or.w #$c000,INTENA
        move.w $dff002,DMACON
        or.w #$8100,DMACON
        rts
RestoreAll
        move.w #$7fff,$dff09a
        move.l IrqVBL,$6c.w
        move.w INTENA(pc),$dff09a
        move.w #$7fff,$dff096
        move.w DMACON(pc),$dff096
        move.l 4.w,a6
        lea GFXlib(pc),a1
        moveq #0,d0
        jsr -552(a6)
        move.l d0,a0
        move.l 38(a0),$dff080
        clr.w $dff088
        move.l d0,a1
        jsr -414(a6)
        jsr -138(a6)
fin moveq #0,d0
    rts
INTENA dc.w 0
DMACON dc.w 0
GFXlib dc.b "graphics.library",0
even

;-----interruptions-----
ITcopper * pour le double-buffering
        move.l #Cop_List2,$80(a6)
        move.l #Pt_cop2+2,ATrois
        btst #0,ecr
        beq.s Lance
        move.l #Cop_List1,$80(a6)
        move.l #Pt_cop1+2,ATrois
Lance clr.w $88(a6) Copperlist à afficher
      jmpVBL jmp 0
      IrqVBL=jumpVBL+2
;-----
ecr dc.w 0
ATrois dc.l Pt_cop1+2

NBmove=47
HAUT=200

Plasma lea $dff000,a6 initialisation
        move.l #-1,$44(a6) du blitter
        clr $42(a6)
        clr.l $60(a6)
        clr.w $64(a6)
        move.w #190,$66(a6)
        move.l #plan,d0Adresse plan1
        lea plan_a,a0 Copper_List1
        lea plan_b,a1 Copper_List2
        move d0,2(a0) mot de poind
        move d0,2(a1) faible
        swap d0
        move d0,6(a0) mot de poind
        move d0,6(a1) fort

Loop
; move #$fff,$180(a6)
; tst.b $06(a6)
; .vsync
; bne.s .vsync
; move #0,$180(a6)
; btst #6,$bfe001 bouton gauche?
```



```

    beq      fPlasma      si oui bye!
    bchg     #0,ecr
    move.l   ATrois(pc),a3  a3=Cop_List1 ou 2

```

**** Change le masque des 'wait' ****

```

    moveq    #0,d6
    move     Z(pc),d6
    addq     #2,d6
    and      #MDecal,d6
    move     d6,Z
    add.l    #Decal,d6
    move     #$9f0,$40(a6)
    move.l   d6,$50(a6)
    move.l   a3,$54(a6)
    move.w   #200*64+1,$58(a6)

```

**** Change le Minterm ****

```

    lea      4(a3),a3
    btst     #10,$dff016      bouton droite
    bne.s    chgMin
    add.b    #1,Minterm
chgMin move   Bltcon0(pc),d6

```

**** Variation des courbes sinusoïdales ****

```

    lea      T1(pc),a4
    move     Va(pc),d3
    addq     #2,d3
    and      #MT1,d3
    move     d3,Va
    move     Vb(pc),d4
    subq     #2,d4
    and      #MT1,d4
    move     d4,Vb
    move     d3,d5

```

**** Copie des sources de couleurs dans la**

**** Copperlist de travail**

```

    move     d6,$40(a6)      BLTCON0
    moveq    #NBmove-1,d7
bcl         #0,d0
    moveq    #0,d1
    moveq    #0,d2
    move     (a4,d3),d0
    addq.w   #4,d3
    and      #MT1,d3
    add.l    #FX_rouge,d0
    move     (a4,d4),d1
    addq.w   #8,d4
    and      #MT1,d4
    add.l    #FX_vert,d1
    move     (a4,d5),d2
    subq.w   #6,d5
    and      #MT1,d5
    add.l    #FX_bleu,d2

    move.l   d0,$48(a6)      BLTC
    move.l   d1,$4c(a6)      BLTB
    move.l   d2,$50(a6)      BLTA
    move.l   a3,$54(a6)      BLTD
    move.w   #HAUT*64+1,$58(a6) BLTSIZE
    lea      4(a3),a3
    dbf     d7,bcl
    bra     Loop

```

;*****

```

Z      dc.w    0
Va     dc.w    0
Vb     dc.w    0
Bltcon0dc.b $0f
Mintermdc.b $fe
even

```

MT1=256*2-1 Modulo de la table de sinus

T1 include df0:rgb_plasma/sin.b

MDecal=64*2-1 Modulo de la table de 'wait'

```

Decal  include df0:rgb_Plasma/wait.b
        include df0:rgb_Plasma/wait.b  4 fois pour
        include df0:rgb_Plasma/wait.b  avoir + de 200
        include df0:rgb_Plasma/wait.b  chiffres
        include df0:rgb_Plasma/wait.b

```

**** CopperLists ****

```

Cop_List1
plan_a dc.w    $e2,0,$e0,0      BPL1
        dc.w    $100,$1100      BPLCON0
        dc.w    $108,-40BPL1MOD
        dc.w    $92,$38         DDFSTRT
        dc.w    $94,$d0         DDFSTOP
        dc.w    $8e,$3781       DIWSTRT
        dc.w    $90,$ffc1       DIWSTOP

```

```

Pt_cop1:
A set $373f
    rept      HAUT
    dc.w      A,$fffe          wait

```

```

        dcb.l   47,$01820000    move
A set A+$100
    endr
    dc.l        $01820000
    dc.l        $fffffffffe

```

Cop_List2

```

plan_b dc.w    $e2,0,$e0,0
        dc.w    $100,$1100,$108,-40
        dc.w    $92,$38,$94,$d0
        dc.w    $8e,$3781,$90,$ffc1

```

Pt_cop2:

```

B set $373f
    rept      HAUT
    dc.w      B,$fffe          wait
    dcb.l     47,$01820000    move
B set B+$100
    endr
    dc.l      $01820000
    dc.l      $fffffffffe

```

**** barres de couleurs ****

FX_rouge ds.w 100

R set 0 rept 15

R set R+\$100 dcb.w 8,R

 endr

 rept 15

R set R-\$100 dcb.w 8,R

 endr

FX_vert ds.w 100

V set 0 rept 15

V set V+\$10 dcb.w 8,V

 endr

 rept 15

V set V-\$10 dcb.w 8,V

 endr

FX_bleu ds.w 100

B set 0 rept 15

B set B+\$1 dcb.w 8,B

 endr

 rept 15

B set B-\$1 dcb.w 8,B

 endr

 ds.w 100

**** 1 plan en une seule ligne grace au**

**** modulo Btpl1 : BPL1MOD**

plan dcb.b 40,\$ff

Listing 2 : MakeWait.bas

```

REM Table des 'wait'
FOR a=0 TO 3.14159*2-3/32 STEP 3.14159*2/64
    b=ABS(INT(7+7*SIN(a))*2)-30
    IF (c AND 15)=0 THEN
        a$a$+CHR$(10)+" dc.w $" +HEX$(b)
    ELSE
        a$a$+"$" +HEX$(b)
    END IF
    c=c+1
NEXT a
OPEN "o",#1,"df0:wait.b" : PRINT #1,a$ : CLOSE #1

```

Listing 3 : MakeSin.bas

```

REM Création table sinus
FOR a=0 TO 3.14159*2 STEP 3.14159*2/256
    b=INT(200+200*SIN(a))
    IF (c AND 31)=0 THEN
        a$a$+CHR$(10)+" dc.w $" +HEX$(b)
    ELSE
        a$a$+"$" +HEX$(b)
    END IF
    c=c+1
NEXT a
OPEN "o",#1,"df0:sin.b" : PRINT #1,a$ : CLOSE #1

```

Par Stéphane Rubinstein

Partout ! Les réducteurs d'octets sont partout ! Que faire, face à une telle invasion ? Le mieux est certainement d'apprendre à les connaître, en vous plongeant par exemple dans l'article qui suit.

Cet article fait suite à celui paru récemment dans Amiga Revue et qui traite de l'utilisation générale de la compression et ne fait que survoler les différents algorithmes. Nous allons donc, dans les mois qui suivent, étudier en détail les différentes méthodes recensées par mes soins. Et en premier lieu, nous commençons doucement avec les algorithmes élémentaires et donc faciles à implémenter. Au programme, compression par topologie, par demi-octet, et suppression des répétitions.

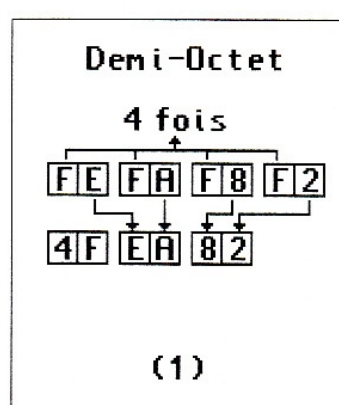
COMPRESSION PAR TOPOLOGIE

Cette méthode consiste à différencier dans un tableau de bits, l'octet le plus courant d'un fichier (en général 0), puis d'en éliminer toutes les occurrences. Par exemple, la séquence originale \$12 \$00 \$80 \$FF \$00 \$00 \$A1 \$00 sera remplacée par la séquence compactée \$12 \$80 \$FF \$A1 suivie du tableau %10110010 (0 désigne le caractère compacté et 1 un caractère non compacté), d'où un gain de 3 octets sur 8.

Cet algorithme, quoique peu rentable, a l'avantage d'être simple et rapide. Il a d'ailleurs été utilisé au temps des micro-dinosaures à faible capacité de traitement.

COMPRESSION PAR DEMI-OCTET

Cet algorithme fut élaboré sur des ordinateurs 8 bits. Il permet d'extraire la partie commune d'une série de N octets (entre autres le demi-octet de poids fort). Celui-ci est alors stocké avec N (nombre codé sur 4 bits) dans un octet, immédiatement suivi par les 4 bits de poids faible de chaque octet ainsi réduit.



Le petit schéma ci-joint mettra les choses au clair. Des résultats ne sont obtenus que pour les données à variation modérée.

Dans la pratique, cette méthode s'utilise couplée à d'autres algorithmes comme le RLE et est alors adaptée au type de l'ordinateur : pour un 16/32 bits, la compression se fait par demi-mots.

LE RUN-LENGTH-ENCODING

Le RLE, plus connu sous le nom de suppression des répétitions, permet de remplacer chaque série d'octets identiques par la longueur de celle-ci et l'octet à répéter. Cet algorithme est largement utilisé lors de la sauvegarde d'images et notamment dans le format IFF. Pour ce type de données, il allie en plus de la vitesse de traitement, la performance.

Prenons une image de 320 de large (40 octets) comportant des zones de même couleur. Deux lignes ne comportant qu'une suite d'octets à zéro peuvent être compressées : les 80 octets de cette zone sont alors remplacés par un compteur (généralement codé sur 8 bits) de valeur la longueur de la série, suivie de l'octet de remplissage (ici, l'on trouve 80 et 0). Par contre la suite 11 12 13 ne peut subir le même traitement. Le compresseur/décompresseur doit alors absolument différencier les deux cas. Une solution simple est de coder le compteur, caractérisant le nombre d'octets compactés ou non, sur 7 bits (valeur < 128) et de garder le bit de poids fort (signe de l'octet) comme identificateur : une valeur négative (bit 7 à 1) indiquera une compression. Cette implémentation est utilisée dans le programme d'aujourd'hui.

LE PROGRAMME

Le format IFF, développé par la société Electronics Arts à la demande de Commodore, peut supporter plusieurs compressions du type RLE. Le programme qui suit traite uniquement de la méthode utilisée dans Deluxe Paint, le ByteRun1, car la plus répandue. Il peut relire une image aux dimensions standard ou overscan dans n'importe quel mode géré par DPaint IV, même le mode HAM entrelacé. De plus, il centrera automatiquement l'image à l'écran. Par contre, pour des raisons évidentes de simplification, l'image ne sera pas lue depuis le disque, mais directement incluse (grâce à la directive *INCBIN* du Devpac) dans l'exécutable.

Si vous voulez réaliser un Slide-Show de vos digitalisations ou de vos superbes réalisations 2D et 3D, n'hésitez pas à reprendre ce programme type.

Le mois prochain, Huffman ou LZW, je ne me suis pas encore décidé. De toute façon, vous aurez droit aux deux, bande de gâtés !

***** Afficheur d'images IFF compressées. Par Emmanuel Hocdet
***** Assemblé avec Genim2 V2.15
***** reconnaît tous les formats de Deluxe Paint IV

```
D_init=$8200
D_PriB=$400
D_BtPl=$100
D_Copp=$80
D_BlIt=$40
D_Spts=$20
D_Disk=$10
Dcon = D_init!D_BtPl

incdir "df0:", "df0:include/" ; A modifier au besoin...

Include "exec/exec_lib.i"
Include "exec/memory.i"

SECTION intro,code
Start bsr SaveAll
move #$7fff,$dff09a * Vide INTENA et DMACON
move #$7fff,$dff096

bsr Main Quel est le programme ?

bottom btst #6,$bfe001 on attend la délicate pression
bne bottom sur le zoli bouton de la souris

move.l Adr_ecr(pc),a1
move.l Taille_ecr(pc),d0
beq.s rien
CALLEXEC FreeMem Désalouons en couleur la mémoire
rien bsr RestoreAll
rts

*****
***** SaveAll & RestorAll
*****
SaveAll move.l 4.w,a6
jsr -132(a6)
lea IrqVBL(pc),a0
move.l $6c.w,(a0)
lea INTENA(pc),a0
move.w $dff01c,(a0)
or.w #$c000,(a0)
lea DMACON(pc),a0
move.w $dff002,(a0)
or.w #$8100,(a0)
rts
RestoreAll
move.w #$7fff,$dff09a
move.l IrqVBL(pc),$6c.w
move.w INTENA(pc),$dff09a
move.w #$7fff,$dff096
move.w DMACON(pc),$dff096
move.l 4.w,a6
lea GFXlib(pc),a1
moveq #0,d0
jsr -552(a6)
move.l d0,a0
move.l 38(a0),$dff080
clr.w $dff088
move.l d0,a1
jsr -414(a6)
jsr -138(a6)
moveq #0,d0
rts
INTENA dc.w 0
DMACON dc.w 0
GFXlib dc.b "graphics.library",0
```



```

even
IrqVBL dc.l 0

*****
***** Main
*****
Main lea $dff180,a3      Debut des couleurs dans a3
      moveq #2,d3        espace entre deux données couleur
      lea picture,a4     Image IFF
      bsr DecrunchIFF    A tout de suite
      bmi FIN

      lea $dff000,a6
      clr.l $102(a6)      * Decalage a 0
      move Nb_Plans(pc),d0 * Mode de l'écran
      ror.w #4,d0
      move Mode_ecr(pc),d1
      and #8804,d1
      or d1,d0
      or #200,d0          Mode couleur
      move d0,$100(a6)    * BPLCON0
      move X_ecr(pc),d3   Préparation pour le calcul
      moveq #17,d4        de la taille de la fenêtre
      move $fff8,d5
      move d1,d0
      and #8000,d0
      beq.s No_Hires
      moveq #9,d4          ...pour le Hires
      move $fffc,d5
      move d3,d0
      lsr.w #1,d0
      move d0,X_ecr

No_Hires
      move Nb_Plans(pc),d0
      move d3,Séparation  ...entre deux bitplans
      clr.l LaceVal
      and #0004,d1
      beq.s No_Lace
      move d3,d1          ...pour le Lace
      mulu d0,d1
      move.l d1,LaceVal   valeur de l'autre serie de bitplans
      add d0,d0
      lea Y_ecr(pc),a0
      move (a0),d1
      lsr.w #1,d1
      move d1,(a0)
      bclr #15,$02a(a6)
      ;
No_Lace subq #1,d0
      mulu d0,d3          * Modulo= Xoctet*(NbPlans-1)
      move d3,$108(a6)    * Modulo plans
      move d3,$10a(a6)
      move Y_ecr(pc),d0   DiwStrtYv=166-Hauteur_Ecran/2
      lsr.w #1,d0
      neg.w d0
      add.w #166,d0
      lsl.w #8,d0
      move X_ecr(pc),d1   DiwStrtXv=289-Largeur_Ecran/2
      lsl.w #2,d1        *8/2 = *4
      neg.w d1
      add.w #289,d1
      or d1,d0
      move Y_ecr(pc),d2   DiwStopYv=Hauteur_Ecran/2-90
      lsr.w #1,d2
      sub.w #90,d2
      lsl.w #8,d2
      move X_ecr(pc),d3   DiwStopXv=Largeur_Ecran/2+33
      lsl.w #2,d3        *8/2 = *4
      add.w #33,d3
      or d3,d2
      sub d4,d1           DdfStrt=(DiwStrtXv-17)/2
      lsr #1,d1
      and d5,d1
      move X_ecr(pc),d3   DdfStop=DdfStrt+(Largeur/16-1)*8
      lsr #1,d3
      subq #1,d3
      lsl #3,d3
      add d1,d3
      move.w d0,$08e(a6)  * Display Window Start
      move.w d2,$090(a6)  * Display Window Stop
      move.w d1,$092(a6)  * Display Data Fetch Start
      move.w d3,$094(a6)  * Display Data Fetch Stop
      move $c020,$09a(a6) * Interruptions (INTENA)
      move #Dcon,$096(a6) * DMACON
      move.l #IT,$6c      Interruption lancée
FIN rts

*****
***** Interruption : Vertical Blank
*****
IT movem.l d0-d2/a0,-(sp)
   move.l Adr_ecr(pc),d1 Adresse de l'Image
   move $dff004,d0      LOF sert pour le mode Lace
   bmi.s Mi             LOF=1 ? oui alors va à Mi
   add.l LaceVal(pc),d1 non
Mi lea $dff0e0,a0        Adresse bitplan1
   moveq #0,d2
   move Séparation(pc),d2 d2=offset entre 2 Btpls

```

```

      move Nb_Plans(pc),d0
      subq #1,d0
loop1 move.l d1,(a0)+      Mise en place de l'image
      add.l d2,d1
      dbf d0,loop1
      move #20,$dff09c    demande d'interruption
      movem.l (sp)+,d0-d2/a0
      rte

***** Décrunchage de l'image IFF
DecrunchIFF
      cmp.l #'FORM',(a4)
      bne erreur
      add.l #8,a4
      cmp.l #'ILBM',(a4)  Image Iff ?
      bne erreur
      add.l #8,a4
      move.l (a4),d5      BMHD long
      move.l 4(a4),X_ecr  vals X et Y de l'écran
      move.b 12(a4),Nb_Plans+1 ; nombre de bitplans
      add.l d5,a4
      add.l #8,a4
      move.l a3,d6        adresse ou mettre les
      beq.s noCMAP        couleurs = 0 ?
      move.l a4,a6        CMAP long
      bsr IFFcols
noCMAP move.l (a4)+,d5
      add.l d5,a4
cherche cmp.l #'CAMG',(a4) modes de l'Amiga
      bne.s recherche
      move 10(a4),Mode_ecr
recherche:
      cmp.l #'BODY',(a4)  debut des données
      lea 2(a4),a4
      bne.s cherche
      move.l 2(a4),d4
      add.l #6,a4
      move X_ecr(pc),d0   calcul de la taille de l'écran
      lsr #3,d0
      move d0,X_ecr
      mulu Y_ecr(pc),d0
      mulu Nb_Plans(pc),d0
      move.l d0,Taille_ecr
      move.l #MEMF_CHIP|MEMF_CLEAR,d1
      CALLEDEC AllocMem  comme son nom l'indique
      move.l d0,Adr_ecr
      beq erreur
      move.l d0,a5
Decrunch:
      moveq #0,d5          * Décompactage
      move.b (a4)+,d5
      bmi.s crunch
      sub.l d5,d4          < 128
recopy move.b (a4)+,(a5)+
      dbf d5,recopy
      bra.s test
crunch neg.b d5           > 128
      move.b (a4)+,d6
      rempli move.b d6,(a5)+
      dbf d5,rempli
test subq.l #2,d4         code + 1er octet = 2
      bhi.s Decrunch
      moveq #0,d7
      rts

IFFcols move.l (a6)+,d7
CMAP moveq #0,d6          * Décode les couleurs
      move.b (a6),d6
      lsl.w #4,d6          3 octets par couleurs (RGB)
      move.b 2(a6),d6
      lsr.b #4,d6
      or.b 1(a6),d6
      move d6,(a3)
      add.l d3,a3
      lea 3(a6),a6
      subq.l #3,d7
      bne.s CMAP
      rts
erreur clr.l Taille_ecr
      moveq #-1,d7
      rts

LaceVal dc.l 0
Séparation dc.w 0
Mode_ecr dc.w 0
Nb_Plans dc.w 0
X_ecr dc.w 0
Y_ecr dc.w 0
Taille_ecr dc.l 0
Adr_ecr dc.l 0

***** Image
picture incbin "1.iff"
end

```

Par Emmanuel Hocdet

**Oui, vous avez bien lu :
Interruptions. Je sais, ça fait le
troisième article sur ce sujet dans
l'ANT, et vous vous attendez
sûrement à du rabachage...**

Eh bien non, car les interruptions ont été traitées seulement sous Exec (cf. ANT 26 et 27). Moi, je vous propose d'aborder les interruptions sans passer par la moindre librairie : on va faire ça à la main comme des grands. De plus, nous allons voir en détail les sources d'interruptions et leur utilisation. Enfin, nous jetterons un coup d'oeil sur les exceptions et les instructions TRAP. Allez, entrons dans le vif du sujet...

LES INTERRUPTIONS HARDWARE

Ces interruptions permettent soit de se synchroniser avec le hardware, soit d'effectuer des actions à des moments bien précis (ni avant, ni après). De plus, l'utilisation d'interruptions permet un gain de temps non négligeable : pendant que l'on attend un événement, on peut effectuer d'autres tâches (qui ne dépendent pas du résultat de cet événement, bien entendu).

Etudions donc les registres de contrôle d'interruptions afin d'en mieux connaître la provenance et l'utilisation.

INTENA \$DFF09A Interruptions autorisées (écriture)
INTENAR \$DFF01C Interruptions autorisées (lecture)

Registres modifiés par l'utilisateur pour spécifier ses choix.

INTREQ \$DFF09C Interruptions demandées (écriture)
INTREQR \$DFF01E Interruptions demandées (lecture)

Registres utilisés par le système pour lancer le processus d'interruption. Egalement modifiés par l'utilisateur une fois l'interruption effectuée (effaçage du bit de déclenchement de l'interruption).

Le format de ces quatre registres (en fait, seulement deux, mais ayant chacun une adresse en lecture et une en écriture) est identique. La description qui suit est donc générale.

Bit 15 : SET/CLR

Lors d'une écriture dans INTENA ou INTREQ, si le bit 15 est mis, tous les autres bits à 1 du mot transféré sont également mis à 1 dans la destination et ceux qui étaient à 0 dans la source resteront inchangés. Si le bit 15 est à 0, tous les bits à 1 de la source seront effacés dans la destination et ceux qui étaient à 0 dans la source resteront inchangés.

Exemples :

Pour effacer les bits 0 à 7 d'INTREQ : `move.w #$00ff,$DFF09C`
Pour placer les bit 0 à 7 d'INTENA : `move.w #$80ff,$DFF09A`

Bit 14 : INTEN (Master Interrupt Enable)

Les interruptions ne peuvent avoir lieu que si ce bit est mis à 1 dans INTENA. En le mettant à 0, vous supprimez toutes les tentatives d'interruption.

Bit 13 : EXTER (IRQ niveau 6)

Ce bit concerne l'interruption externe à travers le CIA-B (voir la note ci-dessous).

Bit 12 : DSKSYN (IRQ niveau 5)

Ce bit concerne l'interruption ayant lieu lorsque la lecture des données d'un lecteur de disquettes est synchronisée avec le contenu du registre DSKSYN (\$DFF07E). Pour que cette interruption soit déclenchée, il faut que la demande de synchronisation ait été faite (bit 9 de ADKCON mis à 1).

Bit 11 : RBF (Receive Buffer Full, IRQ niveau 5)

Ce bit concerne l'interruption pour le port sériel. Elle sera exécutée si le buffer d'entrée (SERDATR, \$DFF018) est plein et peut être lu par l'utilisateur (voir aussi le bit 0).

Bits 10-7 : AUD3-0 (IRQ niveau 4)

Ces quatre bits concernent les interruptions pour les canaux Audio 3 à 0. En mode automatique, elles ont lieu dès que Paula a pris en compte le contenu des registres Audio (que l'on peut alors changer). En mode manuel, elles ont lieu à la fin de chaque transmission d'un mot de donnée sonore (on charge alors le prochain mot). Pour plus de renseignements sur ces interruptions, reportez-vous aux numéros 29 et 30 de l'ANT.

Bit 6 : BLIT (Blitter Finished, IRQ niveau 3)

Ce bit concerne l'interruption ayant lieu dès que le Blitter a fini sa tâche. Notez que le bit 14 de DMACONR (\$DFF002) permet aussi de savoir si le Blitter a fini son travail. L'utilisation de l'une ou l'autre méthode dépend du contexte. Mais, en règle générale, la seconde est plus souvent utilisée, surtout quand le blitter est beaucoup sollicité (sinon, on perd trop de temps lors des interruptions). La première peut rendre de grands services quand deux transferts importants se suivent : le second transfert sera déclenché par interruption dès la fin du premier. Ceci permettra d'effectuer de nombreuses tâches dès l'envoi du premier transfert, sans se soucier du deuxième.

Bit 5 : VERTB (Vertical Blank, IRQ niveau 3)

Ce bit concerne l'interruption de retour du spot de balayage vidéo. Elle a lieu dès que celui-ci est arrivé en bas du moniteur et qu'il s'apprête à remonter.

Bit 4 : COPPER (IRQ niveau 3)

Ce bit concerne l'interruption... Copper (gagné !). C'est l'utilisateur qui décide de la déclencher, à l'aide d'une liste Copper appropriée.

Bit 3 : PORTS (IRQ niveau 2)

Ce bit concerne l'interruption externe à travers le CIA-A (voir la note ci-dessous).

Bit 2 : SOFT (IRQ niveau 1)

Ce bit concerne les interruptions logicielles (voir article sur les interruptions sous Exec du numéro 27).

Bit 1 : DSKBLK (IRQ niveau 1)

Ce bit concerne l'interruption ayant lieu lorsque la transmission des données à travers le canal DMA disque est achevée (fin d'une lecture ou d'une écriture). Cette interruption est le seul moyen de savoir quand le lecteur a fini son travail.

Bit 0 : TBE (Transmit Buffer Empty, IRQ niveau 1)

Ce bit concerne l'interruption pour le port sériel. Elle est déclenchée lorsque le buffer de sortie (SERDAT, \$DFF030) est prêt à recevoir de nouvelles données (voir aussi le bit 11).

Note concernant les interruptions CIA : tout d'abord, notons que les CIAs ont été traités dans les numéros 23, 24, 25 et 28 (l'espion des CIAs, Disques Chouettes et Le Clavier). Vous savez sans doute qu'il existe plusieurs sources d'interruptions pour chaque CIA (A et B) : Timers, clavier, etc. Donc, lorsqu'une interruption de niveau 2 ou 6 survient, il faut être capable de différencier la source de cette interruption au sein du CIA concerné. De plus, il faut pouvoir contrôler les interruptions (autorisation, interdiction) des CIAs. Pour cela, et aussi bizarre que cela puisse paraître, nous ne disposons que d'un registre de contrôle par CIA !

Le fonctionnement est en fait très simple : lors de l'écriture, on fixe les interruptions autorisées et interdites à l'aide d'un masque (même fonctionnement que pour INTENA) et lors de la lecture, on se renseigne sur les interruptions survenues. En gros, c'est comme si nous avions une sorte d'INTENA (écriture) et d'INTREQR (lecture) dans le même registre.



A chaque lecture de ces registres le contenu en est effacé : c'était le seul moyen d'indiquer qu'une interruption avait été traitée puisqu'une écriture n'agit que sur les autorisations d'IT. C'est bien pratique, mais attention à bien traiter toutes les interruptions survenues : il ne peut y avoir qu'une lecture, donc chargez le contenu du registre voulu dans un registre de donnée et effectuez ensuite vos tests, sinon, certaines interruptions risqueraient de ne jamais être traitées.

Un autre problème se pose du fait de la structure de ces registres : comment savoir quelles interruptions sont autorisées avant de les traiter ? A la lecture, le bit 7 permet de savoir si au moins l'une des interruptions survenues est autorisée. Il faudra se contenter de cela, mais dans la plupart des cas ce sera suffisant (il est rare que deux interruptions aient lieu simultanément).

ICR-A \$BFED01 (Interrupt Control Register CIA-A)
ICR-B \$BFED00 (Interrupt Control Register CIA-B)

Attention, seul l'octet de poids faible est utilisé.

Bit 7 : SET/CLR (écriture) - IR (lecture)

En écriture, ce bit a les mêmes fonctions que le bit 15 d'INTENA (voir plus haut).

En lecture, il permet de déterminer quelle interruption est survenue, d'après la table suivante :

- si les bits 0 à 6 sont à 0, IR l'est aussi ;
- si un et un seul des bits 0 à 6 est mis, IR l'est aussi si l'interruption correspondante (pour le CIA s'entend) est autorisée. Sinon, IR est à 0 ;
- si plusieurs des bits 0 à 6 sont mis, IR l'est aussi si au moins une des interruptions correspondantes est autorisée (laquelle ? ben, heu...). Sinon, IR est à 0.

Bit 6 et 5 : inutilisés

Bit 4 : FLG

Interruption par flag.

Bit 3 : SP

Interruption due au port série plein ou vide. Dans le cas du CIA-A, cela signifie qu'une donnée émanant du clavier est arrivée.

Bit 2 : ALRM

Interruption de TOD alarme.

Bit 1 : TB

TIMER B arrivé en dessous de 0.

Bit 0 : TA

TIMER A arrivé en dessous de 0.

INTERRUPTION, ME VOICI !

Je ne reprendrai pas l'explication parfaite accomplie dans le numéro 26 de l'ANT (page 22). Je prends le relais juste au moment où le 68000, une fois passé en mode superviseur, saute au vecteur correspondant au niveau de l'interruption. Rappelons tout de même que ces vecteurs sont situés de \$64 à \$7C pour les interruptions de niveau 1 à 7.

Voyons donc comment se déroule une routine de traitement d'interruption typique.

- 1) Sauvegarde des registres du processeur utilisés dans le programme d'interruption ;
- 2) Vérification d'INTEN afin de déterminer si on doit traiter ou non les interruptions ;
- 3) Vérification de la provenance de l'interruption ;
- 4) Traitement le cas échéant et effacement du drapeau correspondant dans INTREQ (sinon, cette même interruption sera traitée une infinité de fois) ;
- 5) Restauration des registres sauvegardés en 1) ;
- 6) Retour au mode utilisateur par RTE.

Le processus est donc très simple : vous initialisez correctement INTENA pour autoriser votre interruption et vous précisez l'adresse de votre routine dans le tableau de vecteurs en ayant bien soin de sauvegarder l'ancienne adresse afin de la replacer une fois le travail accompli. Comme vous le voyez sûrement, cette méthode n'est pas vraiment compatible avec le multitâche (NDLR : quel doux euphémisme...), puisque vous n'effectuez que votre propre interruption...

Voici maintenant une méthode permettant, dans certains cas, d'éviter la perte de temps causée par l'exécution d'une interruption et de simplifier les choses : primo, ne pas autoriser les interruptions ; secundo, faire une boucle d'attente testant INTREQ ; tertio, vider INTREQ. Cela est intéressant dans des cas d'attente du lecteur de disquettes, d'un timer, etc...

EXEMPLES D'APPLICATION

Les routines suivantes ne sont bien sûr pas directement exécutables, mais vous donneront un ordre d'idée de ce qu'il faut faire : c'est à mon avis le principal (un long source est moins intéressant). Bref, il s'agit d'une synchronisation toute bête avec l'écran.

Première méthode : Interruption simple

```
START:
... (peut-être sauvegarde d'INTENA) ...
move.l $74, SAUVE_IT; On sauve l'ancienne IT
move.l #MON_IT, $74; Voilà la notre
move.w #$C010, $DFF09A; Autorisation d'IT
....
move.l SAUVE_IT, $74; Restaure l'ancienne IT
rts

MON_IT:
movem.l d0-d7/a0-a6, -(a7); Sauvegarde registres
move.w $dff01c, d0; INTENAR
btst #5, d0; Test d'INTEN
beq.s FIN_INT
and.w $dff01e, d0; INTREQ
btst #4, d0; d'où viens tu ?
beq.s FIN_INT; et es-tu autorisée ?

.... traitement d'IT ...
move.w #$0010, $DFF09C; Interruption traitée
FIN_INT:
movem.l (a7)+, d0-d7/a0-a6; Restaure registres
rte

SAUVE_IT: dc.l 0
```

Deuxième méthode : Interruption avec rebranchement sur l'ancienne.

```
START:
... (peut-être sauvegarde d'INTENA) ...
move.l $74, SAUTE+2; On sauve l'ancienne IT
move.l #MON_IT, $74; Voilà la notre
move.w #$C010, $DFF09A; Autorisation d'IT
....
move.l SAUTE+2, $74
rts

MON_INT:
movem.l d0-d7/a0-a6, -(a7); Sauvegarde registres
move.w $dff01c, d0; INTENAR
btst #5, d0; Test d'INTEN
beq.s FIN_INT
and.w $dff01e, d0; INTREQ
btst #4, d0; d'où viens tu ?
beq.s FIN_INT; et es-tu autorisée ?
.... traitement d'IT ...
move.w #$0010, $DFF09C
FIN_INT:
movem.l (a7)+, d0-d7/a0-a6; Restaure registres
SAUTE:
jmp SAUTE; Saut à l'ancienne IT
```

Troisième méthode : Interruption ???

```
START:
....
move.w $DFF01c, d0; INTENAR
or.w #$8000, d0; Bit 15 = 1
move.w d0, SAUVE_INTENA; Sauve INTENA
move.w #$7fff, $DFF09A; Interdit tout
WAIT_IT:
btst #4, $DFF01e; Attend demande d'IT
beq.s WAIT_IT
move.w #$0010, $DFF09C; Supprime cette demande
... programme synchronisé avec l'écran ...
... Retour à Wait_IT ...
move.w SAUVE_INTENA, $DFF09A
rts

SAUVE_INTENA:
dc.w 0
```

Par Philippe Rivailon

HARDWARE : 68030 : MOTOROLA NOUS

**Mettez un turbo dans votre Amiga !
Non, ce n'est pas là une pub pour
une quelconque carte accélératrice,
ni une boutade destinée à faire rire
les heureux possesseurs d'A3000...**

Enfin un domaine technologique où les japonais ne peuvent inquiéter le marché mondial. Pourtant, la mise en oeuvre de ces merveilles de la miniaturisation que sont les microprocesseurs est un véritable travail de fourmi (ah, ah !). De l'heureux père de la famille des 68xxx, Monsieur Motorola, vous connaissez déjà le 68000 et peut-être même les 68008, 68010 et 68012. Si ce n'est pas le cas, reportez-vous vite aux premiers Amiga NewsTech ou par exemple, à *Mise en oeuvre du 68000* (éditions Sybex).

Mais depuis quelques années, la famille s'est agrandie avec les microprocesseurs 32 bits de la seconde génération : les 68020 et 030, avec leurs coprocesseurs arithmétiques 68881 et 882. Sans oublier bien sûr le petit dernier (et premier de la troisième génération) : le 68040. La compatibilité étant ascendante, nous n'étudierons que les améliorations des différents microprocesseurs par rapport au 68000 de base. De plus, puisque le 68030 devient un standard sur Amiga, cet article lui sera dédié et sera suivi d'une série traitant, entre autres, des caches mémoires, du PMMU et des coprocesseurs arithmétiques.

PRESENTATION

Le nom du premier microprocesseur de la série vient, pour la petite histoire, du nombre de transistors le composant, c'est-à-dire 68000. Le 68020 quant à lui, en comporte 200 000 et a été développé en technologie HCMOS, permettant de regrouper vitesse de travail, faible consommation et optimisation des fonctions intégrées. Il opère entre 2 et 3 MIPS (Millions d'Instructions Par Seconde), contre 0.5 pour le 68000 et possède un cache d'instructions de 256 octets. Le 68030 regroupe l'équivalent d'un 68020 et de trois caches fonctionnant en parallèle : une unité de gestion mémoire par pagination (PMMU) et deux caches de 256 octets chacun, l'un de données et l'autre d'adresses. Enfin, le 68040 fait encore plus fort puisqu'il intègre, en plus d'une unité de traitement 68030 à 13.5 MIPS, d'un cache données et d'un cache d'instructions de 4 Ko chacun et d'une PMMU améliorée, une unité de calcul en virgule flottante du type 68881/68882. Un must pour les logiciels d'imagerie.

En clair, le 68030 dispose des caractéristiques suivantes :

- compatibilité avec ses prédécesseurs, du 68000 au 68020 ;
- fréquence de travail allant de 20 à 50 MHz ;
- bus de données et d'adresses 32 bits, avec un adressage atteignant 4 Go (giga-octets) ;
- nouveaux registres de contrôle ;
- nouveaux modes d'adressage (18 en tout) ;
- cache de données et d'adresses indépendants ;
- jeu d'instructions étendu (identique à celui du 68020, sans les instructions de gestion de modules) ;
- dialogue possible avec le 68881 et le 68882 ;
- échanges asynchrone et synchrone avec l'extérieur.

Les caches fonctionnent par défaut en transparent et ne sont donc pas un obstacle, dans un premier temps, à la compréhension de la programmation du microprocesseur. Je les garde en réserve, avec les modes spéciaux du 68030, dont les exceptions et le dialogue avec les coprocesseurs arithmétiques, pour un prochain article.

LES CYCLES DE BUS

La première chose que j'ai faite en ouvrant des ouvrages traitant du 68030 a été de chercher la table des cycles des instructions, afin de comparer avec celle du 68000. Mais à ma plus grande surprise, je n'ai rien trouvé... En effet, on peut comprendre cette absence par la difficulté de faire correspondre à chaque instruction son temps d'exécution en fonction des multiples modes du 68030 : les accès mémoire nécessaires peuvent aussi bien se faire en mode cache (très rapide) qu'en mode sta-

tique, ou même les deux ; les échanges avec l'extérieur varient, de plus, d'entre 1 et 3 cycles ; et il n'y a pas moins de 18 modes d'adressage. Je peux simplement vous dire que le nombre de cycles utilisés par le processeur lors des accès mémoire ou l'exécution d'une instruction a été amélioré par rapport au 68000. Les multiplications et les divisions ne prennent plus qu'une poignée de cycles, et les instructions à adressage direct, de 1 à 2 cycles.

La plus impressionnante amélioration concerne sans aucun doute les instructions de décalage, du type *LSL*, *ROR*, etc., qui s'effectuent en un seul cycle, et cela quelque soit le nombre de décalages demandés, grâce à l'implantation d'un système hardware de décaleur à barillet. L'instruction *CLR* a également été modifiée (en fait, depuis le 68010) : elle efface directement son opérande, sans cycle de lecture préliminaire, comme sur les processeurs précédents.

Le principe des échanges synchrones et asynchrones entre le processeur et l'extérieur est une notion fondamentale des processeurs 32 bits. Il permet une compatibilité totale des échanges, même avec des circuits externes travaillant sur 16 ou 8 bits et facilite grandement la tâche des programmeurs.

Le 68030 travaille essentiellement en 32 bits, à la manière du 68000 en 16 bits, et permet en mode synchrone des transferts de 32 bits en un seul cycle de bus. Dans ce mode, la donnée à traiter doit se trouver à une adresse multiple de 4 (elle est alors dite alignée). Le 68000, lui, avait besoin que chaque donnée 16 bits soit située à une adresse paire. Le mode rafale (burst en anglais) peut, ici, être utilisé pour des transferts 32 bits très rapides.

Dans le cas où la donnée n'est pas alignée, le 68030 évoluera en mode asynchrone : il se débrouillera alors tout seul pour effectuer le transfert (en plusieurs cycles de bus, tout de même). En comparaison avec le 68000, cela revient par exemple, à vouloir lire un mot à partir d'une adresse impaire, ce qui déclenche irrémédiablement un erreur de bus. Au contraire, le 68030 est capable de gérer la donnée. En regardant le tableau des cycles mémoire, vous remarquerez pourquoi les Rams 32 bits augmentent la vitesse de traitement des données. Attention cependant, les instructions doivent obligatoirement se situer à des adresses paires.

REGISTRES

En mode utilisateur, on retrouve tous les registres du 68000, à savoir 8 registres de données 32 bits D0 à D7, 8 registres d'adresses 32 bits A0 à A7 dont un pointeur de pile et le CCR (*Condition Codes Register*, soit en français, Registre des Codes de Condition). Par contre, le mode superviseur a subi de nombreux changements.

Le registre d'état

Le registre SR a été modifié dès la deuxième génération de processeurs : en plus des 5 bits de flags (X, N, Z, V et C), des 3 bits de masquage de priorité des interruptions (I2, I1 et I0), du bit de trace (T) et du bit d'état utilisateur/superviseur (S), le registre d'état supporte 3 nouveaux bits.

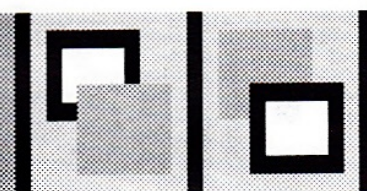
SR	T1	T0	S	M	0	I2	I1	I0	0	0	0	X	N	Z	V	C
bits	15							8	7							0

- T1 et T0 sont les 2 bits d'extension de T en 68000/68010, où T0 n'existe pas. En plus du mode trace normal I0, la combinaison 01 permet de générer une trace uniquement lors d'un branchement (*BRA*, *JMP*, etc). 00 correspond au mode sans trace (encore une nouvelle lessive !) et I1 est réservé.

- M est le bit Maître. Il permet de distinguer deux nouveaux modes : le mode interruption (M=0) et le mode maître (M=1). Si le processeur est en mode superviseur, le système a alors accès à deux nouveaux pointeurs de pile, en plus de USP et SSP : *MSP* et *ISP*.

Le *MSP (Master Stack Pointer)* est le pointeur de pile système, sélectionné lorsque S=1 et M=1.

Le *ISP (Interrupt Stack Pointer)* est un pointeur de pile particulier, utilisé lors des interruptions. Il est sélectionné lorsque S=1 et M=0.



Les registres de contrôle

Le VBR (*Vector Base Register*), présent depuis le 68010, est le registre de base des vecteurs. Il permet de changer l'adresse du début de la table des vecteurs d'exception. Ainsi, le programmeur peut définir sa propre table sans modifier celle d'origine, située en \$0. Ceci est plus particulièrement utile pour les systèmes d'exploitation multitâches (à noter cependant que l'Amiga n'utilise absolument pas cette facilité).

Le SFC (*Source Function Codes*) et le DFC (*Destination Function Codes*), tous deux également présents depuis le 68010, sont les registres des codes fonctions source et destination. Seuls les trois bits de poids faible, nommés FC2, FC1 et FC0, de chaque registre sont utilisés. Ils sont modifiables par l'instruction *MOVEC* et n'ont de rôle qu'avec l'instruction *MOVES*.

Le CACR (*Cache Control Register*) et le CAAR (*Cache Address Register*) servent à la configuration du cache. Vous vous reporterez à l'article traitant des caches pour en savoir plus.

Idem pour les 6 registres *CRP*, *SRP*, *TC*, *TT0*, *TT1* et *MMUSR*, apparus dans le 68030, qui servent à configurer la PMMU.

LES MODES D'ADRESSAGE

Le 68020 et le 68030 possèdent un nombre important de modes d'adressage. L'on y retrouve tous les modes du 68000 en plus complets (32 bits obligent) ainsi que de nouveaux modes, de type adressage indirect par mémoire. Il ne fait aucun doute que le développement de tels modes a été motivé par l'utilisation fréquente, dans les langages évolués, d'indirections à plusieurs niveaux.

Un petit récapitulatif de tous ces modes ne fera certainement pas de mal. Mettons juste au point toutes les annotations utilisées dans le tableau ci-dessous : Xn désigne un registre quelconque, avec X=D pour un registre de données et X=A pour un registre d'adresses ; les déplacements bd et od sont deux valeurs signées codées suivant le nombre de bits b et o (16 ou 32) ; E correspond à un entier de valeur 1, 2, 4 ou 8 ; enfin, T désigne une taille en W (mot) ou L (long mot). Pour T=W, le registre est étendu à 32 bits. Les modes d'adressage spécifiques au 68020 et 68030 seront signalés par un astérisque (*).

Direct par registre de donnée : Dn

Exemple : *ADDX.W D0,D0*

Description : Triviale (ADDX est plus rapide que ROXL #1).

Direct par registre d'adresse : An

Exemple : *MOVEA.L D0,A0*

Description : No comment !

Indirect (par registre d'adresse) : (An)

Exemple : *CLR.W (A0)*

Description : Le registre contient l'adresse de l'opérande à traiter.

Indirect post-incrémenté : (An)+

Exemple : *CMPM.B (A0)+,(A1)+*

Description : Idem, mais une fois l'opération effectuée, le registre est incrémenté de 1, 2 ou 4 suivant la taille de l'instruction.

Indirect pré-décrémenté : -(An)

Exemple : *MOVEM.L D0-D7/A0-A6,-(A7)*

Description : Le registre d'adresse est d'abord décrémenté de 1, 2 ou 4 suivant la taille de l'instruction, puis sa nouvelle valeur est utilisée comme pointeur sur l'opérande.

Indirect avec déplacement 16 bits : (d16,An) ou d16(An) en 68000

Exemple : *OR.L D0,(1024,A0)* ou *OR.L D0,1024(A0)*

Description : La valeur du registre est additionnée à la donnée 16 bits. Le résultat pointe sur l'opérande.

(*) Indirect indexé avec déplacement 8 bits : (d8,An,Xn.T*E)

Exemple : *TST.L (-4,A1,D1.W*4)*

Description : La valeur du registre Xn est multipliée par E puis additionnée au contenu de An et au déplacement 8 bits (d8 + An + Xn*E). Le résultat est un pointeur sur l'opérande à traiter.

(*) Indirect indexé avec déplacement 16 ou 32 bits : (bd,An,Xn.T*E)

Exemple : *JMP (\$FACEFE,A1,A0.L*8)*

Description : Idem que précédemment, si ce n'est que le déplacement est codé sur 16 ou 32 bits au lieu de 8.

(*) Indirect par mémoire post-indexé : ([bd,An],Xn.T*E,od)

Exemple : *MOVE.L D6,([\$BEBE,A0],D1.W*2,\$AFECACA)*

Description : Le long mot pointé par l'adresse An + bd est additionné au déplacement od et au registre Xn multiplié par E. Le résultat est un pointeur sur l'opérande à traiter.

(*) Indirect par mémoire pré-indexé : ([bd,An,Xn.T*E],od)

Exemple : *AND.L D4,([\$BAC,A1],\$CFE)*

Description : Le résultat de l'opération An + Xn*E + bd est un pointeur sur un long mot. Ce dernier, additionné au déplacement od, est lui-même un pointeur sur l'opérande. Encore une double indirection !

Relatif au PC avec déplacement 16 bits : (d16,PC) ou d16(PC) en 68000

Exemple : *MOVE.W (\$BAFF,PC),D7* ou *MOVE.W \$BAFF(PC),D7*

Description : L'adresse de l'opérande est le résultat de la somme effectuée entre le compteur programme et le déplacement 16 bits d16.

(*) Relatif indexé avec déplacement 8 bits : (d8,PC,Xn.T*E)

Exemple : *MULS.L (7,PC,D7.L*8),D2*

Description : Le résultat de l'opération d8 + PC + Xn*E est un pointeur sur l'opérande.

(*) Relatif indexé avec déplacement 16 ou 32 bits : (bd,PC,Xn.T*E)

Exemple : *DIVUL.L (\$FAC,PC,A2.W*2),D0:D1*

Description : Idem que ci-dessus, si ce n'est que le déplacement bd est codé sur 16 ou 32 bits au lieu de 8.

(*) Relatif indirect par mémoire post-indexé : ([bd,PC],Xn.T*E,od)

Exemple : *BFEXTU.L ([\$FEE,PC],D2.L*4,\$FFF),{5:24},D0*

Description : La somme PC + bd est un pointeur sur un long mot qui, additionné à Xn*E et le déplacement od, représente l'adresse de l'opérande.

(*) Relatif indirect par mémoire pré-indexé : ([bd,PC,Xn.T*E],od)

Exemple : *CMPA.W ([\$CAFF,PC,D0.L*11],A0)*

Description : Le résultat de l'addition entre le long mot pointé par PC + Xn*E + bd, et le déplacement od, est un pointeur sur l'opérande.

Absolu court : adresse.W

Exemple : *MOVEA.L 4.W,A6*

Description : Le signe de la donnée 16 bits est étendu à 32 bits et représente une adresse mémoire.

Absolu long : adresse.L

Exemple : *CLR.L \$DFF180*

Description : La donnée représente directement une adresse mémoire.

Immédiat : #valeur

Exemple : *RTD #\$CABA*

Description : la donnée est la valeur à traiter.

La double indirection s'avère très utile pour l'utilisation de tableaux de pointeurs. Par exemple, je désire connaître la couleur du cinquième élément (d0=5-1) du tableau deux, avec A0 pointeur sur le premier pointeur du tableau. En 68000, cela donne :

```
MOVE.L 4(a0),a1      ; a1=pointeur deuxième table
MOVE.L d0,d1         ; sauvegarde de d0
LSL.L #3,d1          ; *E
MOVE.W 6(a1,d1),d1    ; d1=élément recherché
```

Une instruction et un seul registre d'adresse pour la version 68030 :

```
MOVE.W ([4,a0],d0.L*8,6),d1
```

Ce résultat peut aussi être remarqué lors de la compilation de programmes C ou autres langages évolués, d'où une optimisation du code non négligeable.

Par Emmanuel Hocdet

TOOLBOX : YACC (II)

Nous continuons notre étude de Yacc avec d'une part, la synthèse du programme du mois dernier et sa compilation, et d'autre part, son interfacement avec Flex, l'analyseur lexical.

Comme je vous l'ai signalé le mois dernier, Yacc est disponible dans le Domaine Public, et plus particulièrement sur la disquette Fred Fish numéro 419. Vous trouverez outre l'exécutable du programme, les sources compactés avec LhArc, ainsi qu'un fichier documentation des plus succints, puisqu'il se borne à décrire la syntaxe d'utilisation de la commande. En prenant la peine de décompacter ces sources, vous trouverez un répertoire Test qui contient le source Yacc complet de la commande ftp (pour File Transfert Protocol), commande permettant la gestion des transferts de données dans un réseau.

L'installation de Yacc est des plus simples, puisque seul l'exécutable est nécessaire au bon fonctionnement du produit. Il vous suffit donc de posséder Yacc dans votre répertoire courant ou dans un répertoire accessible soit directement (comme c:) soit à l'aide d'un chemin d'accès réalisé grâce à la commande Path.

COMPILATION DE L'EXEMPLE

Première chose à faire, récapituler l'exemple du mois dernier disséminé tout le long de l'article, suite aux explications le concernant.

Le programme Yacc est donc le suivant :

```
%token VERBE NOM ADJECTIF NOMBRE
%%

Phrase : Sujet Predicat '\n' { printf(" Phrase déclarative \n");}
      | Predicat '\n' { printf(" Phrase impérative \n");}
      | error { printf("Erreur de Syntaxe \n");}
      ;
Predicat : VERBE Complement { printf(" Prédicat \n");}
      ;
Suje : Objet { printf(" Sujet \n");}
      ;
Complement : Objet { printf(" Complément \n");}
      ;
Objet : NOM
      | NOMBRE NOM
      | ADJECTIF NOM
      | NOMBRE ADJECTIF NOM
      ;
%%

main()
{
    printf("? ");
    yyparse();
}

yyerror()
{
}

yywrap()
{
}

yylex()
{
}
```

Rappelons que les symboles %% séparent les trois sections du programme (déclarations, règles, fonctions utilisateur). Je vous conseille de laisser une ligne blanche après le %% séparant les règles des fonctions utilisateur : Yacc a en effet une facheuse tendance à ne pas reconnaître ce symbole lors de la compilation.

La compilation Yacc du programme, une fois celui-ci tapé et sauvegardé avec une extension ".y", s'effectue sans encombre. La commande est la suivante : *yacc exemple.y*.

Yacc génère alors un fichier dénommé *y.tab.c* d'environ 8 Ko, représentant le programme que vous avez décrit en quelques lignes (ce programme ne sera pas listé ici mais je vous conseille de l'étudier par vous même).

Dès lors, il ne reste plus qu'à compiler ce fichier source C à l'aide de votre compilateur favori, par exemple le SAS/Lattice C, en tapant *lc -Lc y.tab.c*. L'exécutable généré, à savoir *y.tab*, peut être lancé dès ce stade. Mais comme vous pourrez le remarquer, les résultats obtenus sont invariablement "Erreur de Syntaxe". C'est tout-à-fait normal, puisque la partie d'analyse lexicale n'a pas encore été réalisée.

COMMENT YACC FONCTIONNE T-IL ?

Pour réaliser cette étude, nous allons utiliser l'option -v (pour verbose) qui va générer un fichier *y.output* contenant le fichier descripteur de la logique du programme avant sa transcription en C. Il suffit de taper *yacc -v exemple.y*

Le fichier résultant est le suivant :

```
0 $accept : Phrase $end

1 Phrase : Sujet Predicat '\n'
2         | Predicat '\n'
3         | error

4 Predicat : VERBE Complement

5 Sujet : Objet

6 Complement : Objet

7 Objet : NOM
8         | NOMBRE NOM
9         | ADJECTIF NOM
10        | NOMBRE ADJECTIF NOM

state 0
    $accept : . Phrase $end (0)

    error shift 1
    VERBE shift 2
    NOM shift 3
    ADJECTIF shift 4
    NOMBRE shift 5
    . error

    Phrase goto 6
    Sujet goto 7
    Predicat goto 8
    Objet goto 9

state 1
    Phrase : error . (3)

    . reduce 3

state 2
    Predicat : VERBE . Complement (4)

    NOM shift 3
    ADJECTIF shift 4
    NOMBRE shift 5
    . error

    Complement goto 10
    Objet goto 11

state 3
    Objet : NOM . (7)

    . reduce 7

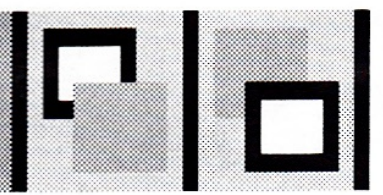
state 4
    Objet : ADJECTIF . NOM (9)

    NOM shift 12
    . error

state 5
    Objet : NOMBRE . NOM (8)
    Objet : NOMBRE . ADJECTIF NOM (10)

    NOM shift 13
    ADJECTIF shift 14
    . error

state 6
    $accept : Phrase . $end (0)
```

```

$end accept
state 7
  Phrase : Sujet . Predicat '\n' (1)
  VERBE shift 2
  . error
  Predicat goto 15
state 8
  Phrase : Predicat . '\n' (2)
  '\n' shift 16
  . error
state 9
  Sujet : Objet . (5)
  . reduce 5
state 10
  Predicat : VERBE Complement . (4)
  . reduce 4
state 11
  Complement : Objet . (6)
  . reduce 6
state 12
  Objet : ADJECTIF NOM . (9)
  . reduce 9
state 13
  Objet : NOMBRE NOM . (8)
  . reduce 8
state 14
  Objet : NOMBRE ADJECTIF . NOM (10)
  NOM shift 17
  . error
state 15
  Phrase : Sujet Predicat . '\n' (1)
  '\n' shift 18
  . error
state 16
  Phrase : Predicat '\n' . (2)
  . reduce 2
state 17
  Objet : NOMBRE ADJECTIF NOM . (10)
  . reduce 10
state 18
  Phrase : Sujet Predicat '\n' . (1)
  . reduce 1
7 terminals, 6 nonterminals
11 grammar rules, 19 states

```

Ce fichier est la description de ce que l'on appelle une "machine à états finis" comportant 19 états. Une machine à états finis est un programme pouvant prendre un nombre fini d'états possibles, en fonction des données d'entrées. Un tel programme ne peut donc jamais se retrouver dans un état imprévisible qui le conduirait à un défaut de fonctionnement. Dans le cas d'analyseurs syntaxiques, l'intérêt est évident puisque l'on ne sait jamais quels peuvent être les réponses de l'utilisateur.

Si nous étudions ce listing en détail, nous nous apercevons que les règles que nous avons définies dans le source Yacc sont toutes reprises dans cet état sous la forme d'un numéro entre parenthèses. Par exemple :

(1) -> Sujet Predicat

est bien la règle numéro 1.

Chaque état de l'analyseur est défini en terme de progrès par rapport à l'état précédent. A chaque fois que l'analyseur sémantique reçoit un

symbole de l'analyseur lexical, il essaye de le faire coïncider avec un symbole de la partie droite de la définition de chaque état.

Lorsque il trouve une correspondance dans une règle, l'analyseur sémantique doit prendre une décision :

- Si il est en mesure, à cet instant, de reconnaître toute l'expression de droite deux cas peuvent se présenter : soit le symbole de gauche est non terminal, et dans ce cas, il réduit cette expression dans l'état de la machine indiqué par la commande *reduce* (il doit alors attendre des informations supplémentaires pour conclure), soit le symbole de gauche est terminal, et dans ce cas, l'analyseur a atteint un état terminal de reconnaissance et est en mesure de donner les résultats associés. C'est-à-dire qu'il va déclencher la fonction C associée à l'état indiqué par la commande *accept*.
- Sinon, il empile cette correspondance et se rend dans l'état suivant indiqué par la commande *shift*.
- Dans tous les autres cas il y a génération d'erreur (commande *error*).

Suivons ce mécanisme pas à pas sur un exemple, à savoir la dorénavant célèbre phrase "Stéphane range trois disquettes vertes n" (ce qui est un exploit de la part de Stéphane). (NDLR : Ben oui, je n'ai que des disquettes rouges). Cette phrase sera convertie par tout bon analyseur syntaxique en :

NOM VERBE NOMBRE ADJECTIF NOM \$end.

Nous connaissons tous ces symboles, à l'exception de \$end qui indique tout simplement la fin de la phrase. L'analyseur syntaxique débute avec cette phrase dans un état (-1,0). Le premier chiffre correspond au symbole reconnu (aucun pour le moment) et le deuxième, à l'état en cours (ici 0).

Pile : (-1,0)

Le premier symbole en entrée est *NOM*. Associé à l'état 0, il déclenche un *shift* vers l'état 3. Le couplet (*NOM,3*) est empilé et l'analyseur pointe maintenant sur l'état 3.

Pile : (-1,0), (*NOM,3*)

Le symbole suivant, *VERBE*, n'est pas reconnu. Par contre, cet état contient une expression de droite parfaitement connue, ce qui déclenche une opération *reduce*. Le couplet (*NOM,3*) est dépilé, *NUM* est remplacé par sa valeur réduite (à savoir un *Objet*) et l'analyseur est renvoyé à l'état 0. Dans cet état, le terme *Objet* est reconnu comme l'état 9. On empile donc (*Objet,9*).

Pile : (-1,0), (*Objet,9*)

Dans l'état 9, le symbole *VERBE* n'est pas reconnu. Par contre, *Objet* déclenche une opération de réduction en *Sujet*. On dépile (*Objet,9*), on remplace *Objet* par *Sujet* et on retourne en 0. Là, *Sujet* nous renvoie en 7. On empile donc (*Sujet,7*).

Pile : (-1,0), (*Sujet,7*)

Dans l'état 7, *VERBE* est reconnu. On va donc en 2, où l'on empile (*VERBE,2*).

Pile : (-1,0), (*Sujet,7*), (*VERBE,2*)

NOMBRE est reconnu et nous amène dans l'état 5.

Pile : (-1,0), (*Sujet,7*), (*VERBE,2*), (*NOMBRE,5*)

Dans cet état, *ADJECTIF* est reconnu, le pointeur est mis sur l'état 14.

Pile : (-1,0), (*Sujet,7*), (*VERBE,2*), (*NOMBRE,5*), (*ADJECTIF,14*)

et l'état 14 reconnaît *NOM* et déclenche l'état 17.

Pile : (-1,0), (*Sujet,7*), (*VERBE,2*), (*NOMBRE,5*), (*ADJECTIF,14*), (*NOM,17*)

(suite page 28)

SUBWAY : C DU SON (I)

Ayant achevé notre série d'articles sur l'animation, nous allons entamer à partir de ce numéro un nouveau thème : la génération de sons via le système, en langage C.

Sans vouloir reprendre l'excellent article "La voix de Paula" (cf. ANT numéros 29 et 30), un bref rappel de la configuration audio hardware de l'Amiga n'est sans doute pas superflu. Paula (le chip chargé, entre autres choses, de la reproduction sonore) est composée de quatre canaux audio indépendants, regroupés en deux voix stéréo : les canaux 0 et 3 sont connectés à la sortie gauche, alors que les canaux 1 et 2 sont reliés à la sortie droite. Ces sorties, je vous le rappelle, sont les deux prises Cinch situées à l'arrière de la machine.

Chaque canal possède un certain nombre de registres de contrôle permettant de piloter la sortie audio :

- *Volume Control Register* : ce registre permet d'assigner à chaque canal un volume de sortie, variant entre 0 et 64.
- *Period Register* : utilisé pour définir la fréquence d'échantillonnage sonore du canal. Comme il s'agit d'une période et non d'une fréquence, plus la valeur inscrite dans ce registre est élevée, plus la fréquence de sortie sera faible ($\text{Fréquence} = 2 * \text{PI} / \text{Période}$) et par la même, plus le son sera grave.
- *Data Register* : le registre de données pointe sur la zone mémoire où sont stockées les données pour ce canal. Ces informations doivent impérativement être stockées en mémoire CHIP.
- *Length Register* : indique la taille de la zone mémoire où sont stockées les données associées au canal.

Chaque canal audio peut retrouver ses données, soit grâce au 68000, soit à l'aide de quatre canaux DMA (Direct Memory Access). En utilisant les canaux DMA, Le système audio peut charger lui-même ses données depuis la CHIP RAM, sans avoir recours au microprocesseur, qui peut alors être utilisé pour d'autres tâches (comme calculer la prochaine onde sonore, par exemple).

Pour produire un son, il faut réserver une zone de mémoire dans laquelle on va stocker une représentation numérique de la forme de l'onde sonore. Si l'on utilise exclusivement le composant hardware, il faudra, après avoir rempli cette zone, positionner le volume au niveau désiré, établir la période du son, indiquer au registre de données où se trouve les informations, inscrire la longueur dans le Length Register et activer le DMA.

Mais cette méthode ne peut s'appliquer si facilement dans un environnement multitâche, et la production de son, comme celle de graphisme, doit s'effectuer en passant par les bibliothèques systèmes de l'Amiga.

LE LOGICIEL

Il existe donc une plateforme logicielle pour la gestion des capacités sonores de notre chère machine. Toutefois, compte tenu du type de services attendus d'une telle couche logicielle, celle-ci est beaucoup plus restreinte que, par exemple, la `graphics.library`.

Communication via l'audio.device

Comme pour les autres devices de l'Amiga, l'audio.device utilise la notion de *IOWrite block* pour la communication avec votre tâche. Cette requête est appelée *IOAudio*. Pour communiquer avec ce device, la première chose à faire consiste donc à l'ouvrir.

On utilisera pour ce faire la fonction *OpenDevice()*, à laquelle on passera comme argument une structure *IOAudio* initialisée à zéro. Cette structure doit être déclarée sous forme de pointeur et allouée dynamiquement dans la CHIP RAM. Les deux fonctions ci-dessous vous permettront respectivement d'allouer et de libérer cette structure.

```
struct IOAudio *allocaudio()
{
    struct IOAudio *ioaud;

    ioaud = (struct IOAudio *)AllocMem(sizeof(struct IOAudio),
                                        MEMF_CHIP | MEMF_CLEAR);
    return(ioaud); /* Renvoie NULL si plus de mémoire */
}

void freeaudio(ioaud)
struct IOAudio *ioaud;
{
    if (ioaud)
        FreeMem(ioaud, sizeof(struct IOAudio));
}
```

Une fois cette allocation effectuée, on utilise le pointeur sur le *IOAudio* request block pour le passer à la fonction *OpenDevice()*. Il faudra ensuite stocker l'adresse du device créé, que l'on pourra réutiliser par la suite. De même, lorsque l'allocation a été effectuée, la fonction *OpenDevice()* a renvoyé une clef d'allocation (*ioa_AllocKey*) qu'il est nécessaire de connaître pour toutes les requêtes à venir. Voici un court listing résumant les différentes actions à effectuer.

```
int erreur;
struct Device *audiodev;
struct IOAudio *ioaud;
WORD clefalloc;

ioaud = allocaudio();
if(!ioaud)
{
    erreur = OpenDevice("audio.device", 0, ioaud, 0);
    if(!erreur)
    {
        /* On stocke l'adresse du device */
        audiodev = ioaud->ioa_Request.io_Device;

        /* On stocke la clef d'allocation */
        clefalloc = ioaud->ioa_AllocKey;
    }
}
```

Fonctionnalités de la plateforme logicielle Audio

Cette plateforme a pour but d'adapter l'utilisation du composant audio de l'Amiga à son système multitâche, en déchargeant le programmeur d'une série de contrôles systématiques sur l'environnement d'exécution.

Voici tout ce que l'audio.device permet de faire :

- Allouer un ou plusieurs canaux pour l'usage exclusif d'une tâche particulière.
- Etablir une priorité d'utilisation, pour autoriser l'émission d'un son important, même si le canal associé est déjà utilisé par une autre tâche.
- Autoriser une tâche de faible priorité à terminer l'utilisation d'un canal avant qu'il ne soit réquisitionné par une tâche de priorité plus élevée.
- Autoriser le lancement ou l'arrêt d'un canal, ainsi que la gestion d'une file d'attente pour jouer une série de sons en séquence.
- Spécifier le type de forme d'onde qui va être joué, ainsi qu'obtenir des informations sur le type de la forme d'onde jouée à un instant donné.
- Enfin, autoriser l'accès direct au composant audio pour une gestion personnalisée de ce dernier.

Allocation des canaux

Maintenant que le device Audio est ouvert, nous allons étudier comment s'allouer un canal audio en vue de son utilisation dans un environnement multitâche.

Le système audio permet de réclamer des canaux audio pour sa propre utilisation, suivant deux méthodes différentes : soit lors de l'appel à la fonction *OpenDevice()*, soit lors d'un appel séparé à la fonction *ADCMD_ALLOCATE*.

Voyons ensemble la première méthode.

On peut établir la priorité avec laquelle une tâche va pouvoir utiliser les canaux audios. Cette priorité s'échelonne entre -128 (minimum) et +127 (maximum). Par défaut, elle est fixée à 0. Pour la modifier, il suffit d'assigner au champ *ioa_Request.io_Message.mn_Node.In_Pri* de l'*IOAudio* Request block la valeur désirée avant l'appel à *OpenDevice()*.

Maintenant, il s'agit de réclamer les canaux pour la tâche. La première chose à faire, c'est d'indiquer le nombre de canaux que l'on désire utiliser : on met ce nombre dans le champ *ioa_Length*. Ensuite, on associe au champ *ioa_Data* un tableau de BYTES de la taille de *ioa_Length*. Chaque BYTE contient sous forme d'un code le numéro du canal à réserver (on l'appelle aussi *Allocation Byte*).

Chaque Allocation Byte contient en fait quatre bits (ceux de plus faible poids), un par canal possible. Un bit positionné à 1 correspond directement au canal sélectionné.

Pour réserver une paire de canaux pour un usage stéréophonique, il faut positionner deux des bits représentant respectivement les canaux gauches et droites désirés. On ne peut allouer que deux canaux en mode stéréophonique. Toutefois, si l'on indique plusieurs Allocation Byte (et par là même, plusieurs possibilités stéréo), le système choisira une paire quelconque, en fonction de la disponibilité des canaux. Si l'on ne propose qu'une seule paire et que l'un des canaux, ou les deux, sont indisponibles, l'allocation ne s'effectue pas. Pour savoir les résultats d'une demande d'allocation, il faut inspecter le champ *ioa_Request.io_Unit*.

Il est à noter que tout canal alloué par une tâche doit être explicitement libéré lorsque cette tâche n'en a plus besoin, cette libération des ressources n'étant pas automatique.

Le tableau ci-dessous reprend l'ensemble des possibilités offertes pour l'allocation des canaux.

Canal Audio	Position du bit	Valeur Bin	Valeur Dec
0	0	0001	1
1	1	0010	2
2	2	0100	4
3	3	1000	8
0 et 1		0011	3
0 et 2		0101	5
3 et 1		1010	10
3 et 2		1100	12

Le listing ci-dessous est un exemple de fonction permettant de s'allouer un certain nombre de canaux avec une priorité donnée en même temps que le device audio.

```
BYTE ouvreaudio(ioaud, canaux, longueur, priorite)
struct IOAudio *ioaud;
BYTE *canaux; /* tableau correspondant aux canaux à ouvrir */
SHORT longueur; /* nombre d'éléments du tableau */
BYTE priorite; /* priorité associé aux canaux */
{
    int erreur;

    ioaud->ioa_Request.io_Message.mn_Nde.ln_Pri = priorite;
    ioaud->ioa_Data = (UBYTE) canaux;
    ioaud->ioa_Length = longueur;

    erreur = OpenDevice("audio.device", 0, ioaud, 0);

    if(erreur)
        return((BYTE) 0);
    else
        return(((UBYTE) (ioaud->ioa_Request.io_Unit)) & 0xFF);
}
```

Passons maintenant à la deuxième méthode.

Dans le cas où l'on voudrait modifier la priorité associée à un canal après l'appel à *OpenDevice()*, il vous faudra utiliser la commande *ADCMD_SETPREC*. La fonction suivante permet de réaliser ce changement de priorité.

```
int ChangePriorite(ioaud, canal, priorite)
struct IOAudio *ioaud;
BYTE canal;
BYTE priorite;
{
    ioaud->ioa_Request.i_Command = ADCMD_SETPREC;
    ioaud->ioa_Request.io_Unit = canal;
    ioaud->ioa_Request.io_Message.mn_Node.ln_Pri = priorite;
    BeginIO(ioaud);
    WaitIO(ioaud); /* On attend la fin de l'opération */
    return(ioaud->ioa_Request.io_Error); /* 0 si aucune erreur */
}
```

Pour allouer les canaux après l'ouverture du device, il faut faire appel à la commande *ADCMD_ALLOCATE*. La fonction ci-dessous permet d'allouer ainsi les canaux.

```
UBYTE allocanaux(ioaud, canaux, longueur)
```

```
struct IOAudio *ioaud;
BYTE *canaux;
SHORT longueur;
{
    int erreur;
    ioaud->ioa_Request.i_Command = ADCMD_ALLOCATE;
    ioaud->ioa_Data = canaux;
    ioaud->ioa_Length = longueur;
    ioaud->ioa_Request.ioa_Flags = ADIOF_NOWAIT | IOF_QUICK;
    BeginIO(ioaud);
    erreur = WaitIO(ioaud); /* différent de zéro si erreur */

    if(ioaud->ioa_Request.io_Flags & IOF_QUICK)
        GetMsg(ioaud->ioa_Request.io_Message.mn_ReplyPort);

    if(erreur)
        return((UBYTE) 0);

    return((UBYTE) (ioaud->ioa_Request.io_Unit & 0xFF));
}
```

Attente d'allocation

Dans toutes les formes d'allocation, votre tâche est mise en sommeil (via la fonction *Wait()* d'Exec) jusqu'à complétion de la requête système. Chaque fois qu'un ou plusieurs canaux sont libérés, le device Audio examine les différentes requêtes, en commençant par les plus prioritaires. S'il n'y a pas assez de canaux libres pour répondre à la requête, la tâche restera en sommeil jusqu'à ce que sa requête puisse être honorée.

Cette attente est bien mise en évidence dans la routine *allocanaux()* ou *changepriorite()*. Par contre, dans toutes les routines utilisant *OpenDevice()*, il n'y a rien de visible. La fonction *OpenDevice()* ne retourne rien tant que l'allocation n'a pas été possible.

Si vous ne désirez pas que votre tâche attende la fin de l'allocation, vous pouvez positionner dans le champ *ioa_Request.io_Flags* la valeur *ADIOF_NOWAIT* : si l'allocation ne peut s'effectuer immédiatement, alors *ioa_Request.io_Error* sera rempli par le système avec la valeur *IERR_ALLOCFAILED*. Si l'allocation a échoué, la valeur retournée est systématiquement zéro.

Utilisation des canaux

Quand vous avez ouvert l'audio.device, associé les priorités et alloué les canaux, la phase d'initialisation est terminée et vous pouvez utiliser ces canaux.

Lorsque vous avez alloué plusieurs canaux audio, le système remplit le champ *ioa_Request.io_Unit* avec une valeur représentant les canaux alloués. De même, il assigne une clef d'allocation unique au champ *ioa_AllocKey* représentant la requête.

Si vous créez plusieurs *IOAudio Request Block*, par exemple à l'aide de la fonction *allocaudio()*, pour créer une file d'attente de commandes audios, vous devez recopier dans chacun des ces blocs : l'adresse du device, le contenu du champ *ioa_Request.io_Unit* et le contenu du champ *ioa_AllocKey*, ceci afin de permettre d'envoyer des requêtes aux canaux audios précédemment ouverts.

Toutefois, quand vous avez alloué une paire de canaux stéréos, vous n'êtes pas obligé de gérer deux Request Blocks : en effet, il suffit de changer, entre chaque envoi de requête, le contenu du champ *ioa_Request.io_Unit* avec le numéro du canal droit et gauche.

Par exemple, vous vous êtes alloué les canaux 1 et 2. Le numéro de Unit renvoyé par le système est donc 0110. Pour envoyer des commandes séparées aux deux canaux, il vous suffira d'utiliser un numéro de Unit de 0100 lors de votre requête pour accéder au seul canal droit, et 0010 pour le gauche.

Vérouillage d'un canal

Lorsque vous utilisez un canal audio quel qu'il soit, vous avez toujours le risque de vous le faire réquisitionner par une autre tâche plus prioritaire. Ceci peut entraîner l'arrêt brutal du son, ce qui, vous en conviendrez, est quelque peu gênant.

Pour résoudre ce problème, deux solutions s'offrent à vous. La première qui vient à l'esprit, consiste à s'allouer le canal avec la priorité maximum (127). Cette solution, bien qu'efficace, fait peu de cas d'un environnement multitâche, d'autant plus qu'une autre tâche peut

déjà avoir eu la même idée.

Une autre solution, respectant bien mieux l'environnement, consiste à verrouiller le canal. Lorsqu'une tâche 1, de plus haute priorité, tente de s'allouer un canal verrouillé par une tâche 2, ce canal n'est pas libéré, mais la tâche détenant le canal (en l'occurrence la tâche 2), reçoit un message lui demandant de le libérer au plus vite. La commande de verrouillage est `ADCMD_LOCK`.

Je ne saurais trop vous conseiller de verrouiller le canal, juste après son allocation, afin d'éviter tout problème. Mais n'oubliez pas de le déverrouiller dès que vous n'en avez plus besoin !

Les deux fonctions ci-dessous vous permettront de verrouiller et déverrouiller un canal audio, alloué précédemment.

```
verroucanal(ioaud, canal)
struct IOAudio ioaud;
UBYTE canal;
{
    ioaud->ioa_Request.io_Command = ADCMD_LOCK;
    ioaud->ioa_Request.io_Flags = 0;
    BeginIO(ioaud);

    /* Lors de l'utilisation de cette commande un WaitIO() n'est pas
    nécessaire puisqu'elle ne rend pas la main à la tâche tant que le
    verrouillage n'est pas effectué */
}

deverroucanal(ioaud)
struct IOAudio ioaud;
{
    /* Attention, le numéro de Unit et la clef d'allocation doivent
    être correctes sous peine d'un crash spectaculaire */

    ioaud->ioa_Request.io_Command = ADCMD_FREE;
    ioaud->ioa_Request.io_Flags = IOF_QUICK;
    BeginIO(ioaud);
    WaitIO(ioaud);
}
```

Quelques autres commandes

Nous terminerons par un descriptif de quelques autres commandes de contrôle de la sortie sonore. Ces commandes sont les suivantes :

- **CMD_WRITE** : insère dans la file d'attente associée à un canal, une forme d'onde. Cette commande peut inclure les données pour la période d'échantillonnage ainsi que le volume de sortie. Par défaut, ce sont les valeurs de la précédente commande `CMD_WRITE` qui sont prises.
- **ADCMD_PERVOL** : change la période d'échantillonnage ou le volume (ou les deux) associée à une forme d'onde qui est en cours d'émission dans le canal de sortie.
- **ADCMD_FINISH** : stoppe l'exécution de la `CMD_WRITE` en cours, interrompant ainsi l'émission de la forme d'onde, soit immédiatement, soit à la fin du cycle en cours.
- **CMD_STOP** : stoppe temporairement l'émission du canal de sortie.
- **CMD_START** : redémarre l'émission sur le canal de sortie (interrompu au préalable par un `CMD_STOP`).
- **CMD_FLUSH** : retire toutes les formes d'onde à jouer qui se trouvent dans la file d'attente associée à un canal de sortie.
- **CMD_RESET** : restaure tous les registres audio à leur état initial. Cette commande affecte tous les canaux de sortie, à l'exception des éventuels verrouillés.
- **CMD_READ (synchronisation)** : renvoie un pointeur sur une structure `IOAudio` vous informant des canaux actuellement en train de jouer.
- **ADCMD_WAITCYCLE (synchronisation)** : bloque la tâche jusqu'à l'accomplissement de la commande `CMD_WRITE` en cours. Vous devez utiliser la fonction `DoIO()` pour envoyer cette commande au device.

En utilisant ces deux dernières commandes, une tâche peut aisément se synchroniser avec la sortie audio. Cette possibilité peut être très utile pour synchroniser des sons avec des actions graphiques, par exemple (émission d'un son lorsqu'un objet heurte un obstacle, etc.).

Le mois prochain, nous aborderons la théorie de restitution des sons et terminerons par un programme d'exemple vous faisant enfin entendre la douce voix de votre Amiga. Attention, les routines décrites dans ce numéro seront utilisées, ne les égarez donc pas.

Par Herr Doktor Von GlutenStimmellmDorf

L'état 17 réduit les trois derniers couplets à l'état d'un *Objet*. L'état intermédiaire est 2. Dans cet état, l'*Objet* nous dirige vers l'état 11, où ce dernier est réduit à l'état de *Complément*, déclenchant l'état 10.

File : (-1,0), (Sujet,7), (VERBE,2), (Complément,10)

L'état 10 réduit l'ensemble *VERBE Complément* en un *Prédicat* dirigeant le pointeur vers 7. Là, le pointeur devient 15 à cause du *goto*.

File : (-1,0), (Sujet,7), (Prédicat,15)

En 15, 'n' est analysé ce qui donne :

File : (-1,0), (Sujet,7), (Prédicat,15), ('n',18)

En 18, il y a réduction de l'expression en une phrase et pointage sur l'état 0.

En zéro, le déclenchement de *accept* signifie l'exécution du source C associé à la règle déclenchée, en l'occurrence (0), soit l'impression de "Phrase Déclarative\n".

Comme vous pouvez le voir, c'est certes fastidieux, mais, au fond, très simple. Je vous laisse imaginer, à la vue de cet exemple le nombre de règles déclenchées dans un analyseur syntaxique complexe !

Sans transition, nous allons maintenant passer à la deuxième partie de cette article, à savoir l'interfaçage entre Lex et Yacc.

LEX ET YACC : UN MARIAGE HEUREUX

Nous allons maintenant utiliser Lex pour créer la partie analyse lexicale de notre programme d'exemple. Les programmes Lex, je vous le rappelle, sont assez similaires aux programmes Yacc possédant en cela des règles pour reconnaître des mots et retourner les symboles associés.

Un programme Lex se compose de trois parties : une section définitions, une section règles et une section sous-programmes.

La section Règles se compose d'une table d'expressions auxquelles sont associées des actions, actions qui sont effectuées lorsque le mot en entrée satisfait les conditions requises dans l'expression.

Pour de plus amples informations sur Lex (FLex sur Amiga), je vous renvoie à l'article de l'Amiga NewsTech Numéro 28 concernant ce produit. Rappelons simplement que FLex est disponible sur la disquette Fred Fish N° 412.

Le petit programme Lex qui va être associé à notre programme Yacc va être en mesure de retrouver les symboles suivants :

```
NOM : Christine, Stéphane, Anne, disquette(s), article(s)
ADJECTIF: Vert(e)(s), Rouge(s), Petit(es)
VERBE : Prend, Range, Donne
NOMBRE de un à dix, Le, La
```

A chaque fois que Lex trouvera une de ces occurrences (ou un dérivé tel Vert et Verte), il déclarera ce qu'il a trouvé et retournera le symbole associé (dans l'exemple, *ADJECTIF*). C'est une tâche assez simple à réaliser, n'est-ce pas ? Je vous laisse donc trouver ce programme par vous-même, mais vous proposerai une solution possible le mois prochain.

CONCLUSION

Le mois prochain, nous continuerons cette étude de Yacc avec à la clef un analyseur complet (sources Yacc et Lex). En attendant, je vous souhaite un excellent mois.

Par Herr Doktor Von GlutenStimmellmDorf

DEMOS : EFFET DE ZOOM

A la demande d'un lecteur, j'ai ce mois ci, programmé une routine de zoom bitmap, telle que l'on peut la voir dans DPaint par exemple avec l'option stretch. Elle a pour but de capturer un rectangle dans une

Pour se servir de cette routine, il faut initialiser les coordonnées du rectangle source dans OX1, OY1, OX2 et OY2. (OX1, OY1) sont les coordonnées du point en haut à gauche du rectangle source et (OX2, OY2) celles du point en bas à droite. De plus, il faut initialiser les coordonnées du rectangle destination (DX1, DY1) pour le coin en haut à droite, et (DX2, DY2) pour en bas à gauche. La routine dans sa version actuelle, ne fait qu'afficher une seule fois la fenêtre 'stretchée', mais en se creusant un peu la tête, il est possible de l'utiliser pour faire des effets vidéo très intéressants.

L'image où l'on prend le rectangle source est chargée dans le programme par un *INCBIN* et est pointée par l'étiquette *PIC_BMP*. Cette image est au format 320 sur 256 en 16 couleurs et est au format bitmap standard (d'abord le plan 1, puis le plan 2, etc.). Elle est suivie de sa palette de couleurs.

On peut penser qu'il est assez stupide d'avoir choisi ce format, puisqu'une routine dont le seul but est de transformer l'image source en un format où chaque plan de ligne est entrelacé, est appelée en début de programme... Ce choix se justifie par le fait que ce format est celui que l'on obtient par défaut avec *IFF_Converter*, donc le plus facile à obtenir pour le lecteur.

FONCTIONNEMENT DU ZOOM

Le principe de base est très simple et est, en même temps, le plus rapide que je connaisse pour une telle application. Il consiste dans un premier temps à zoomer verticalement la fenêtre source, puis horizontalement le résultat obtenu. Cette technique permettant de ne zoomer que dans une seule direction à la fois (horizontale ou verticale), elle permet également l'utilisation du blitter. Ceci est bien plus rapide que la technique de base qui vient tout de suite à l'esprit d'une personne n'ayant jamais réfléchi au problème, à savoir tester la couleur de chaque point de la fenêtre source et de dessiner un rectangle de la même couleur dans la fenêtre destination. Le gain de vitesse de l'algorithme vient du fait que l'on traite des ensembles de points, et non les points un par un. Cette méthode présente cependant un certain désavantage : elle nécessite une étape intermédiaire et donc, utilise plus de mémoire.

Voici plus de détails : l'image d'où l'on extrait les premières données (les lignes horizontales) est appelée *FIRST_SCR_ADR*. On la déforme verticalement et l'on stocke l'étape intermédiaire dans une page appelée *LOG_SCR_ADR*. Puis on déforme horizontalement cette étape et on stocke le résultat dans la page *PHY_SCR_ADR*, qui est celle que l'on voit. C'est pour cette raison que l'on voit l'image se dessiner (rapidement) devant nos yeux quand on lance le programme.

Je vais vous expliquer comment faire un zoom d'une partie d'image en prenant comme exemple le zoom horizontal (qui augmente la hauteur de la fenêtre source). On initialise deux curseurs virtuels : un qui va se balader sur les y de la fenêtre destination du haut vers le bas, par pas de un, et un qui va se balader sur les y de la fenêtre source et dont le pas va varier selon un coefficient pré-défini. A chaque itération, on incrémente chaque y avec le pas qui lui correspond et on copie la ligne du y source sur la ligne du y destination.

C'est très simple, mais cela mérite une petite précision sur la manière de calculer le coefficient : il doit être un nombre réel et non un entier. Or le 68000 ne manipule que les entiers. J'ai donc choisi la technique des bits fractionnés : sur un chiffre de 32 bits, on déclare qu'un certain nombre de bits sont considérés comme étant derrière la virgule. Dans mon cas, ce chiffre est de 16 bits car il permet d'extraire plus

facilement la partie entière. En effet, quand on utilise un registre, une simple instruction swap suffit ; quand on utilise la Ram, il suffit d'adresser en '.w' au lieu de '.l'.

Pourquoi notre coefficient est-il réel, et non entier ? Tout simplement parce qu'il est le résultat de la division de la hauteur de la fenêtre source par la hauteur de la fenêtre destination. Le problème immédiat est donc : comment faire une division de telle sorte que l'on obtienne un résultat sur 32 bits ? J'ai trouvé la réponse dans le livre *Mise en oeuvre du 68000*, paru chez Sybex :

Nous savons maintenant comment fonctionne globalement le zoom et nous allons nous attarder sur les routines appelées *copy_horizontal_line* et *copy_vertical_line*. Pour la copie de ligne horizontale, rien de génial en fait : c'est une simple copie de la ligne avec un mintern de $A = D$. Pour la copie verticale, c'est légèrement plus complexe : dans un premier temps, le mintern doit être différent, car il est nécessaire de ne pas effacer tout les bits du mot destination, mais seulement le bit correspondant à la ligne verticale en cours. Pour cela, on choisit un mintern tel que $AB + aC = D$, avec A désignant le mask, B l'image et C le fond.

Mais tous les problèmes ne sont pas résolus pour autant, car comme chacun sait, le Blitter décale ses bits vers la droite et uniquement vers la droite, d'où un problème évident. Imaginons que l'on veut copier la ligne qui se trouve sur la colonne 13 sur la colonne 5. Le glissement doit s'effectuer vers la gauche, ce dont le Blitter est incapable. On est forcé de le simuler en utilisant le fait que tout les bits qui sortent à droite au bout de la ligne ré-apparaissent au début de la ligne suivante. Pour simuler le glissement à gauche, il faut donc remonter le pointeur d'une ligne (pour que les bits aillent bien là où il faut, car l'image sera descendue d'une ligne) puis augmenter bltsize d'une ligne pour que l'image soit affichée dans son intégralité et qu'il ne manque pas un point tout en bas de la ligne verticale.

Pour finir, je voudrais préciser que l'on aurait pu être plus rapide en utilisant le fait que bouger une ligne verticale est plus long que pour une ligne horizontale. Il est donc possible de faire une routine qui décide quel zoom faire en premier, afin de gagner du temps. Le test est assez simple : si l'on grossit la fenêtre verticalement, alors on zoome en horizontal en premier, sinon on reste dans l'ordre actuel.

La phrase du mois : *On ne manipule pas une marionnette avec un seul fil.*

Dernière minute : emporté par mon enthousiasme, j'ai soudain réalisé, mais trop tard, qu'en mode ascending, notre cher Blitter travaille complètement à l'envers, c'est-à-dire non seulement du bas vers le haut, mais aussi de la droite vers la gauche ! On gagnerait donc encore du temps en utilisant ce mode ! N'ayez crainte, je vous proposerai une routine de remplacement... dès que je l'aurai programmée.

* AUTEUR: j.etienne
* SUJET: zoom bitmap avec le blitter
* ASSEMBLEUR: devpack 2.14

OPT C- * opt case off

incdir 'include:'
include 'exec/exec_lib.i'
include 'hardware/custom.i'

custom = \$dff000
execbase = 4

BPL_X = 320
BPL_Y = 256
BPL_DEPTH = 4
BPL_WIDTH = BPL_X/8
BPL_SIZE = BPL_WIDTH*BPL_Y

WAIT_BLT: macro
 lea CUSTOM,a5
 btst#14,dmaconr(a5)
.LOOP_WAIT_BLT@:
 btst#14,dmaconr(a5)
 bne.s .LOOP_WAIT_BLT@
endm

***** programme principal *****

```

move.l (execbase).w,a6
lea CUSTOM,a5

CALLEXEC Forbid
move.w #$03e0,dmacon(a5) * all dma off except disk

move.w #(BPL_DEPTH<<12)+$200,bplcon0(a5)
clr.w bplcon1(a5)
clr.w bplcon2(a5)
move.w #BPL_WIDTH*(BPL_DEPTH-1),bpl1mod(a5)
move.w #BPL_WIDTH*(BPL_DEPTH-1),bpl2mod(a5)
move.w #$2981,diwstrt(a5) * \
move.w #$29c0,diwstop(a5) * > init un ecran 320*256
move.w #$0038,ddfstrt(a5) * \
move.w #$00d0,ddfstop(a5) */
bsr BUILD_COPLIST
move.l COPLIST_adr,copl1c(a5) * > run my COPLIST
clr.w copjmp1(a5) */
move.w #$83C0,dmacon(a5) * dma blt,copper & bpl
bsr PUT_PIC_IN_PHY_SCR

* fait le zoom
bsr ZOOM

WAIT move.b $bfec01,d0
not d0
ror.b #1,d0
cmp.b #$45,d0
beq INIT_END
btst #6,$bfe001
bne WAIT
bra INIT_END

*****
ZOOM: * init coor du rectangle source
move.l #120*$10000,OX1
move.l #000*$10000,OY1
move.l #220*$10000,OX2
move.l #100*$10000,OY2
* init coor du rectangle cible
move.l #000*$10000,DX1
move.l #000*$10000,DY1
move.l #319*$10000,DX2
move.l #255*$10000,DY2
* fait le zoom vertical puis horizontal
bsr ZOOM_VERTICAL
bsr ZOOM_HORIZONTAL
rts

*****
END_ZOOM

*****
ZOOM_HORIZONTAL:
moveq #0,d0
move.w DX2(pc),d0
sub.w DX1(pc),d0
moveq #0,d1
move.w OX2(pc),d1
sub.w OX1(pc),d1
swap d1
bsr DIV_32_BITS
move.l d0,COEF_HORIZONTAL

move.l DX1,CUR_OX
move.l DX1,CUR_DX
.LOOP_EACH_LINE
move.w CUR_OX(pc),d0
move.w DY1(pc),d1
move.w DY2(pc),d2
move.w CUR_DX(pc),d3
move.w DY1(pc),d4

bsr COPY_VERTICAL_LINE

move.l COEF_HORIZONTAL(pc),d0
add.l d0,CUR_OX
addq.w #1,CUR_DX

move.w CUR_DX(pc),d0
cmp.w DX2(pc),d0
blt.s .LOOP_EACH_LINE
rts

*****
END_ZOOM_HORIZONTAL

*****
COPY_VERTICAL_LINE:
* IN: d0 = CUR_OX = coor origine
* d1 = OY1 = coor origine
* d2 = OY2 = coor origine
* d3 = CUR_DX = coor destination
* d4 = DY1 = coor destination

sub.w d1,d2 * d2 = height
move.l LOG_SCR_ADR(pc),a0
mulu #BPL_WIDTH*BPL_DEPTH,d1
add.l d1,a0

```

```

moveq #$f,d1
and.w d0,d1 * d1 = shift of source
moveq #$f,d6
and.w d3,d6 * d6 = shift of target
sub.w d1,d6 * d6 = shift for blt

move.l PHY_SCR_ADR(pc),a1
mulu #BPL_WIDTH*BPL_DEPTH,d4
add.l d4,a1
and.w #$fff0,d3
lsr.w #3,d3
lea (a1,d3.w),a1 * a1 = target adr

tst.w d6
bge.s .SHIFT_GE_0
add.w #16,d6
lea -BPL_WIDTH*BPL_DEPTH(a1),a1
addq.w #1,d2
.SHIFT_GE_0

move.w #$8000,d5
lsr.w d1,d5
and.w #$fff0,d0
lsr.w #3,d0
lea (a0,d0.w),a0 * a0 = source adr

lsr.w #6,d2
or.w #1,d2 * d2 = BLTSIZE

ror.w #4,d6
and.w #$f000,d6
move.w d6,d1
or.w #$07ca,d1

lea CUSTOM,a5
WAIT_BLT
move.w #BPL_WIDTH*BPL_DEPTH-2,BLTBMOD(a5)
move.w #BPL_WIDTH*BPL_DEPTH-2,BLTCMOD(a5)
move.w #BPL_WIDTH*BPL_DEPTH-2,BLTDMOD(a5)
move.l #-1,BLTAFWM(a5)
moveq #BPL_DEPTH-1,d7
.LOOP_EACH_BPL
WAIT_BLT
move.w d5,BLTADAT(a5)
move.w d1,BLTCON0(a5)
move.w d6,BLTCON1(a5)

move.l a0,BLTBPT(a5)
move.l a1,BLTCPT(a5)
move.l a1,BLTDPT(a5)
move.w d2,BLTSIZE(a5)
lea BPL_WIDTH(a0),a0
lea BPL_WIDTH(a1),a1
dbf d7,.LOOP_EACH_BPL

rts
***** END_COPY_VERTICAL_LINE

ZOOM_VERTICAL:
*****
moveq #0,d0
move.w DY2(pc),d0
sub.w DY1(pc),d0
moveq #0,d1
move.w OY2(pc),d1
sub.w OY1(pc),d1
swap d1
bsr DIV_32_BITS
move.l d0,COEF_VERTICAL

move.l OY1,CUR_OY
move.l DY1,CUR_DY
.LOOP_EACH_LINE
move.w OX1(pc),d0
move.w CUR_OY(pc),d1
move.w OX2(pc),d2
move.w DX1(pc),d3
move.w CUR_DY(pc),d4

bsr COPY_HORIZONTAL_LINE

move.l COEF_VERTICAL(pc),d0
add.l d0,CUR_OY
addq.w #1,CUR_DY

move.w CUR_DY,d0
cmp.w DY2,d0
blt.s .LOOP_EACH_LINE

rts
*****
END_ZOOM_VERTICAL

COPY_HORIZONTAL_LINE: *****
* IN: d0 = OX1 = coor origine

```



```

*      d1 = CUR_OY      = coor origine
*      d2 = OX2         = coor origine
*      d3 = DX1         = coor destination
*      d4 = CUR_DY      = coor destination

      move.l  FIRST_SCR_ADR(pc),a0
      mulu   #BPL_WIDTH*BPL_DEPTH,d1
      add.l  d1,a0

      and.w  #$fff0,d0
      and.w  #$fff0,d2
      lsr.w  #3,d0
      lsr.w  #3,d2

      lea    (a0,d0.w),a0

      move.l  LOG_SCR_ADR(pc),a1
      subq.w  #1,d4
      mulu   #BPL_WIDTH*BPL_DEPTH,d4
      add.l  d4,a1

      and.w  #$fff0,d3
      lsr.w  #3,d3
      lea    (a1,d3.w),a1

      sub.w   d0,d2
      addq.w  #2,d2      * d2 = largeur de la ligne en octet

      move.w  #BPL_WIDTH,d1
      sub.w   d2,d1      * d1 = modulo

      lsr.w  #1,d2
      or.w   #BPL_DEPTH<<6,d2 * d2 = BLTSIZE

      lea    CUSTOM,a5
      WAIT_BLT
      move.l  #-1,BLTAFWM(a5)
      move.l  #$09f00000,BLTCON0(a5)
      move.w  d1,BLTAMOD(a5)
      move.w  d1,BLTDMOD(a5)
      move.l  a0,BLTAPT(a5)
      move.l  a1,BLTDPT(a5)
      move.w  d2,BLTSIZE(a5)
      rts

***** END_COPY_HORIZONTAL_LINE

DIV_32_BITS: *****
* routine de division d'un chiffre 32 bits part un chiffre 16 bit
* avec un resultat 32 bits. Elle est directement tire de mise en
* oeuvre du 68000 de sybex (tres bon livre soit dit en passant).
* in: d0 = X  chiffre 16 bits
*      d1 = YZ chiffre 32 bits
* out: d0 = resultat 32 bits
* d0-d3 sont detruit
      moveq  #0,d3

      divu   d0,d1
      bvc.s  .RESULT
      move.l d1,d2
      clr.w  d1
      swap  d1
      divu   d0,d1
      move.w d1,d3
      move.w d2,d1
      divu   d0,d1
      move.l d1,d0
      swap  d1
      move.w d3,d1
      swap  d1
      move.l d1,d0
      rts

***** END_DIV_32_BITS

*****
PUT_PIC_IN_PHY_SCR:
* Cette routine prend l'image en format bitmap et la met au
* format plan de ligne entrelacee.

      lea    PIC_BMP,a0
      move.l FIRST_SCR_ADR(pc),a2
      move.w #BPL_DEPTH-1,d0
      .LOOP_EACH_BPL
      move.l a2,a1
      move.w #BPL_Y-1,d1
      .LOOP_EACH_LINE:
      move.w #BPL_X/16-1,d2
      .LOOP_EACH_WORD:
      move.w (a0)+,(a1)+
      dbf    d2,.LOOP_EACH_WORD
      lea    BPL_WIDTH*(BPL_DEPTH-1)(a1),a1
      dbf    d1,.LOOP_EACH_LINE
      lea    BPL_WIDTH(a2),a2
      dbf    d0,.LOOP_EACH_BPL
      lea    COLOR+CUSTOM,a1
      move.w #(1<<BPL_DEPTH)-1,d0
      .LOOP_EACH_COLOR:
      move.w (a0)+,(a1)+

```

```

      dbf    d0,.LOOP_EACH_COLOR
      rts
*****
END_PUT_PIC_IN_PHY_SCR

INIT_END:
*****
* reactivation de l'ancienne COPLIST
      lea    CUSTOM,a5
      move.l (EXECBASE).w,a6
      lea    GfxName(pc),a1 * nom de la library ds a1
      moveq  #0,d0          * version 0 (the last)
      CALLEXECOpenLibrary    lib graphique ouverte
      move.l d0,a4          * adr de graphicbase ds a4
      move.l 38(a4),copllc(a5) * chargement de l'adr de
      clr.w  copjmp1(a5)    * l'old COPLIST et lancement

      move.w #$83e0,dmacon(a5) * activation des canaux dma
      CALLEXECPermit        * multi switching autorise

* et on retourne d'ou l'on vient.
      moveq  #0,d0          * flag d'erreur desactive
      rts

GfxName dc.b "graphics.library",0
      EVEN
*****
end_INIT_END

*****
BUILD_COPLIST:
      move.l COPLIST_ADR,a0
      move.l PHY_SCR_ADR,d2
      moveq  #BPL_DEPTH-1,d0
      move.w #bplpt,d1

      .LOOP_INIT_BPL_IN_CLIST
      move.w d1,(a0)+
      addq.l #2,d1
      swap  d2
      move.w d2,(a0)+
      swap  d2
      move.w d1,(a0)+
      addq.l #2,d1
      move.w d2,(a0)+
      add.l  #BPL_WIDTH,d2
      dbf    d0,.LOOP_INIT_BPL_IN_CLIST

      move.l #$fffffffe,(a0)+ * montre la fin de la clist
      rts
*****
END_BUILD_COPLIST

COEF_VERTICAL:      ds.l 1
COEF_HORIZONTAL:    ds.l 1
*
CUR_OX ds.l 1
CUR_OY ds.l 1
CUR_DX ds.l 1
CUR_DY ds.l 1
*
OX1 ds.l 1
OY1 ds.l 1
OX2 ds.l 1
OY2 ds.l 1
*
DX1 ds.l 1
DY1 ds.l 1
DX2 ds.l 1
DY2 ds.l 1

***** datas
* variables
LOG_SCR_ADR: dc.l ECRAN1
PHY_SCR_ADR: dc.l ECRAN2
FIRST_SCR_ADR: dc.l ECRAN3
COPLIST_ADR: dc.l COPLIST
PIC_BMP:      INCBIN Freddy.raw ; C'est l'image
              INCBIN Freddy.pal ; et sa palette

      section ZONE_CHIP,BSS_C

ECRAN3:      ds.l BPL_WIDTH*BPL_Y*BPL_DEPTH/4
ECRAN1:      ds.l BPL_WIDTH*BPL_Y*BPL_DEPTH/4
ECRAN2:      ds.l BPL_WIDTH*BPL_Y*BPL_DEPTH/4
COPLIST:      ds.l 1000*4 * taille inconnue donc on prevoit
gros
      end

```

Par Jérôme Etienne

Cher A.N.T,

Suite à votre article sur Draco, et au fait que vous dites avoir feuilleté les magazines comme Amiga Revue pour y trouver les adresses des distributeurs susceptibles de le distribuer, je tiens à vous dire que je suis très déçu que vous ne disiez pas qu'il y a quelqu'un en France (en l'occurrence moi-même) qui distribue la version française de Draco, et ceci depuis bientôt un an.

Pire encore, la même chose viens de se reproduire dans votre numéro de Janvier, à propos de votre article sur le Pascal ou là encore, vous ne dites pas qu'il existe une version française du compilateur PCQ ! Toutes les énormes documentations de Draco et de PCQ ont été entièrement traduites en Français par mon équipe de traducteurs !

Depuis un an, j'ai fait publier plus de 20 petites annonces dans votre grande soeur Amiga Revue, à raison de deux par mois. Il est donc impossible que vous n'ayez pas vu au moins une de ces annonces, puisque vous dites avoir feuilleté ces journaux ! Rendez-vous compte que certaines personnes ont peut-être besoin de ces versions françaises et qu'elles ne le sauront sans doute jamais si vous le leur cachez à chaque fois de cette façon ! D'autant plus que ce sont des traductions de qualité professionnelle et qu'elles respectent les auteurs. Il ne sert à rien de nous décarcasser pour produire toutes ces traductions si les journaux ne le disent pas, car cet énorme travail de traduction serait un véritable gâchis pour tout le monde. Mon équipe de traducteurs et moi-même pensons que seules de telles traductions pourront faire mieux connaître les auteurs de DP et de Shareware internationaux en France. En effet, un utilisateur Français qui ne comprend pas très bien un logiciel du domaine public en version originale, ne pourra jamais l'apprécier à sa juste valeur et n'aura de ce fait aucune envie de récompenser l'auteur lorsqu'il s'agit, par exemple, d'un logiciel Shareware (comme dans le cas pour Draco). Par conséquent, seules de telles traductions permettront aux auteurs internationaux d'être mieux respectés en France.

Quand à mes tarifs (environ 40 frs par disquette, envoi rapide compris), ils sont très exactement les mêmes que ceux que recommande Fred Fish lui-même pour la distribution de sa collection, à savoir 6 dollars par disquette. Mes tarifs sont donc tout-à-fait du même ordre de grandeur, avec les traductions en plus ! Ceci est possible parce que mes traducteurs ont tous gracieusement accepté de réaliser ces traductions à prix réduit. Je ne peux malheureusement pas diminuer ces tarifs, car vues les documentations de plus en plus énormes des logiciels du domaine public, ces traductions intégrales me coûtent aussi de plus en plus cher. Néanmoins, je tiens beaucoup à ce que ce soit de vrais professionnels de la langue Anglaise qui s'occupent de traduire, parce qu'il ne s'agit pas de donner aux utilisateurs des traductions approximatives qui pourraient les induire en erreur, ce qui irait alors totalement à l'encontre du respect des auteurs et aussi du respect des utilisateurs. Et puis il faut aussi savoir que seuls des traducteurs professionnels peuvent avoir les compétences adéquates et le courage pour traduire de telles documentations.

Par exemple, nous avons pu traduire dernièrement l'intégralité du Hardware Référence Manual. Il est disponible sur deux disquettes contre 100 frs, tous frais d'envoi compris. C'est actuellement notre plus grosse traduction, puisque de plus d'un Méga-Octet, et nous espérons bien que les journaux Amiga français annonceront cette traduction, car si nous l'avons faite, c'est bien pour que l'Amiga soit encore plus fort en France. Et ceci n'est qu'un tout petit exemple car depuis plus d'un an, nous avons traduit à nous seuls la plus grosse partie du DP international. En particulier tous les langages du Domaine Public : assembleur, C, Pascal, Draco, Fortran, PostScript avec Post (c'est aussi un langage !), y compris les 4 volumes du célèbre "C Manual". Ils ont tous été entièrement traduits en Français. Sans oublier les Ray-Tracers comme DWB Render 2.0 et la nouvelle version 2.12 de DKB Trace. Tout cela entièrement traduit en Français, avec des traductions qui respectent les auteurs et des tarifs tout-à-fait raisonnables.

Comment pouvez-vous ignorer tout cela, je me le demande ! Je recommande donc à tous ceux qui sont intéressés de bien feuilletter les annonces d'Amiga-Revue depuis plus d'un an déjà. Vous y trouverez tous les tarifs correspondants.

Il serait trop dommage que ceux qui ont réellement besoin de ces traductions ne le sachent pas et achètent à la place les versions originales, ce serait d'ailleurs un double gâchis : pour eux, parce qu'ils perdront énormément de temps à comprendre et installer ces logiciels (nos versions françaises étant aussi prêtes à l'emploi !) et aussi pour moi et mes traducteurs, parce c'est un énorme travail que de réaliser toutes ces traductions et installations, lesquelles par manque d'information sont condamnées à dormir dans mes tiroirs !

Voilà pourquoi j'ai bien été obligé de vous écrire une seconde fois cette lettre, bien que ce ne soit pas mon rôle de faire tout ce travail d'information, mais vous en conviendrez, il était urgent de réparer ces terribles oublis, ne serait-ce que par respect des personnes susceptibles d'avoir besoin de ces traductions. J'espère donc que cette fois-ci, vous pourrez publier rapidement cette lettre et vous en remercie beaucoup.

HAMMOUCHE Serge

3, Rue Anatole France

13220 Chateauneuf-Les-Martigues

Voilà, z'êtes content ? Votre lettre a été publiée dans son intégralité. Vous hurlez à la désinformation et à la censure, menacez de boycotter l'ANT (en glissant des textes dythirambiques sur vos disquettes) et tout ça pour quoi ? Parce que nous avons refusé de faire votre publicité ! D'après vous, il faudrait que la presse Amiga toute entière s'agenouille devant vous et vous appelle Monseigneur, parce que vous et quelques traducteurs, "professionnels de l'Anglais" (et de l'informatique, j'ose espérer ?) avez réalisé la traduction de plusieurs logiciels du domaine public ?

Certes, l'entreprise est louable, et les programmeurs qui ont du mal en Anglais (on leur en souhaite toutefois moins que vous en Français...) vous en seront reconnaissants. D'autant plus que les tarifs que vous annoncez semblent effectivement parfaitement honnêtes.

Mais ce n'est pas le rôle ni l'envie de l'ANT que de faire de la publicité gratuite ! Nos pigistes eux-mêmes sont sévèrement censurés dès qu'ils s'y essaient (Herr Doktor von GlutenStimmellImDorf est membre fondateur de l'association D2P, et Pierre Philippe Launay est le maître d'oeuvre d'Annabella, une collection de cours de programmation en GFA-Basic). Si publicité il doit y avoir dans l'ANT, elle sera payante et servira à financer des pages supplémentaires. Nous n'en sommes toutefois pas encore là.

Restait la possibilité d'une "news", soit dans Amiga Revue, soit dans l'ANT, soit même, pourquoi pas, dans les deux. Mais pour ce faire, encore eût-il fallu que vous nous envoyiez un exemplaire d'évaluation de vos produits : vous comprendrez que nous ne puissions conseiller un quelconque article à nos lecteurs sans être nous-mêmes sûrs qu'il répondra à leurs attentes... Vous avez tort d'estimer que "ce n'est pas votre rôle que d'effectuer tout le travail d'information". Si vous désirez vendre vos traductions, personne ne le fera à votre place. Prenez donc un peu vos responsabilités et cessez de vous reposer sur les journalistes !

Voilà qui clôt notre Requester de ce mois. Je tiens à préciser que vos lettres sont de plus en plus nombreuses, concernant les nouvelles rubriques de l'ANT (Pascal, GFA, Algos). En moyenne, le pour et le contre s'équilibrent, avec un très, très léger avantage en faveur de ceux qui préféreraient voir les deux premières disparaître de la revue. En tout cas, rien de suffisant pour prendre une décision tout de suite. Continuez donc à nous écrire... Après tout, c'est pour votre bien !

Amiga NewTech (ANT pour les intimes) est une publication mensuelle de seulement 32 pages en noir et blanc, de Commodore Revue SARL, société d'édition très connue dans le milieu Amiga et pas du tout dans le milieu de l'agriculture biologique, sise au 5, rue de la Fidélité, 75010 Paris. Tel : (1) 42.46.92.90. C'est à deux pas de la Gare de l'Est, ce qui est tout de même bien pratique pour partir en vacances en Alsace. En tout cas, plus que pour Albertville. Jean-Yves Primas est le Directeur de la Publication. On n'en dira pas plus, il est recherché par Interpol pour insulte à des douaniers italiens dans l'exercice de leurs fonctions. Stéphane Schreiber, c'est-à-dire votre serviteur, est le Rédacteur en Chef (uniquement l'après-midi). Ça signifie que je peux faire tout ce que je veux, comme raconter des conneries dans l'Ours, glisser des messages subliminaux, personne ne viendra me chercher des poux dans le tête pour autant. Et croyez-moi, je ne me gêne pas. La preuve, j'en profite également pour être Maquettiste. Non pas que ça me rapporte plus (une misère, même pas de quoi m'acheter une Ferrari), mais comme ça, j'ai un contrôle total sur toute la fabrication du canard. Yerk, yerk, yerk. Cela dit, j'ai tout de même quelques pigistes qui m'aident bien à les remplir, ces seulement 32 pages en noir et blanc. Pascal Amiable, Bruno Bailluet, Jérôme Enenne, Olivier Garrigues, François Gueugnon, Emmanuel Hocdet, Denis Jarril, Pierre-Philippe Launay, François Lionet, Philippe Rivailon, Stéphane Rubinstein et Lucien Turski-Marcon. Le flashage des films est réalisé par ColorWay, dont les (superbes) locaux se trouvent juste en dessous des nôtres, soit au 5, rue de la Fidélité, 75010 Paris. Tel : (1) 47.70.87.87. L'impression est l'oeuvre de INO Montrouge, située au 9, rue Auber, 92120 Montrouge. Tel : (1) 46.56.27.69. Les abonnements sont gérés par MCM ANT, 16, Quai Jean-Baptiste Clément, 94140 Afortville. Tel : (1) 43.78.92.10. Enfin, saluons au passage et glorifions à leur juste valeur Yves Huitric, Anne Olivelli, Christine Robert, Valérie Ambrosio, Madame Maria et Mme Manon Primas, sans qui l'ANT ne saurait être bouclé en temps, en heure et en jour. Alleluia !

