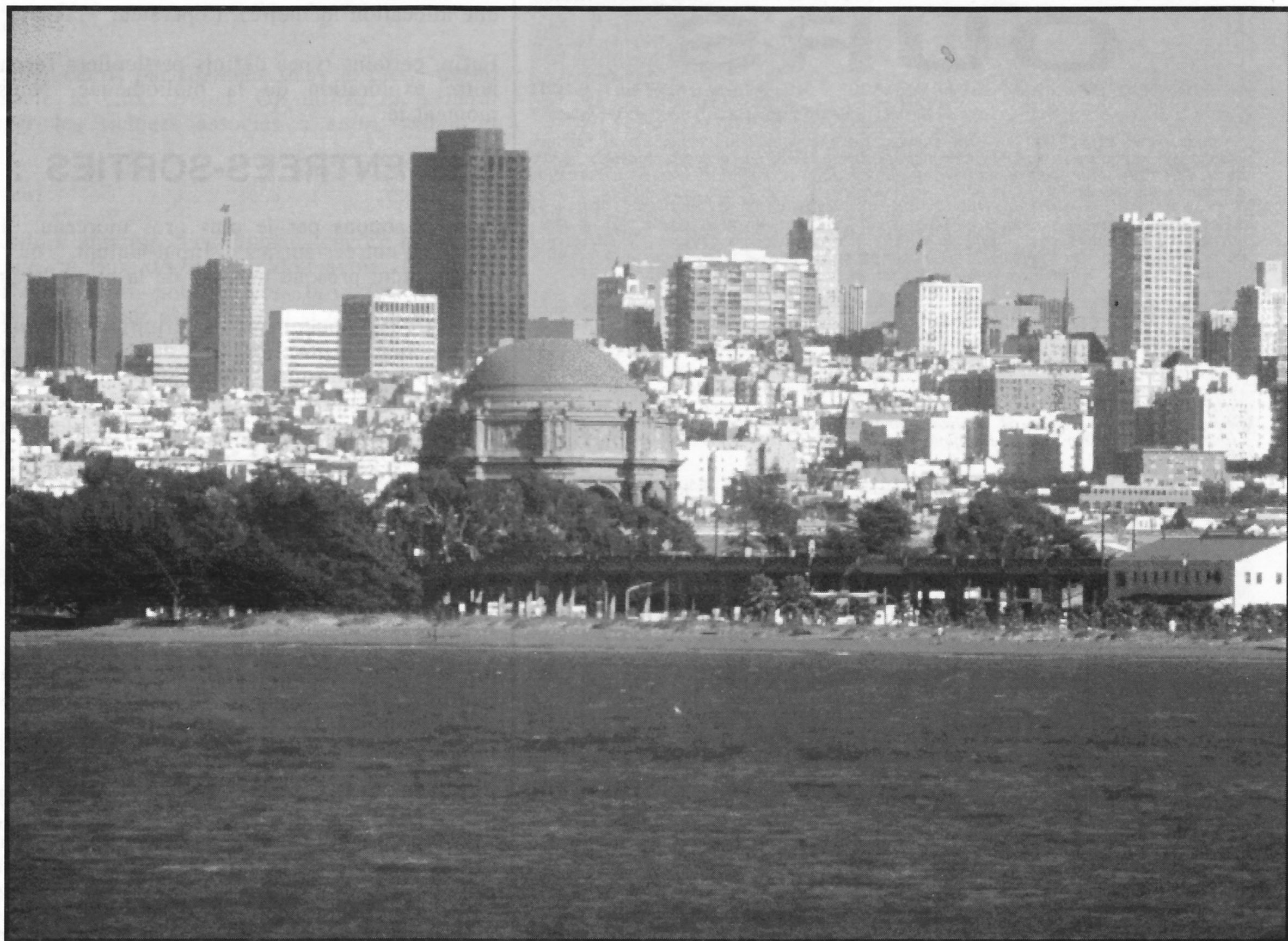


AMIGA NEWS-TECH

EN VENTE UNIQUEMENT PAR ABONNEMENT

ExecBase : le C ANSI (II)	2	Denis Jarril
AMOS : Trucs en vrac (III)	4	François Lionet
Daisy : Multi_Rainbows	5	Daisy Lionet
ARexx : Le déverminage (II)	6	François Gueugnon
Pascal : Présentations (II)	8	Olivier Garrigues
GFA : Le trackdisk.device	10	Pierre Philippe Launay
Algos : Les piles	12	Bruno Bailluet et Lucien Turski
Transactor : Test-Disks	15	François Braün
Hardware : La voix de Paula (II)	16	Philippe Rivaillon
ToolBox : YACC	18	Herr Doktor Von Machin
Utilitaire : Le Greffeur	20	Frédéric Mazué
Demos : Les ellipses	24	Thomas Landspurg
Demos 2 : Etoiles en 3D	26	Jérôme Etienne
Routinettes : Exist()	29	Tonton François Lionet et Max
Requester	31	Ok & Cancel



*Après le pont,
la ville :
bienvenue à
San Francisco !*

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

45 F

Cela fait un bon bout de temps maintenant que nous n'avons pas effectué de sondage pour savoir si l'Amiga NewsTech, dans sa formule actuelle, répondait à vos attentes.

Rassurez-vous, nous allons y remédier dès maintenant. Non pas sous la forme d'un véritable sondage, comme dans le numéro 20, mais plutôt en vous incitant à nous donner encore et toujours votre avis : n'hésitez pas à nous écrire ce que vous pensez de telle ou telle rubrique, ce que vous aimeriez trouver en plus dans l'ANT ou, au contraire, ce qui vous semble inutile.

Ceci est plus particulièrement important pour les rubriques génériques (langages, Algos, Hardware...). Certains d'entre vous nous ont déjà fait part de leur manque d'intérêt pour Pascal et GFA et de leur soutien pour Algos (cf. Requirer). Si cela devait s'avérer être la tendance générale, nous supprimerions sans hésiter les deux premières.

La balle est donc dans votre camp. Vous pouvez nous écrire à l'adresse habituelle, ou par Minitel, 3615 ANT, BAL Redac : elle est lue au minimum une fois par semaine.

A vous de jouer.

Stéphane Schreiber

OURS

Amiga NewsTech (ANT pour les intimes) est une publication mensuelle et vendue uniquement par abonnement, de Commodore Revue SARL, 5, rue de la Fidélité, 75010 Paris. Téléphone de la rédaction : (1) 42.46.92.90. ISSN 1157-8912. Commission Paritaire : et puis quoi encore ?

Bon, j'vous explique comment ça fonctionne : tout en haut de l'échelle, y'a un mec qui allonge les thunes pour fabriquer le canard. C'est Jean-Yves Primas, le **Directeur de la Publication**. On l'appelle "Jean-Yves" en temps normal, "Monsieur le Directeur" quand on veut lui demander une augmentation, et "enfoiré" quand il la refuse.

En dessous, y'a le type qui décide du contenu du magazine. C'est Stéphane Schreiber, le **Rédacteur en Chef**, un type super. J'ai beau chercher, j'trouve aucune saloperie à dire sur lui.

Encore en dessous, y'a les pigistes : ce sont eux qui écrivent tous ces bô zarticles qui vous enchantent chaque mois un peu plus. Par ordre alphabétique, ça nous donne : Pascal Amiable, Bruno Bailluet, François Braün, Jérôme Etienne, Olivier Garrigues, François Lionet, Denis Jarril, Thomas Landspurg, Pierre Philippe Launay, François Lionet, Max, Frédéric Mazué, Philippe Rivaillon, Stéphane Schreiber et Lucien Turski-Marcon.

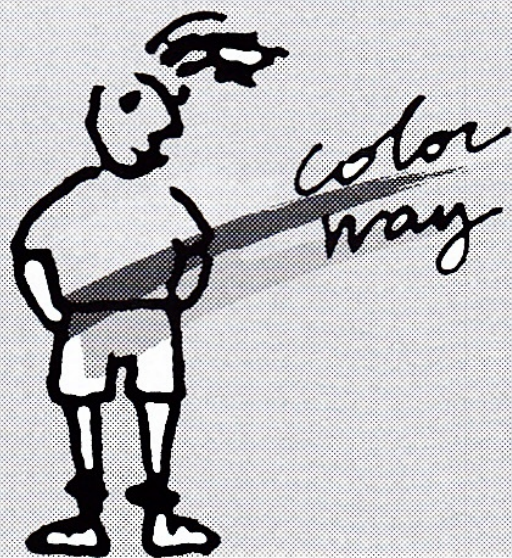
Une fois que les textes sont rendus, Stéphane Schreiber les relit, les censure et corrige les fautes d'orthographe, avec l'aide et le soutien inestimables de Christine Robert, qui a pourtant déjà suffisamment de boulot comme ça avec Amiga Revue.

Après, il faut faire la mise en page. C'est encore Stéphane Schreiber qui s'y colle, on se demande vraiment où il trouve le temps. Quand je vous disais qu'il était super, je n'exagérais pas.

Ensuite, le flashage : à partir des fichiers PostScript générés par PPM, ColorWay (5, rue de la Fidélité, 75010 Paris. Tel : (1) 47.70.87.87) produit des films pour l'imprimeur, lequel se trouve être INO Montrouge (9, rue Auber, 92120 Montrouge. Tel : (1) 46.56.27.69).

Parallèlement à la fabrication, faut gérer tout ça. Par exemple, c'est MCM ANT (16, Quai Jean-Baptiste Clément, 94140 Alfortville. Tel : (1) 43.78.92.10) qui s'occupe des abonnements, de la duplication des disquettes et de l'expédition. Par autre exemple, la voix sensuelle au téléphone, c'est généralement celle de Valérie, et le bruit de l'aspirateur dans le couloir, c'est toujours celui de Madame Maria.

Voilà, vous savez tout, vous pouvez retourner vous coucher. Atchao bonsoir.



Nous l'avons vu, toutes les fonctions et macros que la norme ANSI impose sont regroupées dans une bibliothèque standard et déclarées dans des fichiers d'en-tête, standards eux aussi.

La liste de ces fichiers d'en-tête a été donnée le mois dernier, nous n'y reviendrons donc pas. Rappelons simplement que tout compilateur C se réclamant de la norme ANSI se doit de fournir ces fichiers, ainsi que la bibliothèque elle-même, bien entendu. La suite de cette série sur le C ANSI est donc claire : nous allons détailler chacun de ces fichiers, en expliquant les fonctions, macros et variables qu'il contient.

PETITE PRECISION

Tous les objets que nous allons voir dans ce qui suit, travaillent avec certains types de données, notamment les *int* ou les *float*. D'autres travaillent avec des structures, composées d'objets de plusieurs types.

Le programmeur n'a normalement pas à connaître la taille de ces objets. En d'autres termes, il n'a pas à se soucier de savoir si, sur telle ou telle machine, dans tel ou tel environnement, un *int* est codé sur deux, quatre ou même huit octets.

Si une telle connaissance est souvent primordiale pour accéder aux ressources d'un système donné (par exemple pour l'Amiga, les structures définies par Intuition ou la *graphics.library*), il suffit de savoir que, dans le cas de la bibliothèque standard, telle ou telle fonction renvoie un *int*, un *char*, ou un pointeur quelconque. Si l'on a absolument besoin de connaître la taille d'un *int* (par exemple, pour une allocation mémoire), l'opérateur *sizeof()* doit être utilisé.

Enfin, certains types définis particuliers seront rencontrés au cours de notre exploration de la bibliothèque. Nous les détaillerons à ce moment là.

LES ENTREES-SORTIES : STDIO.H

Nous attaquons par le plus gros morceau... Les fonctions, macros et types d'entrées-sorties (Input-Output, ou plus simplement IO), représentent presque un tiers de la bibliothèque.

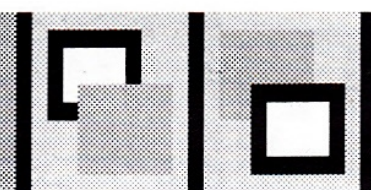
On y trouve notamment la définition des *flux* : il ne s'agit ni plus ni moins que d'un flot de données associées à un périphérique quelconque (disque bien sûr, mais également clavier, écran, imprimante...). Les flux peuvent être gérés soit en mode texte, soit en mode binaire. Dans le premier mode, les données sont organisées en lignes, conclues par le caractère de fin de ligne, '\n'. Ce caractère peut être, de manière interne, représenté soit par CR (ASCII 13), soit par LF (ASCII 10), soit par une combinaison des deux. C'est le système qui gère cela. Un flux binaire est constitué d'une suite d'octets non traités par le système, même s'il devait contenir un ou plusieurs caractères de fin de ligne. C'est le programmeur qui décide d'ouvrir un flux en mode texte ou au contraire en mode binaire.

Au début de l'exécution du programme, le code de démarrage ouvre toujours les trois flux standards *stdin* (entrée), *stdout* (sortie) et *stderr* (sortie d'erreur, peut être différent de *stdout*).

Les fichiers

L'ouverture d'un fichier renvoie un objet de type *FILE*, contenant toutes les informations nécessaires au contrôle du flux. Le type *FILE* est défini ainsi :

```
struct _iobuf
{
    struct _iobuf *_next; /* flux suivant */
    unsigned char *_ptr; /* pointe le buffer IO */
}
```

```
int _rcnt;          /* compteur d'octets en lecture */
int _wcnt;          /* compteur d'octets en écriture */
unsigned char *_base; /* adresse de base du buffer IO */
int _size;          /* taille du buffer IO */
int _flags;         /* flags de contrôle du flux */
int _file;          /* descripteur de fichier */
unsigned char _cbuf; /* buffer pour 1 char. */
}
```

```
typedef struct _iobuf FILE;
```

Les fonctions suivantes concernent le traitement des fichiers.

```
FILE *fopen(const char *filename, const char *mode)
```

fopen() tente d'ouvrir le fichier décrit par *filename*, dans le mode défini par *mode*, et retourne un flux en cas de succès, et *NULL* sinon. *mode* peut être :

- "r" fichier texte en lecture
- "w" fichier texte en écriture. Si le fichier existait déjà, il est écrasé.
- "a" fichier texte en écriture. Si le fichier existait déjà, les nouvelles données lui seront ajoutées.
- "r+" fichier texte en lecture/écriture.
- "w+" fichier texte en lecture/écriture. Si le fichier existait déjà, il est écrasé.
- "a+" fichier texte en lecture/écriture. Si le fichier existait déjà, les nouvelles données lui seront ajoutées.

Si l'on ajoute un b à l'un quelconque de ces modes (par exemple, "rb", ou "w+b"), le flux est ouvert en mode binaire au lieu du mode texte. La longueur des noms de fichiers est limitée à *FILENAME_MAX* caractères et *FOPEN_MAX* fichiers peuvent être ouverts simultanément.

Note : FILENAME_MAX n'est pas défini en Lattice/SAS C 5.10, et FOPEN_MAX est défini sous le nom de _NFILES.

```
FILE *freopen(const char *filename, const char *mode,
FILE *stream)
```

freopen() ouvre le fichier décrit par *filename* dans le mode défini par *mode*, et lui associe le flux *stream*. On utilise en général freopen() pour changer les fichiers associés à *stdin*, *stdout* et *stderr*.

```
int fflush(FILE *stream)
```

fflush() provoque l'écriture anticipée des données encore présentes dans la mémoire tampon du flux *stream*, qui doit avoir été ouvert en écriture. Son effet est indéfini pour un flux ouvert en lecture. Elle retourne *EOF* pour une erreur d'écriture, 0 si tout va bien.

```
int fclose(FILE *stream)
```

fclose() force l'écriture des données encore présentes dans la mémoire tampon du flux *stream*, libère tout tampon alloué automatiquement et ferme le flux, qui devient inutilisable. Elle retourne *EOF* en cas d'erreur, 0 sinon.

```
int remove(const char *filename)
```

remove() détruit le fichier décrit par *filename*. Elle retourne une valeur différente de 0 en cas d'échec (fichier absent, erreur disque...).

```
int rename(const char *oldname, const char *newname)
```

rename() change le nom du fichier *oldname* en *newname*. Elle retourne une valeur différente de 0 en cas d'échec.

```
FILE *tmpfile(void)
```

tmpfile() crée un fichier temporaire dans le mode "wb+", qui sera automatiquement détruit lors de sa fermeture ou lors de la fin normale du programme.

Note : cette fonction n'est pas définie dans le stdio.h du Lattice/SAS C 5.10.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
```

setvbuf() permet de contrôler la mise en mémoire tampon du flux *stream*. Il faut l'appeler avant la première lecture ou écriture sur le flux. Le mode *_IOFBF* provoque un tamponnage complet, *_IOLBF* un tamponnage par ligne de texte et *_IONBF* ne provoque pas de tamponnage. Si *buf* est différent de *NULL*, il sera utilisé comme tampon, sinon il sera alloué. *size* détermine la taille du tampon. setvbuf() retourne une valeur différente de 0 en cas d'erreur.

Note : le type size_t est défini comme typedef unsigned int size_t.

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)
```

fread() lit sur le flux *stream*, *nobj* objets de taille *size* et les place dans le tampon *ptr*. Elle retourne le nombre d'objets effectivement lus, qui peut être inférieur au nombre demandé.

```
size_t fwrite(void *ptr, size_t size, size_t nobj, FILE *stream)
```

fwrite() écrit dans le flux *stream*, *nobj* objets de taille *size* depuis le tampon *ptr*. Elle retourne le nombre d'objets effectivement écrits, qui peut être inférieur au nombre demandé.

```
int fseek(FILE *stream, long offset, int origin)
```

fseek() positionne le pointeur de fichier pour le flux *stream*. Pour un fichier binaire, la nouvelle position est fixée à *offset* octets du début si *origin* vaut *SEEK_SET*, de la fin si *origin* vaut *SEEK_END*, ou de la position courante si *origin* vaut *SEEK_CUR*. Pour un fichier texte, *offset* doit valoir 0 (ou une valeur retournée par ftell()) et *origin* doit valoir *SEEK_SET*. fseek() retourne une valeur non nulle en cas d'erreur.

```
int ftell(FILE *stream)
```

ftell() retourne la position courante dans le fichier pour le flux *stream*, ou -1L en cas d'erreur.

```
void rewind(FILE *stream)
```

rewind() replace le pointeur de fichier au début du flux *stream* et ré-initialise toute erreur détectée. rewind(fp) est l'équivalent de fseek(fp, 0, SEEK_SET); clearerr(fp);

```
int fgetpos(FILE *stream, fpos_t *ptr)
```

fgetpos() place dans **ptr* la position courante dans *stream*. Elle retourne une valeur non nulle en cas d'erreur.

Note : le type fpos_t est défini comme typedef unsigned long fpos_t.

```
int fsetpos(FILE *stream, const fpos_t *ptr)
```

fsetpos() déplace le pointeur de fichier de *stream* à la position mémorisée dans **ptr* par fgetpos(). fsetpos() retourne une valeur non nulle en cas d'erreur.

```
void clearerr(FILE *stream)
```

clearerr() remet à 0 les indicateurs de fin de fichier et d'erreur du flux *stream*.

```
int feof(FILE *stream)
```

feof() retourne une valeur non nulle si l'indicateur de fin de fichier (*EOF*) est positionné pour le flux *stream*.

A suivre...

Par Denis Jarril



Chers amos, bienvenue dans la rubrique Amis... Une page, deux petits programmes très intéressants, une Daisy toujours aussi imbue d'elle-même : la routine, quoi.

Lancer des programmes externes en AMOS... Encore ? vous exclamez-vous. Oui, je vous propose encore une procédure pour faire démarrer un programme externe à partir d'AMOS. Celle-ci marche à tous les coups : même lorsque le Workbench est fermé, même lorsque qu'aucun CLI n'est ouvert, même lorsque l'Amiga est éteint... Non, quand même, il ne faut pas exagérer !

Avant de lancer le programme, la procédure ouvre le device nul (NIL:) en entrée/sortie, grâce à l'instruction AmigaDOS "Open". Ce device est une véritable poubelle faisant disparaître toutes les données qui lui sont envoyées. Son fonctionnement en tant qu'entrée est très simple : il ne renvoie rien du tout.

En communiquant l'adresse de ce device lors du lancement du programme, on évite les plantages : AmigaDOS a quelque chose où envoyer ses données, le programme est lancé correctement.

Une fois la commande partie, on peut fermer le canal.

```

-----
' AMOS Exec: lancer un programme
' externe en AMOS
' >>> ca marche à tous les coups!
' F.Lionet / A.N.T.
-----
' On lance l'AMOS à partir de l'AMOS. Haha.
EXEC["AMOS1.3"]
Print Param
'
Procédure EXEC[C$]
' Ouvre un canal d'entree et de sortie NUL
A$="NIL:"+Chr$(0)
Dreg(1)=Varptr(A$) : Dreg(2)=1004
HAND=Doscall(-30)
'
' Si ouverture réussie
If HAND
' On lance la commande
C$=C$+Chr$(0)
Dreg(1)=Varptr(C$) : Dreg(2)=HAND : Dreg(3)=HAND
F=Doscall(-222)
'
' On peut fermer le canal maintenant
Dreg(1)=HAND
A=Doscall(-36)
End If
'
' Retourne -1 si réussi
End Proc[F]

```

UN ENREGISTREUR MINITEL

Grande nouvelle : Daisy et moi-même allons nous réconcilier et travailler ensemble sur une série d'articles portant sur la réalisation d'un émulateur minitel, en AMOS bien sûr. Il va falloir que je fasse des concessions à cet animal débile.

Nous allons vous proposer une complète émulation, avec possibilité d'enregistrement de pages, macros, et téléchargement. Avec des graphiques et des couleurs en plus. Ce qui nous permettra de voir comment il fonctionne, ce minitel.

Pour vous donner un avant-goût de ces articles, voici un tout petit programme enregistrant et restituant des pages minitel. Il fonctionne avec l'extension série de l'AMOS 1.3.

Connectez-vous sur votre service favori (au hasard, 3615 ANT ou 3615 ComRev), faites fonctionner le programme et choisissez

(E)nregistrement. Faites défiler une page sur le minitel, puis pressez une touche sur l'Amiga et sauvez votre fichier sur disque. Vous pourrez relire la page sauvée avec l'option (L)ecture.

Vous constaterez en regardant le listing, que ce programme n'a vraiment rien de sorcier : il ne fait que bêtement poker les caractères reçus dans une banque, et la sauver sous forme binaire.

Le minitel c'est tout simple, et c'est rigolo. Rendez-vous le mois prochain, où nous allons émuler comme des fous !

```

-----
' Mini Enregistreur/Lecteur de page minitel
' F.Lionet / ANT 1992
-----
' Ouvre un écran "sérieux"
Screen Open 0,640,200,4,Hires
Colour 1,0
'
' Banque servant de buffer
Reserve As Work 10,100000
'
' Ouvre l'interface série. Les paramètres par
' par défaut sont ceux du minitel
Serial Open 0,1
'
' Boucle de prise d'ordre
Do
Input "(E)nregistrement, (L)ecture, (F)in ?";A$
A$=Upper$(A$)
If A$="F" : Exit : End If
If A$="E" : ENREGISTREMENT : End If
If A$="L" : LECTURE : End If
Loop
'
Serial Close
Edit
'
Procédure LECTURE
'
' Lecture du fichier .Nitel
'
N$=Fsel$("*.*Nitel","", "Entrez le nom du fichier","")
If N$<>" "
' Trouve la longueur
Open In 1,N$ : LF=Lof(1) : Close 1
' Charge dans la banque
Bload N$,Start(10)
' Envoie le fichier
Serial Out 0,Start(10),LF
Print "Envoi en cours!"
End If
'
End Proc
'
Procédure ENREGISTREMENT
'
' Enregistrement d'un fichier
'
Print "Enregistrement en cours!"
ABANK=0
' Boucle d'enregistrement
Repeat
' Demande le caractère, sans attendre
A=Serial Get(0)
' Si un caractère est disponible
If A>0
' Si l'on est pas au bout de la banque
If ABANK<Length(10)
' On le sauve
Poke Start(10)+ABANK,A : Inc ABANK
Print At(0,);"Caractères reçus:";ABANK;
Else
' Au bout de la banque!
Bell
End If
End If
' Jusqu'à ce que l'on appuie sur une touche
A$=Inkey$
Until A$<>" "
Print
'
' Quelque chose à sauver?
If ABANK-Start(10)
' Demande le nom
N$=Fsel$("*.*Nitel","", "Entrez le nom du
fichier","Taille:"+Str$(ABANK))
' Sauve un simple fichier binaire.
If N$<>" "
Bsave N$,Start(10) To Start(10)+ABANK
End If
End If
'
End Proc

```

Par François Lionet

Daisy au micro, pour encore une fois relever cette rubrique. Comment ça, nous réconcilier ? Pas question ! Ou alors, double ration de Royal-Canin. Ha ! Je ne vais pas me laisser faire...

Il est possible, vous le savez, d'afficher en AMOS des dégradés de couleurs, les *Rainbows*, ou arcs-en-ciel. Le problème est que l'on ne peut pas avoir plus d'une couleur par ligne. Pourquoi cette limitation ? Je vais vous le dire : c'est l'autre idiot qui en a fait la programmation. Alors aujourd'hui, je vais essayer de le rattraper un peu, avec le magnifique programme qui suit.

Multi_Rainbow recrée une liste Copper à partir de la liste originale de l'AMOS, en mode *Copper Off*. Il recopie la liste en insérant les nouvelles couleurs.

Vous pouvez ainsi fabriquer votre affichage, en chargeant ou en ouvrant des écrans. Lorsque tout est prêt, vous lancez la fabrication de la liste.

Le nombre de *Rainbows* est défini au début du programme, il n'est limité que par la taille de la liste Copper (11 Ko) : on peut avoir environ 6 arcs-en-ciel en même temps, et plus en augmentant la taille mémoire affectée au Copper à l'aide du programme de configuration.

Les couleurs se trouvent dans des datas à la fin du programme. Chaque groupe de données contient les valeurs de couleurs pour chaque ligne. Un label de la forme RNx, ou "x" est le numéro du *Rainbow*, se trouve au début de chaque liste.

Vous pouvez taper les datas à la main, bon courage. Le mieux est d'utiliser le programme domaine public *Rainbow-Warrior* : il sauve des fichiers de datas immédiatement utilisables dans ce programme. Comme je suis bonne patte, j'ai envoyé *Rainbow-Warrior* à Stéphane. Il sera mis très rapidement en téléchargement sur le 3615 ANT (le programme, pas Stéphane).

Allez, une lèchouille et à bientôt. *Shuuuuurp !*

```
'-----
' Un nouveau miracle signé Daisy:
' Multi-Rainbows. Plein d'arc-en-ciel
' sur un meme ecran!
' (c) Daisy Lionet / ANT 1992
'-----
' Nombre de Rainbow à traiter
NRAIN=3
' Dimensionne les tableaux de couleurs
Dim C(NRAIN),CC(NRAIN,279)
' Affectation des Rainbows aux couleurs
C(0)=0 : C(1)=1 : C(2)=2
' On lis les datas produits par Rainbow-Warrior
For N=0 To NRAIN-1
  Restore "Rn"+Mid$(Str$(N),2)
  For C=0 To 279
    Read CC(C(N),C)
  Next
Next
'-----
' Ici, on fait son propre affichage. Autant d'ecrans
' que l'on veut.
Screen Open 0,320,256,16,Lowres
Cls 1
Paper 1 : Pen 5 : For X=0 To 205 : Print "Daisy!"; : Next
Cls 2,64,70 To 320-64,170
Screen Open 1,640,32,4,Hires
Screen Display 1,,70,,
Cls 0 : Paper 0 : Pen 3 : For X=0 To 25 : Print "Ecran HIRES!"; :
Next
'
' Penser à enlever la souris!
Hide On
'-----
' On s'assure que les ecrans sont affichés.
Wait Vbl : View : Wait Vbl : View : Wait Vbl
'
' On recupere les listes coppers dans une banque
Reserve As Work 10,11*1024
```

```
L=0
Repeat
  C=Leek(Cop Logic+L)
  Loke Start(10)+L,C
  L=L+4
Until C=$FFFFFFFE
'
' A pus copper
Copper Off : Wait Vbl
' Adresses ou poker
ACOP=Cop Logic : ACH=Start(10)
'
' Met les definitions des sprites
For N=0 To 15
  Cop Move 0,0
  Loke ACOP,Leek(ACH)
  Add ACOP,4 : Add ACH,4
Next
'
' Boucle de creation de la liste
Y=30 : YRAIN=0
Do
  ' Attente du debut de ligne
  Cop Wait 7,Y : ACOP=ACOP+4
  '
  ' Adresse dans la liste
  CY=Deek(ACH)/256+YBASE
  '
  If Y=CY
    ' C'est le debut d'un ecran
    ACH=ACH+4
    C1=Deek(ACH)
    If (C1 and 1)=0
      ' On copie betement la liste originale
      Do
        C1=Deek(ACH) : C2=Deek(ACH+2)
        Exit If (C1 and 1)<>0
        ' Sauf si on change l'une des couleurs
        ' du rainbow
        For C=0 To NRAIN-1
          If C1=C(C)*2+$180
            C2=CC(C(C),YRAIN)
          End If
        Next
        Add ACH,4
        Cop Move 0,0
        Doke ACOP,C1 : Doke ACOP+2,C2 : Add ACOP,4
      Loop
    End If
  Else
    ' Ce n'est pas un ecran, on met
    ' simplement les couleurs
    For C=0 To NRAIN-1
      Cop Move $180+C(C)*2,CC(C(C),YRAIN) : Add ACOP,4
    Next
  End If

  Inc Y : Inc YRAIN
  If Y=256 : YBASE=256 : End If
'
Exit If Y>311
Exit If YRAIN>279
Exit If ACOP-Cop Logic>10*1024
Loop
'
' La fin de la liste
Cop Wait 255,255
'
' On affiche!
Cop Swap : Wait Vbl : Wait Key
Copper On : Wait Vbl
'
' Et voila
Edit
'-----
' Définition des Rainbows
' Integrer les liste de Rainbow-Warrior, précédées de
' labels RNx, ou X est le numéro du Rainbow...
RN0:
Data $100,$210,$320,$430,$540,$650,$761,$872
... Pleins de Datas dont je vous fait grace
'
' Deuxieme rainbow
RN1:
Data $BEE,$CC,$CC,$CC,$BEE,$CC,$CC,$CC
... Meme remarque!
'
' Troisieme rainbow
RN2:
Data $AF,$AF,$AF,$AF,$AF,$AF,$AF,$AF
... Commandez la disquette ANT!
```

Par Daisy Lionet

AREXX : LE DEVERMINAGE (II)

Nous avons vu comment on pouvait suivre, et éventuellement mémoriser, le déroulement d'un programme afin d'en détecter les défauts divers. Il reste encore quelques possibilités intéressantes, que nous allons analyser.

Nous avons précédemment laissé de côté le traçage interactif, non pour le négliger, bien au contraire : il permet de modifier momentanément le programme en cours (valeur ou instruction). L'intérêt en est considérable et la mise en oeuvre très simple. Il y a deux façons d'entrer dans le mode interactif :

- à partir de l'intérieur du programme, en ajoutant un ? à l'instruction **TRACE** en complément avec n'importe quelle option. Exemple **TRACE ?** ou encore **TRACE ? INTERMEDIATES**. Attention à la place relative de l'option et du ?. La commande du mode interactif agit en mode alterné (toggle) : **TRACE ?** lance le mode interactif, puis **TRACE ?** à nouveau l'arrête, et ainsi de suite.
- à partir de l'extérieur, en lançant le programme **TS** (fourni avec AREXX). Dans ce cas, tous les programmes AREXX en cours entrent en mode interactif. **TS** (Trace Start) a son opposé **TE** (Trace End), qui permet de sortir du mode interactif.

Le mode interactif arrête le programme après chaque instruction AREXX sauf pour **CALL**, **DO**, **ELSE**, **IF**, **THEN** et **OTHERWISE** qui pourraient poser des problèmes à l'interpréteur. Pour ces instructions, l'arrêt est effectué à la fin de l'exécution totale de l'instruction. En outre l'interpréteur ne s'arrêtera pas après une instruction qui aurait déclenché une erreur, son traitement étant prioritaire.

Lorsque l'interpréteur est arrêté par le mode interactif, quel qu'en ait été le mode de lancement, le symbole **+++** apparaît dans la fenêtre de traçage. A partir de cet instant, trois actions peuvent être réalisées :

- une ligne vide, sanctionnée par un simple appui sur la touche retour, permet au programme de continuer son exécution jusqu'au point d'arrêt suivant.
- un signe = suivi de l'appui sur la touche retour permet à l'instruction précédant l'arrêt d'être exécutée une nouvelle fois.
- l'entrée d'une instruction valide quelconque, qui est exécutée tout de suite après l'appui sur la touche retour. Comme il a été vu précédemment, les instructions de type **DO** ou **SELECT** doivent être complétées avant l'appui sur la touche retour pour ne pas plonger l'interpréteur dans une profonde angoisse qui l'inciterait à faire méditer le GURU.

TRAITEMENT DES ERREURS

Heureusement, AREXX procure un traitement d'erreurs spécial pendant qu'il fonctionne en mode interactif, qui fournit un rapport pour chaque erreur mais ne provoque pas l'arrêt du programme (ce qui en serait normalement la conséquence). Cette particularité ne s'applique qu'aux instructions qui ont été entrées en mode interactif (ouf!). Les erreurs dues aux défauts du programme sont traitées par la méthode normale.

En plus de ce traitement spécial et en mode interactif uniquement, l'interpréteur désactive les indicateurs internes d'interruption. C'est indispensable pour éviter le transfert vers le traitement d'erreur éventuellement intégré au programme (voir chapitre suivant). Une exception cependant pour les instructions de type **SIGNAL** qui continuent non seulement à autoriser le transfert mais qui font abandonner ce qui pouvait rester comme entrée interactive en suspens.

INTERRUPTIONS

AREXX entretient un système interne d'interruption qui peut être utilisé pour détecter et piéger certaines conditions d'erreur. Lorsqu'une interruption est activée (mise en état de fonctionner) et que les conditions de déclenchement correspondantes se présentent, le programme est dévié vers un sous-programme dont le nom est spécifique de l'interruption, pour peu qu'on ait pris soin d'en écrire un.

Les conditions d'interruption peuvent être synchrones, telles que les erreurs de syntaxe, ou asynchrones, telles que les demandes au clavier, **CONTROL-C** par exemple. Il faut noter que les interruptions extérieures sont complètement séparées des interruptions "hardware" gérées par Exec.

Liste des interruptions reconnues

Certaines des interruptions ont des comportements différents selon qu'elles sont activées ou non, d'autres n'ont qu'un type de comportement efficace seulement après activation.

- **BREAK_C** : réagit à **CONTROL-C** au clavier. Si cette interruption n'est pas activée, le programme se termine immédiatement et le message *'Execution halted'* est présenté à l'opérateur, agrémenté d'un *'error code n°2'*.
- **BREAK_D** : réagit à **CONTROL-D** au clavier. Si cette interruption n'est pas activée, elle est ignorée.
- **BREAK_E** : réagit à **CONTROL-E** au clavier. Si cette interruption n'est pas activée, elle est ignorée.
- **BREAK_F** : réagit à **CONTROL-F** au clavier. Si cette interruption n'est pas activée, elle est ignorée.
- **ERROR** : cette interruption est générée par les commandes de l'hôte (ici, le système) qui retournent un code non zéro traduisant une impossibilité quelconque (voir ci-après).
- **HALT** : si elle est activée, cette interruption piège une requête d'arrêt extérieure sinon, le programme se termine immédiatement et le message *'Execution halted'* est présenté à l'opérateur, agrémenté d'un *'error code n°2'*.
- **IOERR** : si elle est activée, cette interruption piège les erreurs détectées par le système d'Entrées/Sorties.
- **NOVALUE** : si elle est activée, cette interruption piège les variables non initialisées. On rappelle à cette occasion qu'il n'est pas besoin de déclarer une variable : **DO a=1 to 14** ne demande pas que *a* soit déclaré d'une façon quelconque auparavant mais **a=a+3** n'est légal que si *a* a déjà une valeur, bien évidemment.
- **SYNTAX** : cette interruption piège les erreurs de syntaxe ou d'exécution. Pas toutes cependant, en particulier les erreurs arrivant avant l'exécution réelle du programme ou celles détectées par l'interface externe de AREXX ne sont pas traitées.

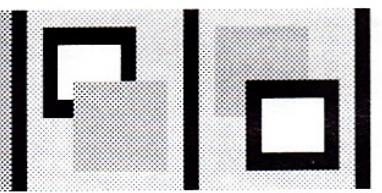
Qu'on se rassure (mais est-ce bien nécessaire ?) lorsque qu'une interruption est prise en compte, tous les contrôles en cours actifs et aussi l'interruption qui vient de se déclencher sont désactivés. Ceci pour prévenir d'éventuelles boucles inextricables. Cependant, cette disposition n'affecte que l'environnement courant, et pas l'environnement général.

UTILISATION DES INTERRUPTIONS

Nous venons de voir que les interruptions ont un comportement différent suivant qu'elles sont activées ou non. Le comportement non activé ayant été décrit, voyons comment les utiliser en mode activé.

Il faut réaliser deux opérations : activer l'interruption souhaitée et écrire le sous-programme de traitement.

La forme générale d'activation est **SIGNAL ON label**, où *label* est le nom d'une des interruptions décrites plus haut. Par exemple, **SIGNAL ON NOVALUE** s'inquiètera dès que le programme tentera d'utiliser une variable non initialisée. A l'arrivée de la condition, le déroulement du



programme sera dévié vers un sous-programme de traitement de cette interruption. L'instruction **SIGNAL ON** est semblable au **GOTO** prohibé de tous les programmes respectables. Dans le cas présent, et ailleurs aussi, parce que c'est une nécessité, il faut réaliser un transfert impératif, d'où cette forme. Le sous-programme intéressé se nommera donc **label**: (dans le cas de l'exemple, **NOVALUE**:). A partir de l'entrée dans le sous-programme on revient à de la programmation classique. On peut ainsi donner une valeur arbitraire à la variable non initialisée et retourner à la suite du programme, ou encore solliciter l'opérateur avec un texte lui demandant une valeur de son choix et utiliser la capacité du **PARSE** (**PARSE PULL** ou tout simplement la forme abrégée **PULL**) et retourner au programme pour continuer. Le choix du retour et à la continuation du programme dépend de la volonté et du besoin du programmeur.

Enfin, pour désactiver **SIGNAL ON label** on utilisera son contraire, **SIGNAL OFF label**, ce qui permet une jolie série de combinaisons **ON/OFF** de toutes les formes possibles d'interruption, sélective en forme, en activation, et donc en position et en répétition.

Signalons (sans jeu de mots...) une forme un peu plus complexe de **SIGNAL**: **SIGNAL VALUE expression**, qui agit comme un **GOTO** calculé c'est-à-dire qu'il y a transfert vers un sous-programme de traitement d'erreur qui dépend de l'évaluation de l'expression. Le type de l'interruption est défini par une variable spécifique dont nous allons nous occuper maintenant.

VARIABLES "SIGL" et "RC"

Ces deux variables ont des noms réservés, au même titre que les mots spécifiques du langage: **DO**, **PARSE** et tous les autres. Elles sont constamment mises à jour pendant l'exécution du programme et peuvent être consultées à tout moment.

La variable **SIGL** (pour **SIGnaL**) contient le numéro de la ligne qui précède celle qui a déclenché le transfert vers le traitement d'erreur. Attention, si vous avez utilisé la faculté légale d'écrire plusieurs instructions sur la même ligne, séparées par un point-virgule, vous risquez de ne pas retrouver aisément celle qui est la cause de l'interruption.

La variable **RC** (pour **Return Code**), contient, elle, le résultat des interruptions **ERROR** et **SYNTAX** en cas d'erreur et une autre valeur en cas de bon fonctionnement. Pour **ERROR**, le résultat usuel est le niveau de sévérité de l'erreur et il peut venir de l'extérieur. Pour **SYNTAX**, le résultat est toujours un code erreur **ARexx**. Par exemple, si l'erreur est la numéro 3 et la sévérité 20, on en déduit qu'il y a une quantité de mémoire insuffisante en se référant à la liste des codes d'erreurs fournie avec la documentation d'**ARexx**.

La variable **RC** n'est pas toujours d'un maniement aussi limpide, surtout lorsqu'elle est affectée par un élément extérieur à **ARexx**. En principe, tout appel à une fonction extérieure, de l'OS par exemple, devrait donner une valeur au **RC** et, si cela est réputé vrai avec l'OS 2 de l'**AMIGA 3000** (**ARexx** fait partie de son environnement) cela ne l'est sûrement pas pour les OS 1.1 à 1.4 des "petits" Amigas. L'emploi du **RC** doit donc être soumis à l'essai pour avoir une idée de la vérité. Enfin, avant de donner des exemples d'emploi, il faut avoir en tête que le **RC** est affecté par le retour de l'appel à une fonction et qu'il a de ce fait, une valeur très rapidement caduque.

EXEMPLES D'UTILISATION DE RC

La valeur de retour de **RC** est dite booléenne, c'est-à-dire qu'elle vaut 1 en cas de mauvaise exécution et 0 sinon. C'est en tout cas le fonctionnement qu'elle devrait avoir d'après les livres... Pour se faire une idée, on pourra avec profit utiliser le petit programme suivant qui permet d'effectuer divers essais. Ce programme permet l'entrée au clavier de diverses instructions **ARexx** et l'on pourra regarder avec intérêt ce qui se passe en cas de défaut dans l'orthographe, la syntaxe

ou encore dans le système. Il est prévu trois cas de **RC**: 0, 1, autre.

```

1  /**/
2  PRAGMA('W','NULL')
3  SIGNAL ON BREAK_F
4  DO FOREVER
5      PULL instruction
6      INTERPRET instruction
7  SAY''
8  SAY'Le résultat RC est:'
9      SELECT
10         WHEN rc=0 THEN SAY'Ok...RC=' rc
11         WHEN rc=1 THEN SAY'pas Ok...RC=' rc
12         OTHERWISE      SAY'Autre valeur...RC=' rc
13      END
14  END
15
16  BREAK_F:
17  PRAGMA('w','WorkBench')
18  EXIT

```

En ligne 1, on interdit à l'OS d'envoyer des Requesters selon son habitude en cas de mécontentement (on remet tout en place en ligne 17). En ligne 3, on active l'arrêt sur **CONTROL-F**. De la ligne 4 à la ligne 14, c'est le programme en boucle infinie à cause de la ligne 4. En 5, attente de l'instruction à entrer au clavier et en ligne 6, exécution de ladite instruction avec **INTERPRET**. De 9 à 11, on analyse le résultat dans **RC**. En ligne 16, on trouve le label de branchement de **CONTROL-F**, qui nous sort de la boucle infinie et où l'on fait le ménage.

INSTALLATION

Nous allons maintenant présenter une organisation qui permet d'écrire les lignes de code, de voir le résultat à l'exécution et de surveiller le déroulement dans la fenêtre spécialisée.

On doit trouver les programmes suivants dans le répertoire C: courant: **HI**, **TCO**, **TCC**, **TE**, **TS**, en plus de **RX**, **RXC** et **RXSET**.

On ouvre une console **CONMAN** par exemple (recommandé).

On ouvre l'éditeur de texte de son choix.

On ouvre enfin la fenêtre de messages d'erreur par **TCO**.

Les fenêtres sont réparties de la façon suivante: l'éditeur occupe les 2/3 supérieurs de l'écran, la console le 1/3 inférieur et la fenêtre de surveillance est en haut à droite.

Tout ceci doit se trouver dans la *Startup-Sequence*, comme il est indiqué dans la documentation. De plus, pour éviter de cliquer à chaque fois dans les fenêtres, il est agréable d'installer **DMouse** de Matt Dillon. Enfin, on peut fermer la fenêtre de début qui ne sert plus à rien.

On peut alors taper un programme dans la fenêtre de l'éditeur, le sauvegarder, le faire s'exécuter dans la console... A ce propos, il est recommandé de lancer l'exécution par **RUN RX** plutôt que par **RX** tout court, car en cas de boucle infinie imprévue dans le programme, l'envoi de **HI** permet de tout arrêter... sans devoir re-booter! On se méfiera toutefois des programmes qui utilisent **PARSE PULL** pour demander quelque chose à l'opérateur, car ils lisent d'abord la console et on obtiendrait une erreur car ils liront le retour de **RUN**. Pour ceux-là, si l'on ne veut pas changer de méthode d'appel, on placera une condition d'arrêt interne et on ne les lancera pas avec **RUN**. On lancera de la même façon le traçage interactif et toutes les options que nous avons analysées.

Par François Gueugnon 

PASCAL : PRESENTATIONS (II)

De nouveau du Pascal ! Ceux qui ont cru que le simple article du mois dernier expliquait tout du Pascal se trompaient lourdement.

L'article d'aujourd'hui se décompose en deux parties : la première sera essentiellement théorique et portera sur l'architecture des blocs. La seconde continuera l'exploration du compilateur PCQ et sera beaucoup plus pratique (NDLR : rappelons simplement que le compilateur PCQ peut être trouvé sur la disquette ANT numéro 22 et que vous pouvez toujours la commander en utilisant le bon page 32. Fin de la pub).

LES BLOCS

Le Pascal est un langage structuré : tout ce qui le compose obéit à une forme syntaxique. L'architecture principale d'un programme est basée sur le bloc (nous en avons vu un le mois dernier). Il existe différentes sortes de blocs qui disposent chacune de propriétés propres.

La forme générale d'un bloc est un en-tête, suivi d'une zone de déclarations locales et d'un segment d'instructions. L'en-tête se divise lui-même en un mot introductif qui indique le type de bloc (à ne pas confondre avec les types de variables), suivi d'un identificateur qui sera le nom donné au bloc pour de futures utilisations et, éventuellement, d'autres informations comme les passages de paramètres (ordre et sens). Il se termine par un point-virgule.

La déclaration locale est la zone qui permet de définir de nouveaux objets (par l'intermédiaire d'autres blocs) connus exclusivement du bloc courant. Le segment d'instructions, lui, est toujours le couple BEGIN... END.

Tous les blocs ont nécessairement un en-tête. Le premier mot d'un bloc (et de son en-tête) est celui qui définira son type.

- **PROGRAM** indique, je l'ai déjà expliqué, le bloc programme. Il ne doit apparaître qu'une seule fois dans tout le listing. Dans le fichier qui le contient, il doit être le plus grand bloc, c'est-à-dire que tous les autres blocs doivent se trouver dans sa zone de déclarations locales.

- **EXTERNAL** définit un bloc externe qui permet de ne pas associer des procédures et des fonctions à un programme sans créer d'erreur de compilation. Le fichier objet obtenu est nommé une "bibliothèque", qu'il ne faut pas confondre avec les ".library" d'Exec qui, elles, doivent être ouvertes pour pouvoir être utilisées. Celles construites à l'aide d'un bloc externe sont utilisées avec le linker évitant la compilation répétée d'une partie d'un programme. Je ne cite ici ce bloc que pour des raisons d'exhaustivité, nous l'étudierons en détails dans un autre article.

Les points communs de ces deux blocs, sont qu'il sont uniques dans un fichier, qu'ils ne peuvent se contenir mutuellement, et qu'ils contiennent tous les autres blocs dans leur zone de déclarations locales (j'insiste car ceci est très important). A l'opposé d'un bloc programme, le bloc externe ne possède pas de nom, et n'a pas de segment d'instructions.

- **VAR** (que nous avons déjà vu), permet de déclarer un bloc (ou plusieurs blocs) de déclarations de variables.

- **CONST** déclare des constantes (qui ne changent pas de valeur tout le long du programme) mais aussi, des variables qui possèdent une valeur par défaut. Pourquoi avoir mis ce type de variables dans "constantes" ? Afin de respecter une décision de Borland dans le Turbo Pascal (il en est à la version 6) tenant de la syntaxe de la déclaration. Nous y reviendrons, alors patience.

- **TYPE** permet de construire des types nouveaux avec les quelques types de bases et des constructeurs de types tels que ARRAY, RECORD

(see later, pour ne pas se répéter). Le point commun de ces trois blocs est qu'ils ne sont composés que d'un en-tête ; ils n'ont ni zone de déclarations locales, ni segment d'instructions.

- **PROCEDURE** et **FUNCTION** sont les déclarateurs de blocs procéduraux. Le premier déclare une procédure et l'autre une fonction (avouez que vous ne l'aviez pas trouvé). Une fonction est simplement une procédure qui renvoie un objet. Il faut donc préciser son type. Nous verrons comment il faut s'y prendre. Ce type de blocs possède les trois zones : un en-tête (pour définir son nom, l'ordre des arguments et leur type pour l'appel), une zone de déclarations locales, et un segment d'instructions.

En fait, nous avons six en-têtes différents pour cinq types de blocs. Ce qu'il faut bien comprendre est que cette structuration permet de créer des blocs dans d'autres blocs, comme des boîtes qui contiendraient d'autres boîtes plus petites. Ainsi, certaines de ces boîtes nous seraient cachées ou accessibles suivant l'endroit où l'on travaille (effet de localité).

Un exemple, toujours très formel (ce n'est pas la partie théorique pour rien), pour ceux qui seraient perdu :

```
PROGRAM Vue_Des_Blocs;           { >> début du bloc programme }

VAR                               { >> début du bloc
déclaration }
  <noms de variable>:<type défini>;
  <autres des déclarations>
                                { >> fin du bloc déclaration }

PROCEDURE <nom>( <déclarations des paramètres> );
  { >> début du bloc procédural }
TYPE  { >> début du bloc déclaration interne }
  <nom de type> = <types construits>
  { >> fin du bloc déclaration interne }
VAR   { >> début d'un autre bloc déclaration interne }
  <autres des déclarations>
  { >> fin du bloc déclaration interne }
BEGIN { début du segment d'instructions du bloc procédural }
  <instructions>
END;  { >> fin du bloc procédural (et du segment) }

                                { >> début d'un nouveau bloc procédural }
FUNCTION <nom> ( <déclarations des paramètres> ): <type>;

TYPE  { >> début du bloc déclaration interne }
  <nom de type> = <types construits>
  { >> fin du bloc déclaration interne }
VAR   { >> début d'un autre bloc déclaration interne }
  <autres des déclarations>
  { >> fin du bloc déclaration interne }
BEGIN { début du segment d'instructions du bloc procédural }
  <instructions>
END;  { >> fin du bloc procédural (et du segment) }

BEGIN { >> début du segment d'instructions du bloc principal }
  <instructions>
END.  { >> fin du segment et bloc programme }
```

Remarquez que dans cet exemple, j'utilise le terme interne pour bien différencier la zone de déclarations locales du programme et les zones de déclarations locales de chaque bloc procédural.

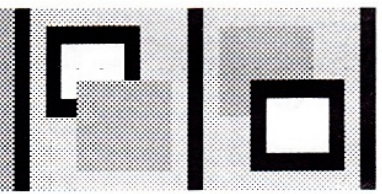
LA PORTEE

Tout objet qui a un nom possède une portée qui lui est associée, c'est-à-dire qu'à un certain moment, on ne pourra plus l'utiliser. Pour qu'un objet (variable, type, procédure ou fonction) soit défini dans la zone où l'on veut le référencer (en général, dans un segment d'instruction), il faut non seulement que sa déclaration soit située plus haut dans le texte, mais aussi dans un bloc qui contient (par emboîtements successifs) le bloc où l'on travaille. Un petit exemple, commenté au delà du raisonnable, ne fera pas de mal.

```
PROGRAM Portee; { en-tête du bloc programme et début du fichier }
{ ce programme n'a pas de signification réelle mais il est là
pour aider à la compréhension du problème de la portée }

var i,j:integer;
  { i et j sont des entiers }

PROCEDURE EcrUnEntier();
{déclaration d'une procédure. Le bloc procédural}
{ a comme nom ECRITUNENTIER }
```

```

var n,l:integer; { on ne peut les appeler i et j car i et j
sont }
    { accessible : c'est un effet de bord... }
    { nous avons un bloc de déclaration interne au }
    { bloc EcrireUnEntier }

procedure InterneAEcrit(); { ce bloc connaît i,j,n et l comme
variables}
    { il est connu dans tout EcrireUnEntier mais pas dans }
    { le bloc programme }
    begin
        n:=n*12; { segment d'instructions de InterneAEcrit }
        { n prend douze fois son ancienne valeur }
        l:=l+1;
    { Je fais une référence aux variables n et l, qui ne sont
    ni des paramètres ni des variables locales de InterneAEcrit.
    Je me sers d'un effet de bord qui ne faudra pas utiliser sauf
    pour des constantes }

        end; { fin du segment d'instructions de InterneAEcrit }

    begin { segment d'instructions de EcrireUnEntier }

        l:=j; { même remarque que ci-dessus: un effet de bord qui
mène a }
        n:=i*j; { l'erreur assez facilement }
        InterneAEcrit; { appel de la procédure définie plus haut }
        writeln('vive le Pascal, \n un chiffre ',n);
    { je suis sûre que vous ne vous y attendiez pas ! }
    { ligne d'écriture du nombre n et repassage à la ligne.}
        end; { fin du segment d'instructions de EcrireUnEntier }

BEGIN { début du segment d'instructions du bloc programme }
    i:=1;
    j:=i+2;
    { seules ces deux variables sont accessibles tout les autres ne le
sont plus. Seule, aussi la procédure EcrireUnEntier est
accessible; l'InterneAEcrit ne l'est plus du fait que l'on
ne travaille plus dans le bloc EcrireUnEntier (comme si on avait
fermé le couvercle de la boîte. }

    EcrireUnEntier { appel de la procédure EcrireUnEntier }
END. { fin du bloc programme et fin du fichier }

```

NOUVELLES PROCEDURES

Bon, finie la théorie, passons à la pratique. Je vous avais laissé un petit devoir (tiens, plus personne ne me regarde...). La spécification était : "saisie de l'âge d'une personne, ainsi que l'année de sa naissance sur l'entrée standard pour calculer approximativement l'année actuelle, le tout très familièrement". Le but de ce programme était de vous présenter mister **READLN**, dont voici la fiche signalétique :

READLN()

syntaxe : **READLN (var1, var2, ..., varN)**
 type : procédure multiparamètres
 spécification : identique à celle de **READ**, mais avec un retour-chariot après saisie de toutes les variables paramètres.

Pour comprendre **READLN**, il faut connaître **READ**.

READ()

syntaxe : **READ (var1, var2, ..., varN)**
 type : procédure multiparamètres
 paramètres : variables du type integer, real, string ou array of char.
 spécification : lit sur l'entrée standard (généralement le clavier) ou, si var1 est du type file, dans le fichier désigné par var1, une séquence de caractères, puis fait la conversion suivant le type de la variable en cours de saisie. L'ordre de saisie est de la gauche vers la droite, et il faut séparer les entrées par des espaces. Enfin, un écho à l'écran est fait si la saisie passe par le clavier.

Vous pouvez observer que d'autres types de variables peuvent être utilisés avec **READ()** ou **READLN()**. Ceux-ci seront expliqués plus tard.

L'implémentation fournie par PCQ pour **READ()** n'est pas tout-à-fait standard. **READ()** demande toujours que tous ses arguments soient séparés par un blanc, d'où certains effets indésirables, que montre l'exemple suivant.

```

program Read_Essai;
{ saisie sur l'entrée standard de deux nombres entiers }
{ sur la même ligne }

var i,j:integer ;

BEGIN
    writeln('Veuillez taper deux nombres entier :');
    write('premier nombre :');
    read(i);
    write('second nombre :');
    read(j);
    writeln('merci. Les nombres sont :',i,',',j);
END.

```

Pour pouvoir saisir les deux nombres, il faut les taper l'un derrière l'autre, séparés par un blanc. Par contre, l'implémentation de **READLN()** est respectée, comme le montre ce petit programme :

```

program Readln_Essai;
{ saisie sur l'entrée standard de deux nombres entiers }
{ sur deux lignes }

var i,j:integer ;

BEGIN
    writeln('Veuillez taper deux nombres entier :');
    write('premier nombre :');
    readln(i);
    write('second nombre :');
    readln(j);
    writeln('merci. Les nombres sont :',i,',',j);
END.

```

Alors, **READ()** ou **READLN()** ? Pour ne pas avoir de surprises, je vous conseille **READLN()** quand vous utilisez PCQ. Pour ceux qui utilisent d'autres compilateurs Pascal, testez-les avec le programme ci-dessus et faites-vous une idée.

Pour mémoire, voici la fiche signalétique des procédures **WRITE()** et **WRITELN()** :

WRITE()

syntaxe : **WRITE (exp1, exp2, ..., expN)**
 type : procédure multiparamètres
 paramètre : expX peut être une expression du type array of char, string, integer, booléen, réel mais aussi des constantes string (comme par exemple "Hello boy") ou array of char (comme 'hello boy')
 spécification : envoi sur la sortie standard (ou si var1 est du type "file of" dans le fichier désigné par var1), les valeurs sous forme de chaînes de caractères, des paramètres restants.

WRITELN()

syntaxe : **WRITELN (exp1, exp2, ..., expN)**
 type : procédure multiparamètres
 paramètres : identiques à ceux de **WRITE()**
 spécification : identique à **WRITE()**, mais avec ajout d'un retour-chariot après tous les paramètres.

Vous vous demandez peut-être à quoi correspond une variable de type *file of*... En Pascal, on désigne un fichier par un nombre qui lui est propre (au même moment, un nombre ne désigne qu'un et un seul fichier). L'entrée et la sortie standard sont aussi considérées comme des fichiers par Pascal (et par AmigaDOS). Une variable de type "file of" peut recevoir ce nombre. Les procédures **WRITE()** et **READ()** sont des procédures d'entrées-sorties générales qui ne se bornent pas au clavier et aux fenêtres (CLI ou Shell) ; elles permettent des "accès disques" à travers les fichiers que l'on aura préalablement ouverts. Mais ceci est une autre histoire.

Le mois prochain, si tout va bien, nous aborderons, pour la partie théorique, les mystérieux types prédéfinis ainsi que les effets de bord et, dans la partie pratique, l'implémentation des chaînes de caractères (y a du boulot !).

Par Olivier Garrigues

GFA : LE TRACKDISK.DEVICE

Il existe plusieurs fonctions très agréables pour lire les fichiers, mais elles sont insuffisantes pour une véritable gestion des secteurs. Il faut alors utiliser les structures et les routines propres au système.

Vous savez que toutes les routines C ont été ré-écrites en GFA 3.52F, aussi nous ne serons pas vraiment dépayés. La démarche est tout-à-fait identique, voire un peu plus simple en GFA : le trackdisk.device utilise bien sûr une structure d'entrée-sortie étendue (IOExtTD). Pour plus de détails, je vous conseille de lire le message des thaïlandaises de Max dans l'ANT 24, ainsi que *la Bible* pages 357, 543 et 687. Voici ce que cela donne pour notre programme :

```
structure IOExtTD
0 structure IOSTdReq
  0 Structure Message
    0 Structure Node
      0 ln_succ Pointeur Noeud suivant LONG%
      4 ln_pred Pointeur Noeud précédent LONG%
      8 ln_type Type NT_MSGPORT BYTE|
      9 ln_pri Priorité dans la liste BYTE|
      10 ln_name Nom de la structure LONG%
    14 mn_ReplyPort Pointeur port réponse LONG%
    18 mn_Lenght Longueur WORD&
  20 io_device Pointeur structure Device LONG%
  24 io_unit Pointeur structure Unité LONG%
  28 io_command Commande WORD&
  30 io_flags Direction de la commande BYTE|
  31 io_error Erreur du device BYTE|
  32 io_actual Nb données transmises LONG%
  36 io_lenght Nb données LONG%
  40 io_data pointeur de données LONG%
  44 io_offset N° du secteur en octet LONG%
  48 iotd_count Nb changements de disk LONG%
  52 iotd_seclabel 16 octets (copyright) LONG%
```

Ainsi, une structure Message est associée à un message ne devant pas dépasser 64 ko. Ce nombre est une limite qui peut avoir ses contraintes.

GFA ET C, MON GENERAL !

Les bibliothèques et les devices ont une logique de base assez semblable, ou tout au moins assez similaire (gasp, j'ai du dire la même chose). Les structures d'unité sont également assez classiques et nous devons seulement savoir que nous disposons ici des unités DF0: à DF3: et que nous écrirons pour cela 0, 1, 2 ou 3 dans l'IO.

- *io_data* contiendra l'adresse où débutent les données à lire. Attention, le système s'avère ici particulièrement ségrégationniste : tout d'abord, celles-ci doivent avoir une taille au minimum égale à 512 octets, ce qui élimine Super OctetoPhage. Le pauvre. Et ensuite on ne pourra les trouver qu'en mémoire Chip (MALLOC).

- *io_offset* doit également toujours être un multiple de 512 octets. Il n'est donc pas possible de se positionner à cheval sur deux secteurs en écrivant par exemple 300.

- *iotd_count* indique le nombre d'insertions et de retraits de disquettes dans le lecteur. Cette valeur augmente de 2 pour un simple échange de disquettes. Or, comme la valeur de référence correspond au dernier réamorçage du système, un nombre impair signalera immanquablement un lecteur vide et un nombre pair, la présence d'une disquette.

- *iotd_seclabel* doit avoir 16 octets d'un seul tenant et est soit réservé aux lettres du copyright, soit rempli avec des 0. SuperOctetoPhage y écrit d'ailleurs un 0 par lettre de son nom et comme il dit, personne ne pensera à les modifier.

TOUJOURS DU C, MON GENERAL !

Il reste à parler de la structure ReplyPort :

```
0 Structure MsgPort
0 Structure Node
  0 ln_succ Pointeur Noeud suivant LONG%
  4 ln_pred Pointeur Noeud précédent LONG%
  8 ln_type Type NT_MSGPORT BYTE|
  9 ln_pri Priorité dans la liste BYTE|
  10 ln_name Nom de la structure LONG%
  14 mp_flags BYTE|
  15 mp_sigbit N° du bit signal envoyé BYTE|
  16 mp_sigtask Pointeur où va le signal LONG%
```

ET ENFIN DU GFA, MON GENERAL !

Tout cela nous permet de savoir où nous allons et grâce aux remarques, le codage se laissera ainsi déchiffrer très facilement.

On trouve d'abord l'adresse de notre propre tâche par la méthode "longue" : *FindTask(0)*. Il existe en effet la possibilité de remplacer cette ligne par un appel direct plus rapide à la structure Exec à l'offset 276, appelé justement *ThisTask*, mais attention, ce procédé ne peut fonctionner qu'avec la recherche de notre propre tâche car ce pointeur n'indique que la tâche actuellement traitée et ne peut donc indiquer les autres tâches. Ouf, Super OctetoPhage peut reprendre sa respiration après cette longue tirade. Nous modifions ensuite le pointeur mp_sigtask et nous pouvons alors nous ajouter au système.

ET QUE LE DEVICE SOIT

L'ouverture du dispositif se fait bien entendu avec la fonction *OpenDevice()*. Voici les erreurs possibles :

0 IOERR_NOERROR	
-1 IOERR_OPENFAIL	L' a pas pu
-2 IOERR_ABORTED	Fin provoquée par l'action de ABORTIO()
-3 IOERR_NOCMD	Commande impossible
-4 IOERR_BADLENGTH	Longueur impossible
20 TDERR_NOTSPECIFIED	Erreur du trackdisk non reconnue
21 TDERR_NOTSECHDR	Absence d' entête de secteur
22 TDERR_BADSECPREAMBLE	Mauvais préambule de secteur
23 TDERR_BASECID	Mauvais Disk Interrupt dans le secteur
24 TDERR_BADHDRSUM	Mauvaise somme de contrôle du handler
25 TDERR_BADSECSUM	Mauvaise somme de contrôle du secteur
26 TDERR_TOOFWSECS	Pas assez de secteurs sur la piste
27 TDERR_BASECHDR	Entête de secteur illisible
28 TDERR_WRITEPROT	Disquette protégée contre l' écriture
29 TDERR_DISKCHANGED	Lecteur de disquette vide
30 TDERR_SEEKERROR	Mauvaise position de la tête de lecture
31 TDERR_NOMEM	Pas assez de mémoire. Flûte alors
32 TDERR_BADUNITNUM	Lecteur de disquettes non connecté
33 TDERR_BADDRIVETYP	Mauvais lecteur de disquettes
34 TDERR_DRIVEINUSE	Lecteur déjà en cours d' utilisation
35 TDERR_POSTRESET	Pas d'accès autorisé après Reset

Voilà. Nous lisons les données grâce à *Io* et celui-ci modifiera la structure *diskio%* avant de se lancer avec *~DoIO()*. L'affichage du secteur tient compte de quelques offsets spécifiques, puis nous libérons le système.

Le mois prochain, un vrai plasma... entièrement en GFA !

```
' *****
' * CONCEPTION DU PROGRAMME      Pierre Philippe Launay *
' * EXTRAIT DE                  Artiodactyl Bloc Editor *
' * DATE DE CREATION             30 AVRIL 1991             *
' *                               *
' *****
programme                          ! Lecture des blocs
'
PROCEDURE programme                ! En C c' est l' instruction Main()
ouvre.Écran                        ! Oui, certe, que diantre
variable                           ! Les constantes du programme
```



```

mÉmoire%=&H10002      ! Chip
rÉservation           ! Espace pour les structures utilisées
tache%=FindTask(0)    ! Recherche de notre tâche
LONG{readreply%+&H10}=tache% ! SigTask sur notre tâche
~AddPort(readreply%)  ! Ajout de notre structure port messages
ouvre.device          ! Ouverture du trackdisk.device
io(readreply%,cmd_read%,longueur%,diskbuff%,saut%)
io(readreply%,etd_motor%,stop%,diskbuff%,saut%)
affichage             ! Présentation des données
final                 ! Salut
RETURN
,
PROCEDURE ouvre.Écran ! Met une fenêtre
OPENS 1,0,0,640,250,3,&H8000
OPENW #1,0,10,640,240,0,&H1000
TITLEW #1,SPACE$(62)+"Pierre Philippe, passez une agréable
journée"
RETURN
,
PROCEDURE variable    ! Les constantes du programme
cmd_read%=2
etd_motor%=9
espace%=SPACE$(1)
dispositif%="trackdisk.device"+CHR$(0)
dispositif%=V:dispositif$
bloc%=880
longueur%=512
saut%=bloc*512
secteur%=1
flag%=0
disque%=0
tÂche%=0
stop%=0
RETURN
,
PROCEDURE rÉservation ! Création de mémoire pour les structures
, -----
' Création des variables nécessaires aux structures GFA
' Réservation de 512 octets de mémoire pour les transmissions
, -----
tampon_diskio%=512
tampon_readreply%=512
tampon_diskbuff%=512
diskio%=MALLOC(tampon_diskio%,mÉmoire%)
readreply%=MALLOC(tampon_readreply%,mÉmoire%)
diskbuff%=MALLOC(tampon_diskbuff%,mÉmoire%)
, -----
port%=ADD(diskio%,80) ! Pointeur sur la structure message port
, -----
RETURN
,
PROCEDURE ouvre.device ! Mise en action d' un périphérique
erreur%=OpenDevice(dispositif%,disque%,diskio%,flag%)
IF erreur%
! Est-il là ?
~RemPort(readreply%) ! Suppression structure
~MFREE(diskio%,tampon_diskio%) ! Libération io
~MFREE(readreply%,tampon_readreply%) ! Libération message
~MFREE(diskbuff%,tampon_diskbuff%) ! Libération device
ALERT 0,"Trackdisk Device non ouvert|Erreur : "
+STR$(erreur%,3),0," Merci ",rÉponse|
SYSTEM ! C' est la plantade
ENDIF
RETURN
,
PROCEDURE
io(ln_replyport%,io_command%,io_length%,io_data%,io_offset%)
, -----
' Création des structures
' Action
, -----
LONG{ADD(diskio%,14)}=ln_replyport% ! Pointeur port réponse
INT{ADD(diskio%,28)}=io_command% ! Commande
LONG{ADD(diskio%,36)}=io_length% ! Octets à transférer
, -----
IF NOT (io_length%=0) ! Si pas à 0
LONG{ADD(diskio%,40)}=io_data% ! Pointeur tampon données
LONG{ADD(diskio%,44)}=io_offset% ! Saut relatif
ENDIF
~DoIO(diskio%) ! Lecture des secteurs
, -----
RETURN
,
PROCEDURE affichage ! Lecture de diskbuff
DISPLAY OFF
CLS

```

```

COLOR 6,5
TEXT 10,20," B L O C      880 "
offset
colonne
contenu
fermeture
DISPLAY ON ! L' écran redevient visible
RETURN
,
PROCEDURE offset ! Quelques valeurs
COLOR 5,3
TEXT 10,30,"Type      "
TEXT 10,40,"Somme de contrôle "
COLOR 3,0
TEXT 160,30,"Bloc racine de la disquette "
+HEX$(LONG{diskbuff%},8)
TEXT 160,40,HEX$(LONG{ADD(diskbuff%,20)},8)
RETURN
,
PROCEDURE colonne ! Les colonnes d' offset
FOR y|=1 TO 16
y1|=MUL(ADD(y|,5),10)
TEXT 10,y1|,STR$(MUL(PRED(y|),32),3)
TEXT 570,y1|,STR$(PRED(MUL(y|,32)),3)
NEXT y|
y|=50
RETURN
,
PROCEDURE contenu ! Le résultat
couleur1|=4
couleur2|=6
FOR x&=0 TO PRED(tampon_diskbuff%) STEP 4 ! Farfadet long%
ADD pasx&,66 ! 8 lettres en topaz 8
SWAP couleur1|,couleur2| ! Alternance de couleurs
IF x& MOD 32=0 ! Si on est en fin de ligne
ADD y|,10 ! Ligne suivante
pasx&=40 ! Début de la ligne sur l' écran
ENDIF ! Fin du test
lecture%=LONG{ADD(diskbuff%,x&)}
SELECT x&
CASE 432 TO 462
COLOR 4,7
TEXT pasx&,y|,MID$(MKL$(lecture%),1,1)+espace$
TEXT ADD(pasx&,16),y|,MID$(MKL$(lecture%),2,1)+espace$
TEXT ADD(pasx&,32),y|,MID$(MKL$(lecture%),3,1)+espace$
TEXT ADD(pasx&,48),y|,MID$(MKL$(lecture%),4,1)+espace$
DEFAULT ! Les autres cas
COLOR couleur1|,couleur2| ! Changement de couleurs
TEXT pasx&,y|,HEX$(lecture%,8)
ENDSELECT
NEXT x&
TEXT 150,220,"Fin de la démonstration en cliquant la souris"
RETURN
,
PROCEDURE clic ! Mickey et Maousse costaud
WHILE MOUSEK=0
WEND
REPEAT
UNTIL MOUSEK=0
RETURN
,
PROCEDURE fermeture ! A bientôt, my dear
~RemPort(readreply%) ! Suppression du port
~CloseDevice(diskio%) ! Fermeture du trackdisk
~MFREE(diskio%,tampon_diskio%) ! Libération io
~MFREE(readreply%,tampon_readreply%) ! Libération message
~MFREE(diskbuff%,tampon_diskbuff%) ! Libération device
RETURN
,
PROCEDURE final ! Le domaine gris
clic ! Clic clic
CLOSES 1 ! Fermeture de l' écran
SYSTEM ! La fin la plus sûre
RETURN
,
' C'EST FINI !!
,

```

Par Pierre Philippe Launay

Définition : une pile est une structure de données dans laquelle il n'est possible d'ajouter ou de retirer un élément que par son sommet (méthode dite LIFO : Last In First Out).

Si vous avez bien compris cette petite définition, alors les piles n'auront plus aucun secret pour vous... Elles sont le plus souvent représentées par un tableau, mais comme nous maîtrisons maintenant les listes chaînées (mais si, souvenez-vous, c'était dans l'ANT numéro 29, celui du mois dernier. C'est pourtant pas si vieux ! (NDLR : bof, tu sais, en informatique, tout va si vite...)), nous allons étudier les deux cas.

PRIMITIVES POUR LES TABLEAUX

Les deux premières primitives (en pseudo-code) que nous vous proposons, permettent de tester l'état de la pile, à savoir si elle est pleine ou au contraire vide.

```
Fonction Pile_Pleine (sommet : entier) : Booleen ;
Début
  Si sommet = Taille_Pile Alors
    Pile_Pleine <- Vrai ;
  Sinon
    Pile_Pleine <- Faux ;
  Fsi ;
Fin ;
```

```
Fonction Pile_Vide (sommet : entier) : Booleen ;
Début
  Si sommet = Limite_Inferieure_Pile Alors
    Pile_Vide <- Vrai ;
  Sinon
    Pile_Vide <- Faux ;
  Fsi ;
Fin ;
```

Ces deux fonctions permettent de savoir si nous dépassons les limites du tableau. Taille_Pile est le plus souvent une constante qui détermine le nombre de cases du tableau. Quant à Limite_Inferieure, il s'agit comme son nom l'indique de la borne inférieure du tableau. Ainsi, nous pouvons parfaitement n'utiliser qu'un seul grand tableau que nous partagerons en plusieurs piles.

```
Procédure Empile (val : Type_Choisi) ;
Début
  Si Non(Pile_Pleine(sommet_pile)) Alors
    sommet_pile <- sommet_pile + 1 ;
    pile[sommet_pile] <- val ;
  Sinon
    Afficher ("erreur : pile pleine") ;
  Fsi ;
Fin ;
```

```
Fonction Depile : Type_Choisi ;
Début
  Si Non(Pile_Vide(sommet_pile)) Alors
    Depile <- pile[sommet_pile] ;
    sommet_pile <- sommet_pile - 1 ;
  Sinon
    Afficher ("erreur : pile vide") ;
  Fsi ;
Fin ;
```

La procédure Empile ajoute l'élément qu'elle a reçu en paramètre après avoir testé l'état de la pile (pleine ?) et incrémenté le sommet. Le retrait d'un élément de la pile se fait après avoir testé l'état de la pile (vide ?) et avant de décrémenter le sommet.

Nous aurions pu choisir une autre méthode pour empiler, qui consiste à mettre la valeur dans le tableau avant d'augmenter le sommet. De ce fait, il aurait fallu être cohérent avec soi-même en modifiant en conséquence les autres fonctions, ainsi que les différentes bornes (essayez pour voir, c'est un bon exercice).

PRIMITIVES POUR LES LISTES CHAÎNÉES

Comme vous avez pu le constater, si l'on représente une pile sous forme de tableau, il est nécessaire d'avoir une variable (appelée sommet) indiquant l'indice du dernier élément de la pile. Par contre, si l'on utilise une liste chaînée, cette variable devient inutile. Pourquoi ?

Créons un noeud vide que nous appelons *DEBUT* et qui est le point de départ de notre liste. Puis, à chaque fois qu'un élément devra être empilé, il sera inséré entre *DEBUT* et d'éventuels autres éléments précédemment empilés. La variable indiquant le sommet de la pile devient inutile puisque nous savons toujours où se trouve le dernier élément empilé. Il en est de même pour dépiler : nous prenons le noeud qui suit *DEBUT*. Une liste simplement chaînée suffit amplement, car nous ne la parcourerons pas. Nous n'avons donc pas besoin de conserver la trace du noeud précédent (intérêt de la liste double).

```
Empiler :
  nouveau.suivant <- DEBUT.suivant ;
  DEBUT.suivant <- nouveau ;
  nouveau.info <- valeur ;
```

Pour empiler, il suffit d'insérer le nouveau noeud après *DEBUT*. Nous devons donc affecter à nouveau.suivant l'adresse du noeud qui suit *DEBUT* (cette adresse se trouve dans *DEBUT.suivant*). Ensuite, nous affectons à *DEBUT.suivant* l'adresse du nouveau noeud (ce dernier devient donc le successeur immédiat du noeud *DEBUT*). Ces deux étapes doivent se faire dans cet ordre sous peine de perdre le contenu de *DEBUT.suivant*.

Le test *Pile_Pleine* se fait au moment de l'allocation mémoire du nouveau noeud. Si celle-ci échoue, on considère la pile pleine.

```
Dépiler :
  Si Non(Pile_Vide) Alors
    noeud_temp <- DEBUT.suivant ;
    valeur <- noeud_temp.info ;
    DEBUT.suivant <- noeud_temp.suivant ;
  Sinon
    Afficher ("Pile Vide") ;
  Fsi ;
```

Si la pile n'est pas vide, nous repérons l'adresse de l'élément à dépiler (ceci nous est très facile puisqu'il s'agit toujours du successeur de *DEBUT*). Puis nous sauvegardons sa valeur dans une variable intermédiaire (variable valeur dans notre primitive). Enfin, pour le supprimer de la liste, nous le court-circuitons en faisant pointer *DEBUT.suivant* sur le successeur du noeud à dépiler (il s'agit de noeud_temp.suivant).

```
Pile_Vide :
  Si DEBUT.suivant = NIL Alors
    Pile_Vide <- Vrai ;
  Sinon
    Pile_Vide <- Faux ;
  Fsi ;
```

Si *DEBUT.suivant* est égal à *NIL*, la liste ne contient aucun élément hormis le premier noeud *DEBUT* qui ne contient pas d'information pertinente.

PRATIQUE

Pour se familiariser avec la pile, nous vous proposons ce mois-ci un petit programme permettant l'évaluation d'une expression parenthésée.

Cependant, quelques contraintes sont imposées :

- un membre se compose de deux opérandes séparés par un signe
- tout membre doit être mis entre parenthèses
- les opérandes peuvent être des membres

Ainsi, on entrera $((a+b)*c)$ et non $(a+b)*c$.

```
Algorithme Expression_Parenthesee ;
(* version pour les tableaux *)
Constante
  limite_inferieure_pile <- 0 ;
  max_operande <- 26 ;
  max_pile_operande <- 50 ;
```



```

max_pile_operateur      <- 50 ;

Declaration
  operande      : Tableau [1..max_operande] De Reel ;
  pile_operande : Tableau [1..max_pile_operande] De Reel ;
  pile_operateur : Tableau [1..max_pile_operateur] De
    Caractere ;
  expression    : Chaine [80] ;

  oper, operateur      : Caractere ;
  x, y, long, cpt      : Entier ;
  sommet_operateur, sommet_operande : Entier ;
  valeur, droite, gauche, result : Reel ;

Procédure Init_Tout ;
Debut
(* initialisation du tableau des operandes *)
Pour (cpt <- 1 Jusqu'à 26) Faire
  operande [cpt] <- 0 ;

(* Les tableaux commencent a 1 et nous avons choisi *)
(* d'incrémenter le sommet avant d'y ajouter un *)
(* element. Les sommets sont alors initialisés a 0 *)
(* pour qu'au premier ajout ils pointent sur la *)
(* premiere case *)
sommet_operande <- 0 ;
sommet_operateur <- 0 ;
Fin ;

(* Ici nous sommes contraints d'avoir une paire de fonctions *)
(* pour chaque pile car chaque sommet est géré de manière *)
(* indépendante. On évite ainsi de sortir de la zone mémoire *)
(* et donc de remplir des portions réservées. *)
Fonction Pile_Operateur_Vide (indice : entier) : Booleen ;
Début
  Si indice = limite_inferieure_pile Alors
    Pile_Operateur_Vide <- Vrai ;
  Sinon
    Pile_Operateur_Vide <- Faux ;
  Fsi ;
Fin ;

Fonction Pile_Operateur_Pleine (indice : entier) : Booleen ;
Début
  Si indice = max_pile_operateur Alors
    Pile_Operateur_Pleine <- Vrai ;
  Sinon
    Pile_Operateur_Pleine <- Faux ;
  Fsi ;
Fin ;

Fonction Pile_Operande_Vide (indice : entier) : Booleen ;
Début
  Si indice = limite_inferieure_pile Alors
    Pile_Operande_Vide <- Vrai ;
  Sinon
    Pile_Operande_Vide <- Faux ;
  Fsi ;
Fin ;

Fonction Pile_Operande_Pleine (indice : entier) : Booleen ;
Début
  Si indice = max_pile_operande Alors
    Pile_Operande_Pleine <- Vrai ;
  Sinon
    Pile_Operande_Pleine <- Faux ;
  Fsi ;
Fin ;

(* nous avons choisi d'incrémenter avant d'empiler l'element *)
Procédure empile_operande (val : Reel) ;
Debut
  Si Non(Pile_Operande_Pleine(sommet_operande)) Alors
    sommet_operande <- sommet_operande + 1 ;
    pile_operande [sommet_operande] <- val ;
  Sinon
    Afficher ("erreur : pile operande pleine") ;
  Fsi ;
Fin ;

(* Il faut sortir avant que l'infame GURU ne déboule. Une *)
(* bonne idée serait de sauvegarder les piles dans des *)
(* fichiers juste avant de sortir, et de les consulter par *)
(* soi-même. Car tous les compilateurs du DomPub ne possèdent *)
(* pas forcément de Debogueur *)
Quitter ;
Fsi ;
Fin ;

Procédure empile_operateur (val : Caractere) ;
Debut
  Si Non(Pile_Operateur_Pleine(sommet_operateur)) Alors
    sommet_operateur <- sommet_operateur + 1 ;
    pile_operateur [sommet_operateur] <- val ;
  Sinon
    Afficher ("erreur : pile operateur pleine") ;
    Quitter ;
  Fsi ;
Fin ;

```

```

Fonction depile_operande : Reel ;
Debut
  Si Non(Pile_Operande_Vide(sommet_operande)) Alors
    depile_operande <- pile_operande [sommet_operande] ;
    sommet_operande <- sommet_operande - 1 ;
  Sinon
    Afficher ("erreur : pile operande vide") ;
    Quitter ;
  Fsi ;
Fin ;

Fonction depile_operateur : Caractere ;
Debut
  Si Non(Pile_Operateur_Vide(sommet_operateur)) Alors
    depile_operateur <- pile_operateur [sommet_operateur] ;
    sommet_operateur <- sommet_operateur - 1 ;
  Sinon
    Afficher ("erreur : pile operateur vide") ;
    Quitter ;
  Fsi ;
Fin ;

Debut
(* initialisation generale *)
Init_Tout () ;

(* saisie des operandes et de leur valeur associee. Ex: A=3 *)
(* ou bien saisie d'un point pour sortir du mode de saisie *)
Faire
  Afficher ('entrez vos operandes : ') ;
  Lire (oper) ;
  Afficher(Majuscule(oper)) ;

  Si (oper <> '.') Alors
    Afficher(' = ') ;
    Lire (valeur) ;

    (* remplissage direct du tableau des operandes par conversion *)
    (* ex : a = 3 ; a = A grace a Majuscule ; puis Numero[A] = 65 *)
    (* enfin 65 - 64 = 1 et donc 3 est stocke dans operande [1] *)
    (* Ceci n'est possible que si les caracteres ont des valeurs *)
    (* qui se suivent dans la table du jeu de caracteres. Ce qui *)
    (* est le cas pour la table ASCII *)
    operande [Numero(Majuscule(oper)) - 64] <- valeur ;
  FinSi ;
  Jusqu'a (oper = '.') ;

  (* j'insiste sur le "correctement parenthesee" *)
  (* ex : ((A+B)-(C/D)) et non (A+B)-(C/D) *)
  Afficher('expression correctement parenthesee : ') ;
  Lire (expression) ;

  (* recupere la longueur de l'expression saisie afin de *)
  (* pouvoir l'analyser caractere par caractere *)
  long <- Longueur (expression) ;
  cpt <- 1 ;
  Tant Que (cpt <= long) Faire
    Debut
      Choix expression[cpt]

    (* si c'est une lettre, on empile sa valeur respective *)
    'A'..'Z','a'..'z' :
      empile_operande (operande[
        Numero(Majuscule(expression[cpt])) - 64]) ;

    (* empiler le signe *)
    '+','*','-','/' :
      empile_operateur(expression[cpt]) ;

    (* recuperer les deux valeurs qui entourent le dernier signe *)
    (* empile. Puis depiler ce signe et effectuer l'operation en *)
    (* consequence. Ex : (A+B) avec A=3 et B=2 ; droite = 2 ; *)
    (* gauche = 3 ; operateur = '+' ; result = 3 + 2 *)
    ')' :
      Debut
        droite <- depile_operande ;
        gauche <- depile_operande ;
        operateur <- depile_operateur ;

        Choix operateur
          '+' :
            result <- gauche + droite ;
          '-' :
            result <- gauche - droite ;
          '*' :
            result <- gauche * droite ;
          '/' :
            result <- gauche / droite ;
        Fin ; (* Choix operateur *)

    (* le resultat devient une operande, dans l'exple:empile(5) *)
    empile_operande (result) ;

  Fin ; (* pour la parenthese fermentee *)

```



```

    Fin ; (* Choix expression [cpt] *)

(* incrementer le compteur de progression de l'expression *)
    cpt <- cpt + 1 ;
    Fin ; (* Tant Que (cpt <= long) Faire *)

(* depiler la dernière operande qui est le resultat *)
    result <- depile_operande ;

(* sortie du resultat *)
    Afficher ('le resultat est : ', result) ;
Fin. (* programme principal *)

```

Listing 2 : Pile.c

```

/* Compile avec LC -lm -O */
/* Vous devez créer un fichier de données */
/* ayant la structure suivante */
/*
a 10.0
b 12.0
c 3.0
d 0.5
*/
/* Vous passerez ensuite ce fichier en paramètre */
/* à l'exécutable : <nom executable> <nom fichier de données> */

```

```

#include <exec/types.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

```

```

struct Pile_Operande {
    float operande;
    struct Pile_Operande *suivant;
};

```

```

struct Pile_Operateur {
    char operateur;
    struct Pile_Operateur *suivant;
};

```

```
float lettre[26];
```

```

struct Pile_Operande *debut_operande;
struct Pile_Operateur *debut_operateur;

```

```

VOID Empile_operande (float);
VOID Empile_operateur (char);
float Depile_operande (VOID);
char Depile_operateur (VOID);

```

```

main(int argc, char **argv)
{
    char expression[80], signe, c;
    float oper_gauche, oper_droite, result, val;
    int i, indice;
    FILE *fichier;

    debut_operande = (struct Pile_Operande *) malloc (sizeof
(struct Pile_Operande));
    debut_operande -> operande = -1000000.00;
    debut_operande -> suivant = NULL;

    debut_operateur = (struct Pile_Operateur *) malloc (sizeof
(struct Pile_Operateur));
    debut_operateur -> operateur = '\0';
    debut_operateur -> suivant = NULL;

    for (i = 0; i < 26; i++)
        lettre[i] = 0.0;

    printf ("Lecture du fichier de donnees...");
    fichier = fopen (argv[1], "r");
    if (! fichier)
    {
        puts ("Fichier introuvable !!!");
        exit (5);
    }
    while (feof (fichier) == 0)
    {
        fscanf (fichier, "%c %f\n", &c, &val);
        indice = c - 'a';
        lettre[indice] = val;
    }
    fclose (fichier);
    printf (" OK\n");

    printf ("Entrez l'expression correctement parenthesee\n>");
    fgets (expression, 80, stdin);
    for (i = 0; i <= strlen (expression); i++)
    {
        if ( isalpha (expression[i]))
        {
            indice = expression[i] - 'a';
            Empile_operande (lettre[indice]);
        }
        else
        {
            switch (expression[i])
            {
                case '(' :

```

```

oper_droite = Depile_operande();
signe = Depile_operateur();
oper_gauche = Depile_operande();
switch (signe)
{
    case '+' :
        result = oper_gauche + oper_droite;
        break;

    case '-' :
        result = oper_gauche - oper_droite;
        break;

    case '/' :
        result = oper_gauche / oper_droite;
        break;

    case '*' :
        result = oper_gauche * oper_droite;
        break;
}
Empile_operande (result);
break;

```

```

case '+' :
    Empile_operateur(expression[i]);
    break;

case '-' :
    Empile_operateur(expression[i]);
    break;

case '/' :
    Empile_operateur(expression[i]);
    break;

case '*' :
    Empile_operateur(expression[i]);
    break;

case '(' :
    break;
}
}

```

```

printf ("Le resultat est : %f\n", result);
free (debut_operateur);
free (debut_operande);
}

```

```

VOID Empile_operande (float nombre)
{
    struct Pile_Operande *nouveau;

    nouveau = (struct Pile_Operande *) malloc (sizeof (struct
Pile_Operande));

    nouveau -> operande = nombre;
    nouveau -> suivant = debut_operande -> suivant;
    debut_operande -> suivant = nouveau;
}

```

```

VOID Empile_operateur (char signe)
{
    struct Pile_Operateur *nouveau;

    nouveau = (struct Pile_Operateur *) malloc (sizeof (struct
Pile_Operateur));

    nouveau -> operateur = signe;
    nouveau -> suivant = debut_operateur -> suivant;
    debut_operateur -> suivant = nouveau;
}

```

```

float Depile_operande (VOID)
{
    struct Pile_Operande *element;
    float oper;

    element = debut_operande -> suivant;
    oper = element -> operande;
    debut_operande -> suivant = element -> suivant;
    free (element);
    return (oper);
}

```

```

char Depile_operateur (VOID)
{
    struct Pile_Operateur *element;
    char operat;

    element = debut_operateur -> suivant;
    operat = element -> operateur;
    debut_operateur -> suivant = element -> suivant;
    free (element);
    return (operat);
}

```

Par Bruno Bailluet et Lucien Turski-Marcon

Trop de mégademos et de jeux sur plusieurs disks assument encore que seul le drive DF0: est présent, alors que la plupart des possesseurs d'Amiga ont également au moins un second drive.

Pourtant, les Rom Kernel Manuals donnent le moyen de détecter directement, en interrogeant le Hardware, le nombre de drives présents et surtout leur numéro (rappel : sur un A2000, le premier drive externe est DF2: et non DF1:). Enfin, quand je dis "donnent le moyen", c'est un bien grand mot : les explications sont succinctes (NDLR : et buggées, c'est tout de même un comble !) et aucune routine n'est présentée. Le petit programme qui suit rattrape ce manque.

CE QUE DISENT LES RKM

D'après l'Amiga Hardware Reference Manual (Revised Edition), page 311, je cite : "un mode d'identification est prévu pour lire un flux d'identification sériel 32 bits depuis un périphérique externe. Pour initialiser ce mode, le moteur doit être mis en marche, puis arrêté (...). Le signal SELxB doit être inactif. Maintenant, entrez dans une boucle dans laquelle vous activez SELxB, lisez RDY (broche 1) et désactivez SELxB. Bouclez 32 fois pour lire 32 bits de données. Le bit le plus significatif est reçu en premier." Pas génial, comme explication...

Mais en cherchant un peu plus loin, page 357 exactement, on trouve une seconde description de ce mode d'identification, un peu plus étoffée. On y apprend que le code d'identification (ID) n'est plus sur 32 bits mais sur 16, et qu'il y a 9 étapes à suivre pour le trouver. Enfin, on y trouve une table des IDs possibles :

```
0000 0000 0000 0000 ($0000) Pas de drive
1111 1111 1111 1111 ($ffff) Drive Amiga 3" standard
1010 1010 1010 1010 ($aaaa) - réservé -
0101 0101 0101 0101 ($5555) Drive 5" DS DD
1000 0000 0000 0000 ($8000) - réservé -
0111 1111 1111 1111 ($7fff) - réservé -
0000 1111 xxxx xxxx ($0xxx) Librement utilisable
1111 0000 xxxx xxxx ($f0xx) - réservé -
xxxx 0000 0000 0000 ($x000) - réservé -
xxxx 1111 1111 1111 ($xfff) - réservé -
0011 0011 0011 0011 ($3333) - réservé -
1100 1100 1100 1100 ($cccc) - réservé -
Note : un 'x' représente un bit quelconque (0 ou 1)
```

Cette fois-ci, la description de la marche à suivre est correcte, mais les IDs sont faux (du moins les deux premiers, je n'ai pas le matériel nécessaire pour tester les autres) : on trouve \$ffff pour un drive Amiga 3" standard et \$0000 si aucun drive n'est connecté. Si quelqu'un parmi vous possède un drive 5" ou un lecteur haute-densité, je serais curieux de connaître leur ID (NDLR : si c'est le cas, écrivez-nous à la rédaction, on transmettra à François).

CE QU'IL FAUT FAIRE

La démarche pour le test d'identification est donc la suivante :

- 1 - mettre MTRXD à l'état bas
- 2 - mettre SELxB à l'état bas ('x' est le n° du drive à tester)
- 3 - mettre SELxB à l'état haut
- 4 - mettre MTRXD à l'état haut
- 5 - mettre SELxB à l'état bas
- 6 - mettre SELxB à l'état haut
- 7 - mettre SELxB à l'état bas
- 8 - lire et sauvegarder l'état de DSKRDY
- 9 - mettre SELxB à l'état haut
- 10- Répéter les pas 6 à 9 seize fois en tout.

Arrivé ici, un petit rappel des signaux n'est peut-être pas superflu :

```
MTRXD : bit 7 du port B du CIA-B ($bfd100)
SEL3B : bit 6 du port B du CIA-B ($bfd100)
SEL2B : bit 5 du port B du CIA-B ($bfd100)
SEL1B : bit 4 du port B du CIA-B ($bfd100)
SEL0B : bit 3 du port B du CIA-B ($bfd100)
DSKRDY : bit 5 du port A du CIA-A ($bfe001)
```

Le programme qui suit met en oeuvre le mode d'identification des drives. La routine TestDisks attend un seul paramètre : un buffer de 4

mots pointé par A0, dans lequel elle placera l'ID de chacun des quatre drives maximum qu'il est possible de connecter à l'Amiga. Elle est directement utilisable telle quelle et ne modifie aucun registre du 68000. Cette routine est compatible avec tous les modèles d'Amigas passés et présents (on ne parie jamais sur l'avenir !) et avec toutes les versions du Workbench (y compris le 2.0).

J'espère que cette routine aidera les programmeurs de jeux et de mégademos à prendre en compte tous les drives qui peuvent être connectés à l'Amiga (et pas seulement DF0:) et qu'à l'avenir, leurs oeuvres se débrouilleront toutes seules comme des grandes pour savoir d'où charger la suite du programme...

```
*****
* TestDisks.s
*****
* Interroge le hardware pour connaître le nombre de lecteurs de
* disquettes présents.
*****
* Par François Braün (Altaïr) pour Amiga NewsTech (ANT)
*****
* Assembler avec Devpac 2
*****
```

```
opt o+,ow-,c+
```

```
INCDIR include:
INCLUDE exec/exec_lib.i
INCLUDE libraries/dos_lib.i
```

```
*****
Demo lea DOSName(pc),a1 ; Ouvre la dos.library
moveq #0,d0 ; pour pouvoir écrire dans
move.l 4,a6 ; la fenêtre CLI courante
jsr _LVOpenLibrary(a6)
move.l d0,d7
beq Sortie
move.l d0,a6
jsr _LVOutput(a6)
move.l d0,d6
lea Buffer(pc),a0 ; Buffer de 4 mots
bsr.s TestDisks ; Teste les drives présents
lea Buffer(pc),a2 ; Affiche le buffer
moveq #3,d4 ; 4 lignes à afficher
Affiche lea OuiTxt(pc),a0
tst.w (a2)+
bne.s Oui
lea NonTxt(pc),a0
Oui moveq #3',d0 ; Place le n° du drive
sub.b d4,d0 ; dans la chaîne ASCII
move.b d0,2(a0)
move.l d6,d1 ; Affiche la phrase correspondante
move.l a0,d2 ; à l'état du drive
moveq #13,d3 ; (présent ou absent)
jsr _LVWrite(a6)
dbra d4,Affiche ; Boucle pour tous les drives
move.l d7,a1 ; Referme la dos.library
move.l 4,a6
jsr _LVCloseLibrary(a6)
Sortie moveq #0,d0 ; Retour au CLI
rts

Buffer dc.w 0,0,0,0 ; 1 mot par drive possible
DOSName dc.b "dos.library",0
OuiTxt dc.b "DFx: présent",10
NonTxt dc.b "DFx: absent ",10
even
```

```
*****
TestDisks:
move.l d0-d3/a1,-(sp) ; Sauve les registres utilisés
addq.l #8,a0 ; a0 = fin du buffer parce qu'on
; teste en sens inverse (3 à 0)
; a1 = CIAB-PRB
; 3 drives (DF3: à DF1:)

lea $bfd100,a1
moveq #2,d0
TstDsk move.b d0,d1
addq.b #4,d1 ; d1 = bit pour DSKSELx (4 à 6)
bclr #7,(a1) ; DSKMOTOR low
bclr d1,(a1) ; DSKSELx low
bset d1,(a1) ; DSKSELx high
bset #7,(a1) ; DSKMOTOR high
bclr d1,(a1) ; DSKSELx low
moveq #0,d2 ; d2 = ID
moveq #15,d3 ; 16 tests (1 mot)
TstLoop lsl.l #1,d2
bset d1,(a1) ; DSKSELx high
bclr d1,(a1) ; DSKSELx low
btst #5,$f01(a1) ; = $bfe001 = CIAA-PRA = DSKRDY
bne.s TstNext
bset #0,d2 ; Met le bit dans l'ID
TstNext bset d1,(a1) ; DSKSELx high
dbra d3,TstLoop ; Boucle pour les 16 bits
move.w d2,-(a0) ; Place l'ID dans le buffer
dbra d0,TstDsk ; Boucle pour les 3 drives
move.w d0,-(a0) ; DF0: est toujours présent !
movem.l (sp)+,d0-d3/a1 ; Récupère les registres
rts
```

Par François Braün

HARDWARE : LA VOIX DE PAULA (II)

Il faut distinguer deux modes d'utilisation du circuit audio : le mode automatique, utilisant les canaux DMA, et le mode manuel, utilisant le 68000.

Le premier devrait maintenant vous être familier, puisque c'est celui que nous avons développé jusqu'ici. Pourtant, vous ignorez encore certaines choses sur ce mode de fonctionnement, que nous allons maintenant disséquer. L'Amiga ne dispose pas d'un circuit audio, mais de quatre, c'est-à-dire un pour chaque voix. De même, Agnus utilise quatre canaux DMA pour transférer les données (rappelons qu'Agnus n'est qu'un larbin qui porte les colis ; c'est Paula qui orchestre tout).

Après avoir initialisé les registres correspondant à un canal, vous autorisez les transferts DMA à travers ce canal. A ce moment, que ce passe-t-il ? Paula transfère l'adresse et la longueur du son dans ses propres registres internes (spécifiques à chaque canal). Après, elle génère une interruption de niveau 4 pour avertir le 68000 qu'elle a fini l'opération (nous reviendrons sur cette interruption plus tard). Le processeur peut alors changer le contenu des registres concernant l'adresse et le volume du son, cela n'influera en rien la sortie audio en cours. Pendant ce temps, Agnus charge un mot de donnée toutes les deux périodes dans le registre AUD*DAT.

AUD*DAT \$DFF0+A Données du son (16 bits)

* = numéro du canal, + = \$A pour le canal 0, \$B pour le canal 1, \$C pour le canal 2 et \$D pour le canal 3

Vous vous rappelez sûrement que je vous avais dit qu'une coupure du canal DMA était sans effet si elle durait moins de temps que deux fois la période du son joué... Cela vient du fait que l'état du canal DMA n'est vérifié qu'avant chaque transfert de données. Donc, si vous le coupez et le ré-autorisez entre deux transferts, c'est comme s'il ne s'était rien passé. Et le temps entre deux transferts est le double de la période du son, puisque les données sont chargées mot par mot.

Les données situées dans AUD*DAT seront ensuite converties du format numérique au format analogique, pour être exportées vers les prises de sortie audio de l'Amiga. Une fois arrivée à la fin de l'échantillon sonore, Paula rechargera le contenu des registres sonores dans ses propres registres internes et générera à nouveau la même interruption, et ainsi de suite jusqu'à ce que les transferts DMA soient stoppés.

QUELQUES PROBLEMES... ET LEUR SOLUTION

Premier problème. Imaginons que nous voulions lancer en même temps le même son sur les quatre canaux. Si les sorties ne sont pas synchronisées entre elles, cela va produire une jolie cacophonie, voire cacophonie. Cela ne se produira toutefois que dans un cas précis : si l'ordinateur jouait déjà un son sur l'une ou plusieurs des voix. Dans ce cas, Paula ne tiendra compte du nouveau son qu'après avoir fini celui qu'elle était en train de jouer, d'où un décalage par rapport au son qui a débuté immédiatement.

J'en vois déjà qui pensent qu'il suffit d'interdire les transferts DMA de tous les canaux sonores, d'initialiser les registres qu'il faut puis d'autoriser ses propres canaux. Eh bien non, car qui vous dit que la coupure des canaux DMA aura duré moins de deux fois la période du son ? Pour en être sûr, il faut attendre le temps que dure deux fois la plus longue période possible, 65535 cycles, ou 20 millisecondes environ. Soit vous utilisez un timer des CIAs pour cela, soit vous utilisez une petite boucle faisant au moins 65535 cycles. Mais la première solution semble la plus propre et assurera un fonctionnement sur tout Amiga, même équipé d'une carte accélératrice. Pour résumer : avant tout travail avec Paula, s'assurer que les précédents transferts DMA sont bel et bien terminés.

Deuxième problème. Imaginons que vous vouliez raccorder deux sons (par exemple, deux morceaux d'une digit trop longue pour être jouée en une fois). Comment faire ? Voilà la recette : vous lancez le premier morceau normalement. Après avoir chargé le contenu des registres, Paula provoque une interruption de niveau 4. C'est là que vous intervenez : vous saisissez l'interruption au vol, ré-initialisez l'adresse et la longueur du son s'il le faut avec celles du deuxième morceau (sans toucher à DMAON) et le tour est joué. En effet, lors de la seconde lecture, ce sont les registres nouvellement initialisés que Paula chargera, et ce sera donc le nouveau morceau qui sera joué. Si vous avez plus de deux morceaux, le système est bien entendu exactement le même.

Mais revenons sur cette interruption qui n'est peut être pas connue de tous. Lorsqu'une interruption est générée, l'Amiga bloque le système et saute à l'adresse correspondante (pour les interruptions de niveaux 1 à 7, l'adresse de saut est indiquée en mots longs de l'adresse \$64 à \$80). L'interruption est alors traitée et la main est ensuite rendue au système. Une routine de traitement d'interruption doit toujours commencer par sauvegarder les registres du processeur qu'elle utilise, vérifier la provenance de l'interruption et voir si elle est autorisée, la traiter le cas échéant (puis indiquer qu'elle a été traitée, sinon l'ordinateur sera bloqué en traitant à l'infini la pauvre interruption qui n'en demandait pas tant), restaurer les registres et impérativement finir par un RTE.

INTENA \$DFF09A Interruptions autorisées (écriture)
INTENAR \$DFF01C Interruptions autorisées (lecture)
INTREQ \$DFF09C Interruptions demandées (écriture)
INTREQR \$DFF01E Interruptions demandées (lecture)

Bit	Nom	Fonction
15	SET/CLR	Même fonctionnement que pour DMAON
14	INTEN	Les interruptions ne peuvent avoir lieu que si ce bit est mis dans INTENA
10	AUD3	Interruption pour le canal 3
9	AUD2	Interruption pour le canal 2
8	AUD1	Interruption pour le canal 1
7	AUD0	Interruption pour le canal 0

Pour clarifier tout ça, voici un petit exemple de routine de gestion de l'interruption sonore.

```
START:
; D'abord, on sauvegarde quelques trucs
move.l $70,INT4 ; int. niveau 4
move.w $dff01C,INTA
or.w #$8000,INTA ; met bit 15
move.w $dff01E,INTD
or.w #$8000,INTD ; met bit 15
; On met les notes à la place
move.l #MON_INT,$70
move.w #$7fff,$dff09A
move.w #$7fff,$dff09C
move.w #$c080,$dff09A ; int. canal 0
; On lance le premier son
move.w #500,$DFF0A6 ; PERIODE
move.w #0050,$DFF0A8 ; VOLUME
move.l #DATA_SON,$DFF0A0 ; ADRESSE
move.w #50000,$DFF0A4 ; LONGUEUR en mots
move.w #$8201,$DFF096 ; Envoie canal 0
BOUCLE_WAIT:
btst #6,$BFE001 ; bouton gauche
bne.s BOUCLE_WAIT ; de la souris
; On remet tout comme il faut
move.w #0001,$dff096 ; SILENCE !
move.w #$7fff,$dff09A
move.w #$7fff,$dff09C
move.l INT4,$70
move.w INTA,$dff09A
move.w INTD,$dff09C
rts

MON_INT:
movem.l d0/a0,-(a7)
move.w $dff01c,d0
btst #$E,d0 ; autorisation ?
beq.s FIN_INT
and.w $dff01e,d0
btst #$7,d0 ; d'où viens tu ?
beq.s FIN_INT
```



```

; traitement interruption:
move.l POINTEUR,a0      ; pointeur sur table
move.l (a0)+,$DFF0A0    ; nouvelle adresse
move.w (a0)+,$DFF0A4    ; nouvelle longueur
cmp.l #FIN_TABLE,a0     ; détection fin table
bne.s SUITE_INT
lea TABLE,a0           ; Rebouclage de la digit
SUITE_INT:
move.l a0,POINTEUR      ; réactualise pointeur
move.w #$0080,$dff09C   ; Interruption traitée
FIN_INT:
movem.l (a7)+,d0/a0
rte

DATA_SON:
blk.b 384000,0          ; le son

TABLE:
dc.l DATA_SON
dc.w 50000
dc.l DATA_SON+100000
dc.w 50000
dc.l DATA_SON+200000
dc.w 50000
dc.l DATA_SON+300000
dc.w 42000
FIN_TABLE:

POINTEUR:
dc.l TABLE+6
INTA: dc.w 0
INTD: dc.w 0
INT4: dc.l 0

```

Attention ! Si le morceau suivant n'a pas la même fréquence, ou si vous voulez changer son volume, vous devrez attendre le commencement de ce morceau (chargement des registres) pour effectuer les changements. En effet, le volume et la fréquence ne sont pas sauvegardés par Paula : elle travaille directement dessus.

ON EST JAMAIS MIEUX SERVI QUE PAR SOI-MEME !

Bon, il est temps de passer au mode manuel. Ici, plus question de remplir AUD*LCH, AUD*LCL et AUD*LEN ni d'autoriser un quelconque DMA : vous devrez vous occuper vous-mêmes de ces choses. Par contre, il est toujours aussi nécessaire d'indiquer le volume (AUD*VOL) et la période (AUD*PER) de l'échantillon sonore. Après, vous transférez le premier mot de données sonores dans l'AUD*DAT correspondant au canal choisi (vous vous rappelez ? C'est l'adresse où Agnus chargeait les données sonores pour chaque canal). Cela aura pour effet d'amorcer le processus : le mot sera lu à la fréquence indiquée et une interruption de niveau 4 sera générée (la même que tout à l'heure) indiquant que vous pouvez envoyer le mot de données suivant. Si vous ne saisissez pas l'occasion, la transmission sonore prendra fin. Pour repartir, vous serez obligé de réamorcer le processus. Sinon, tout se poursuivra normalement.

Je ne vais pas vous présenter une routine d'application parce qu'elle serait la même que celle ci-dessus, à quelques détails près :

- plus de table définissant les fragments, mais seulement un pointeur et les données sonores (la détection de fin se fait ici sur la fin de l'échantillon).
- plus d'autorisation de DMA concernant les canaux utilisés. Il faut même les interdire, sinon les intervalles entre interruptions ne seront plus ceux attendus.
- plus de transfert d'adresse et de longueur durant l'interruption. Un seul mot de données sonores dans AUD*DAT (ne pas oublier le premier transfert permettant l'amorçage).

Ce mode de fonctionnement est tout de même assez contraignant, puisqu'il demande une attention constante de votre part. De plus, il coûte cher en cycles, puisque le 68000 est sans cesse requis (de ce fait, il n'est pas trop conseillé dans une démo ou tout autre programme demandant un travail conséquent en parallèle). Mais là s'arrêtent ses défauts... Vous me direz : c'est déjà pas mal... A mon avis, ce système est parfait pour un programme se bornant à faire de la musique. Certains semblent sceptiques... Je m'en vais les convaincre.

Le premier avantage de ce système est qu'il permet d'utiliser des échantillons dont la longueur ne dépend que de la capacité mémoire (c'est sûr, ce n'est pas très percutant comme argument).

Le second avantage est que vous pouvez faire de la modulation de fréquence et d'amplitude en utilisant uniquement la voix que vous voulez moduler. En effet, rien n'empêche de modifier le volume et la période du signal en même temps que vous chargez un mot de donnée dans AUD*DAT. Vous serez bien synchronisé cette fois-ci (là, ça commence à devenir intéressant...).

Le troisième avantage est que vous pouvez réaliser assez "facilement" de nombreux effets, tels que de l'inversion de son, du scraching, entre-coupage de samples, etc... (certes, ce serait possible en mode automatique, mais ce ne serait guère plus simple).

Le quatrième avantage n'est pas des moindres : vous pouvez mélanger en temps réel plusieurs signaux. En d'autres termes, vous pouvez faire croire que l'Amiga a plus de 4 voix (comment croyez-vous que font les logiciels de musique en 8 voix ?). Le subterfuge est simple : on sait que le son est défini par ses variations d'amplitude. Pour mixer deux sons, il suffit donc d'ajouter les amplitudes instantanées des deux signaux, en tenant compte des différences de volume et de fréquence (eh oui, sinon vous devrez toujours avoir des signaux dont la fréquence d'échantillonnage est la même et des notes de musique identiques sur un même canal). De plus, vous devez vous assurer que le résultat de l'opération a les dimensions requises (amplitude comprise entre +127 et -128). Je ne développerai pas ici les méthodes permettant de le faire, mais si cela vous intéresse, n'hésitez pas à écrire à l'ANT.

NDLR : Cinquième avantage du mode manuel, les données sonores n'ont plus besoin de se trouver en mémoire CHIP puisque le DMA n'est pas sollicité. Ca aussi, c'est intéressant !

On le voit, ce mode d'utilisation est tout de même assez riche. De plus, il existe des cas particuliers dans lesquels on peut difficilement s'en passer. Par exemple, pour messieurs les samplers fous, si vous voulez écouter ce que vous allez (où êtes en train de digitaliser) ce mode est obligatoire.

UN FILTRE D'AMMMMMMM...(IGA) ?

Mais à quoi sert-il donc, le filtre passe-bas ? Ceux qui pensent qu'il s'agit d'une sorte de loudness ont tout faux ! Ce filtre a pour effet de couper toutes les hautes fréquences (sons aigus). Il ne reste alors que des sons graves (basses fréquences). Plus de risques de distorsion (en anglais : aliasing) qui apparaît dans ces bandes de fréquences. C'est donc pour ravir vos oreilles que l'Amiga est équipé de ce filtre. Pour ma part, je préfère quand il n'est pas actif, car la bande de fréquence est alors beaucoup plus complète. Pour l'activer ou le désactiver, il suffit d'enlever ou de mettre le bit 1 du port A du CIA-A (adresse \$BFE001). Cela a aussi pour effet d'allumer et d'éteindre la power-led (pour les anciens Amigas uniquement).

Exemple :

```
eor.w #2,$BFE001 ; inverse l'état du filtre
```

DERNIERES RECOMMANDATIONS

Essayez l'effet d'écho obtenu en désynchronisant légèrement deux voix jouant le même son (l'effet est encore meilleur si elles sont sur le même canal stéréo).

Quelques conseils pour avoir des sons de bonne qualité :

- Utilisez toute la bande d'amplitude qui vous est offerte (+127 à -128). Sinon, il y aura plus de bruit qu'autre chose.
- N'échantillonnez pas trop bas en fréquence (7000 est vraiment un minimum !).
- Faites débiter et finir vos sons par la même valeur pour éviter les parasites lors des bouclages où des enchaînements.

Et voilà, vous savez tout... A vos claviers !

Par Philippe Rivaillon 

Comme promis, voici (enfin) l'étude de la version Amiga de Yacc, un utilitaire Unix aussi célèbre que Lex et que l'on utilise d'ailleurs avec ce dernier pour réaliser, entre autres choses, des compilateurs.

Le nom de Yacc est la contraction de "Yet Another Compiler Compiler", c'est-à-dire un compilateur de compilateurs. En effet, Yacc a été créé pour réaliser rapidement des compilateurs de langage : ces programmes lisent en entrée un source sous la forme d'un langage de programmation et génèrent un code de sortie dans un autre langage. Il s'agit d'un point de départ pour l'écriture de compilateurs Basic, Pascal, C ou de toute autre langage de votre choix. Il ne s'agit bien sûr que d'un outil d'aide à la réalisation de telles applications, il ne fait pas tout, tout seul non plus.

COMPARAISON ENTRE FLEX ET YACC

Sous bon nombre d'aspects, Yacc est similaire à Flex. Les deux programmes lisent un fichier d'entrée qui contient une liste de règles et la convertissent en une fonction C qui peut être compilée et exécutée (fig. 1). La différence est toutefois essentielle dans l'utilisation que l'on peut faire de ces deux logiciels : Flex est utilisé pour produire un analyseur lexical dont le but est de reconnaître dans un flot d'entrée, un certain nombre de symboles, alors que Yacc permet de réaliser un analyseur grammatical dont le but est de reconnaître des associations de symboles sous la forme de phrases. Ces deux programmes sont donc totalement complémentaires, car la base de tout programme de transcription est la combinaison d'un analyseur lexical et d'un analyseur syntaxique.

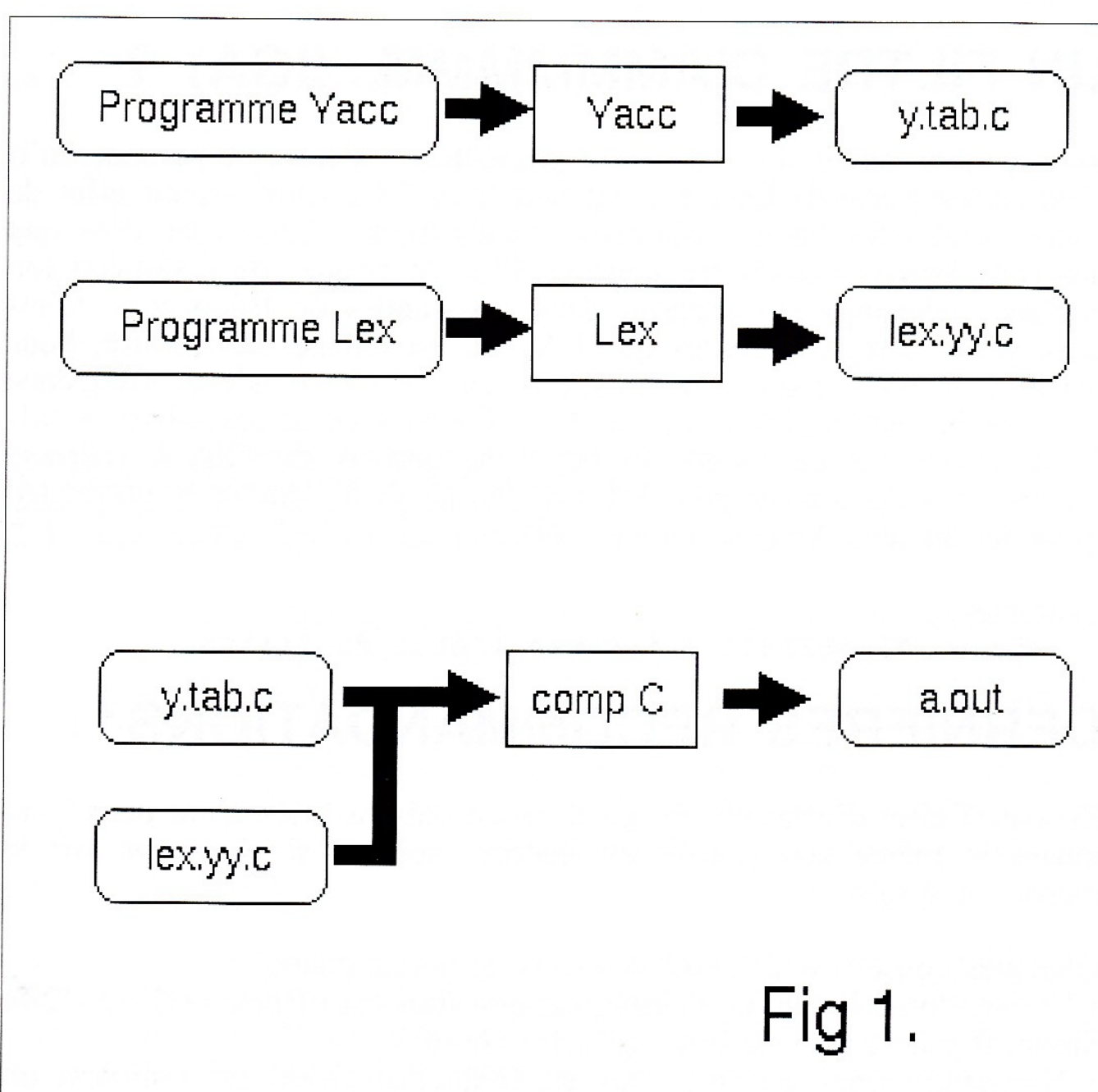


Fig 1.

UNE APPROCHE PAR L'EXEMPLE

Afin de mieux vous sensibiliser au fonctionnement de Yacc, nous allons réaliser un petit exemple d'analyseur syntaxique en prenant

comme base notre langue, le Français. Toute phrase en Français est constituée de mots agencés suivant des règles précises. Par exemple, une phrase va être constituée d'un nom, d'un verbe et d'un complément. La reconnaissance d'un mot appartenant à la langue française est du travail de Lex, alors que l'analyse de l'assemblage de ces mots pour déterminer si la phrase formée a un sens du point de vue grammatical est du ressort de Yacc.

En Français, la grammaire est basée sur une série de règles utilisant des notions basiques tels les noms, verbes, adjectifs et d'autres plus complexes, comme les phrases, attributs, sujets, objets.... Dans un programme Yacc, toutes ces notions sont représentées par des symboles grammaticaux.

Notre exemple d'analyseur va pouvoir traiter les notions suivantes :

- VERBE	pour un verbe
- NOM	pour un nom
- ADJECTIF	pour un adjectif
- NOMBRE	pour représenter un indication numérique
- Phrase	pour représenter une phrase
- Predicat	pour représenter un prédicat
- Complement	pour un complément
- Sujet	pour le sujet
- Objet	pour représenter un objet

Une différence a été faite intentionnellement entre majuscules et minuscules, pour marquer les différences de niveau grammatical. Un symbole comme "VERBE", qui représente un mot de la langue française, est une entité de bas niveau. Ce symbole est dit aussi "terminal", car il représente l'entité manipulable de plus bas niveau. Les symboles plus complexes comme "Phrase", qui sont réalisés à partir d'un assemblage de mots, sont dits de haut niveau ou "non-terminaux", car formés d'autres symboles, terminaux ou non.

Nous allons maintenant étudier comment transposer cette hiérarchie à Yacc, en prenant comme exemple la phrase suivante : "Stéphane range trois disquettes vertes".

En Français, l'ordre normal pour représenter une phrase est un sujet suivi d'un prédicat (c'est le cas de notre exemple). Toutefois, sous la forme impérative, seul le prédicat subsiste ("range trois disquettes vertes !"). Cette première règle peut être symbolisée en Yacc sous la forme :

```

Phrase : Sujet Predicat
       ;
  
```

"Phrase" représente ce qui va être défini. Il est suivi du caractère ":" et de sa définition, composée de "Sujet" et "Predicat". A la ligne suivante, la définition continue avec une barre verticale "|", symbole du "ou", lui-même suivi d'une deuxième définition. Ce genre de syntaxe est appelée une définition alternative ; la définition de Phrase s'arrête à la présence du point-virgule.

La définition de la phrase n'est pas encore complète, dans la mesure où elle est composée de symboles non-terminaux. En effet, Predicat et Sujet n'ont pas encore été définis.

Un prédicat est constitué, dans notre cas, d'un verbe et d'un complément, ce qui se représente fort simplement par :

```

Predicat : VERBE Complement
         ;
  
```

Comme on peut le remarquer, nous avons déjà une branche de notre arbre d'analyse qui se termine, puisque VERBE est terminal.

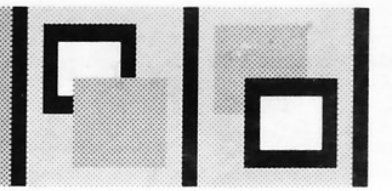
Etudions maintenant le sujet et le complément : tous deux peuvent être définis par un objet, ce qui se traduit par :

```

Sujet : Objet
      ;
Complement : Objet
           ;
  
```

Enfin, l'objet peut être :

- un nom
- un nombre + un nom
- un adjectif + un nom



- un nombre + un adjectif + un nom
Soit en Yacc :

```
Objet : NOM
      | NOMBRE NOM
      | ADJECTIF NOM
      | NOMBRE ADJECTIF NOM
      ;
```

Etant arrivés à décrire l'ensemble des entités complexes sous forme de symboles terminaux, le travail d'analyse est, par conséquent, terminé. Bien entendu, les règles de grammaire française sont beaucoup plus complexes que ce petit exemple, en particulier pour les exceptions qui font tout le charme de notre langue, et le malheur des programmeurs d'analyseurs syntaxiques.

PROGRAMMATION DE L'EXEMPLE

Maintenant que nous avons décrit les règles syntaxiques de notre exemple en langage Yacc, nous allons pouvoir bâtir un programme complet autour de ces règles.

Comme pour Flex, un programme Yacc est décomposé en trois sections :

- la section Déclarations
- la section Règles
- la section Fonctions Utilisateur

Chaque section étant séparée par "%%".

Nous commencerons notre programme par la section "Règles", qui représente le corps du programme.

La section REGLES

Nous allons donc transformer nos règles grammaticales en un programme qui reconnaît les phrases françaises. Nous devons également spécifier quelles sont les actions qui vont être effectuées lorsqu'une règle sera déclenchée : ces actions sont des fragments de code C qui sont positionnés à la droite de la déclaration de la règle, entourées par des accolades. Pour notre exemple précédent, la section règles devient :

```
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"); }
      ;

Sujet : Objet { printf(" Sujet \n"); }
      ;

Complement : Objet { printf(" Complément \n"); }
      ;

Objet : NOM
      | NOMBRE NOM
      | ADJECTIF NOM
      | NOMBRE ADJECTIF NOM
      ;
```

Ce morceau de programme représente le coeur de notre analyseur. Pour le faire fonctionner correctement, il nous faut désormais ajouter les autres sections.

Par rapport à ce que nous avons défini précédemment, nous avons ajouté la directive `error` pour le traitement des erreurs de syntaxe ainsi que des caractères "retour chariot", pour déterminer la fin d'une phrase.

La section DECLARATIONS

Nous allons maintenant étudier la première section du programme Yacc, à savoir la section Déclarations. Dans cette zone du programme, nous allons définir les symboles terminaux ainsi que toutes les variables globales dont nous aurons besoin dans le reste du programme.

Dans notre exemple, les symboles terminaux sont *VERBE*, *ADJECTIF*, *NOM*, *NOMBRE*. Ce sont des constantes entières, appelées "tokens", qui représentent des symboles grammaticaux en provenance de Flex et passés à Yacc.

La directive *token* fait en sorte que chaque symbole reçoive une valeur constante. Cette valeur est systématiquement supérieure à 256 pour qu'il n'y pas de conflit avec le passage de caractères, dont les codes ASCII sont compris entre 0 et 255.

Dans notre exemple, la directive *token* serait :

```
%token VERBE NOM ADJECTIF NOMBRE
```

Le caractère % se place immédiatement devant la directive, sans espace entre les deux. Lors de la phase de compilation, Yacc assignera une valeur à chaque symbole. Cette valeur est totalement transparente pour le programmeur qui n'a, du reste, pas à s'en soucier.

La section FONCTIONS UTILISATEUR

La dernière section du programme Yacc contient les fonctions de support, comme le programme principal, les routines de gestion des erreurs et le programme Lex. Dans un premier temps, nous allons créer les routines minimales pour chaque support.

Pour éviter de taper ces routines, nous pourrions nous en remettre directement à la bibliothèque de fonctions associée à Yacc. Mais pour une meilleure compréhension de l'ensemble et, compte tenu de la faible taille des fonctions, il est préférable de les écrire soi-même.

Programme principal : comme dans tout programme C, le programme principal s'appelle *main()*. Il est le point de départ du programme généré par Yacc. Dans la première version de notre exemple, le programme principal se contentera d'appeler la fonction *yyparse()*, qui est le nom de l'analyseur syntaxique qu'aura généré Yacc (de l'anglais *parse*, acte de séparer en plusieurs parties). Notre programme principal est donc :

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

On demande une phrase, on l'analyse et on rend la main à l'utilisateur. C'est tout.

Gestion des erreurs : deux fonctions de gestion des erreurs sont normalement nécessaires : *yyerror()* et *yywrap()*. La première est appelée lorsqu'en cours de fonctionnement, le programme Yacc découvre une erreur. La deuxième est invoquée à la fin, pour tout remettre en ordre avant de terminer l'exécution du programme. Pour une première version, ces deux routines seront vides.

```
yyerror()
{}

yywrap()
{}

```

La fonction Lex : elle peut être définie dans cette section. Elle doit s'appeler *yylex()*. Son rôle est de créer des tokens pour Yacc. Dans un premier temps, nous laisserons cette routine vide.

```
yylex()
{}

```

Voilà, nous avons maintenant tout ce qu'il nous faut pour réaliser un analyseur syntaxique. Le mois prochain, nous verrons comment compiler ce programme Yacc sur Amiga et nous continuerons notre apprentissage de ce langage en améliorant notre exemple.

En attendant, pour ceux qui connaîtraient déjà ce langage, et également pour vous permettre de vous le procurer pour le mois prochain, je vous signale que Yacc version Amiga est disponible dans la collection Fred Fish sur la disquette numéro 419.

Je me permets de vous souhaiter une bonne et heureuse année 1992, avec l'ANT, bien entendu. A bientôt.

Par Herr Doktor von GlutenStimmellImDorf



Votre pantagruélique serviteur revient faire des siennes dans les pages de votre cher ANT, suite à une absence prolongée pour cause de programmation intense d'utilitaires déments.

Et puisque l'on parle d'utilitaires, celui que je vous propose aujourd'hui, n'est pas dément, mais presque. Il s'agit d'un programme capable de greffer des programmes sur des programmes. Autrement dit, vous disposez d'un excellent programme, mais vous regrettez que lors de son lancement, telle ou telle fonction ne soit pas accomplie : il vous suffit de programmer vous-même ladite fonction et de la greffer sur le programme fautif grâce à l'utilitaire du mois.

RESTEZ SAGES !

Je vois venir certains petits malins, qui ont immédiatement compris qu'ils leur sera possible de se servir de cet utilitaire (au demeurant génial) afin de répandre d'odieux virus coquilles tout partout. Oui cela est possible, mais je maudis les coupables jusqu'à la 52ième incarnation. Voilà !

COMMENT CELA SE PEUSSE ?

Puisque l'Amiga est multitâche, n'en déplaise aux bidouilleurs de démos (que je maudis cette fois jusqu'à la 104ième incarnation), il n'est pas possible de savoir à l'avance à quelle adresse mémoire sera chargé tel ou tel programme.

Pour résoudre cette difficulté, les développeurs de l'Amiga ont employé la solution suivante : un programme exécutable n'est pas sauvé tel quel sur disquette. Ainsi, toutes les adresses de saut ne sont pas réelles mais comptées à partir de l'adresse 0. Lors du chargement du programme en mémoire, AmigaDOS se charge d'ajouter à chaque adresse répertoriée, l'adresse absolue de chargement et le programme se trouve relogé et donc en état de fonctionner. Bien entendu, pour que cela se fasse correctement, le fichier binaire contient, en plus du code lui-même, toute une pagaille de renseignements que nous verrons plus loin.

Ce n'est pas tout. En plus de ceci, et pour des raisons analogues (bonne gestion de la mémoire), un programme peut être constitué en seul bloc ou en plusieurs. Ce sont les segments ou *HUNKS*.

Pour y voir clair, entrez sous CLI la commande suivante :

```
type c:list opt h
```

Vous obtiendrez un dump hexa de la commande List, qui devrait ressembler à celui reproduit ci-dessous :

```
0000: 000003F3 00000000 00000002 00000000 .....
0010: 00000001 0000004F 00000961 000003E9 .....0...a....
0020: 0000004F 286A0164 700C4E95 2401223C ...O(j.dp.N.$."<
0030: 00000095 49FAFFEE ..... etc etc ...
```

000003F3 signifie que l'on a à faire à un programme exécutable
 00000000 ne signifie rien d'important
 00000002 signifie que le programme est en deux morceaux (HUNKS)
 00000000 est le numéro du premier hunk chargé
 00000001 est le numéro du second hunk chargé. Donc, il y a deux hunk à charger, puisque qu'en informatique 0 plus 1 égale 2, comme chacun sait
 0000004F est la longueur en mots longs du premier hunk
 00000961 est la longueur en mots longs du second hunk
 000003E9 est un identificateur particulier

Si le programme comportait un nombre plus élevé de segments, trois par exemple, on aurait 00000003 à la place de 00000002, suivi de

00000000, 00000001 et 00000002, puis trois mots longs donnant leur longueur respective, et ainsi de suite.

Vous avez compris ? Je continue.

Pour parvenir à nos fins, il suffit de connaître tous les identificateurs existants (ou plutôt ceux utilisés dans un fichier exécutable) et de savoir comment chaque hunk est structuré.

Nous allons donc passer tout ça en revue, sachant que les renseignements proviennent de la *Bible de l'Amiga*, donc qu'il vaut mieux de méfier. Mais c'est le seul livre à ma connaissance où il soit possible de trouver ces renseignements, depuis que Commodore a décrété que *l'AmigaDOS Technical Manual* était devenu obsolète. Il fallait oser...

REVUE DES TROUPES

\$3E9 hunk_code

Précède une partie de programme susceptible de fonctionner une fois relogé en mémoire. L'identificateur \$3E9 est suivi du nombre de mots longs de cette partie de programme, puis du code lui-même.

\$3EA hunk_data

Précède une partie de datas (données initialisées). Le reste est comme ci-dessus.

\$3EB: hunk_bss

Précède une zone BSS. Il s'agit d'une zone de mémoire sans valeur initiale particulière que le système alloue dynamiquement avant de la mettre à disposition du programme. L'identificateur \$3EA n'est donc suivi que de la taille de cette zone, exprimée en mots longs.

\$3EC hunk_reloc32

Précède la fameuse table de relogement. Ceci nous intéresse de près. Derrière \$3EC se trouvent le nombre d'offsets à corriger, le numéro de hunk concerné et enfin, les offsets eux-mêmes. Ces informations se répètent autant de fois que nécessaire (jamais plus souvent qu'il n'y a de hunks !) et la fin de la table est marquée par un nombre d'offsets égal à 0.

Pour greffer un programme sur un autre, il suffit donc de modifier l'en-tête afin de déclarer la présence du hunk supplémentaire, puis d'incrémenter de 1 chaque numéro de hunk concerné par un relogement. C'est aussi simple que cela.

\$3ED hunk_reloc16

\$3EE hunk_reloc8

Ces deux identificateurs sont tout-à-fait similaires au précédent, si ce n'est qu'ils concernent des relogements sur 16 ou 8 bits respectivement. Les tables elles-mêmes sont toujours constituées de mots longs.

\$3F0 hunk_symbol

Précède une table de symboles susceptibles d'intéresser un débogueur. D'après la Bible, cette table de mots longs est conclue par un mot long nul. Je suis désolé, mais j'ai vu de mes yeux des tables de symboles conclues par deux mots longs nuls. Mais la Bible sans erreurs ne serait pas la Bible...

\$3F1 hunk_debug

Dans cette table peut être mis à peu près n'importe quoi qui peut aussi intéresser un débogueur en tant qu'informations supplémentaires à celles contenues dans *hunk_symbol*. Voyez par exemple ce qu'en fait le Lattice (pardon le SAS) lorsque vous compilez et liez avec les options de débogage activées. \$3F1 est suivi du nombre de mots longs constituant la table, puis des longs mots eux-mêmes. Aucun format n'est donc imposé.

\$3F3 hunk_header

C'est le "mot de passe" qui, placé en début de fichier, assure le DOS qu'il a bien à faire à un programme exécutable.

Y'EN A D'AUTRES

Il existe encore d'autres identificateurs, qui concernent les fichiers objets et les overlays. Le programme que je vous propose devrait pouvoir greffer sans problème sur un programme comportant des overlays, mais je n'ai pas eu le temps d'essayer (bon d'accord, je n'ai pas eu le courage).

COMMENT ECRIRE LA GREFFE

C'est très simple. Il faut l'écrire en assembleur avec un macro-assembleur (SEKA ist streng verboten!), en deux segments dont un bidon (cf. exemples), sauver les registres au début, les récupérer puis sauter avec un JMP dans le segment bidon.

ET CA MARCHE

Parfaitement. J'ai même greffé un programme sur *Excellence!*, qui "pèse" tout de même la bagatelle de 221 segments. Lorsque vous avez greffé un programme, rien ne vous empêche, si ça vous amuse, de le greffer un fois de plus ou d'en greffer un autre, il n'y a pas de limite.

DEUX EXEMPLES

Le premier : Cls. Vous pouvez le greffer sur une commande CLI quelconque, list par exemple, en procédant comme suit :

```
Grefeur c:List QuelquePart:Cls Ram:SuperList
```

Vous obtenez en Ram: l'exécutable SuperList. Si vous lancez ce programme, vous avez un effacement du CLI courant avant le listage du répertoire.

Le second : CLI_Regs. Avez vous essayé de tracer sous Monam2 une commande CLI telle que List ? Oui ? Alors vous avez constaté que cela n'était pas possible, car les commandes CLI utilisent le contenu des registres transmis par le CLI. Greffez donc CLI_Regs sur vos commandes CLI et vous pourrez les tracer ! Il est vrai que cela plantera (beaucoup) plus tard au cours du traçage, mais pour une autre raison... que je vous laisse élucider.

CA MARCHE TOUJOURS ?

Non, certains programmes, comme par exemple *Deluxe Paint III*, ont des hunks trafiqués. En ce cas, le greffeur le signale et essaie quand même, mais vous aurez presque à coup sûr un gourou lorsque vous essaieriez le résultat (c'est pas ma faute !).

LE LISTING

Ecrit en C, il ne présente aucune difficulté de compréhension si vous maîtrisez convenablement les pointeurs, sinon bonsoir, je suis fatigué.

```
/*
/* *****
/* Programme: Greffeur.c
/*
/* Auteur: L'incroyable HUNK F Mazué pour ANT le 04/11/1991
/*
/* Fontion: Greffer un exécutable écrit par vous sur
/* n'importe quel exécutable de votre choix
/*
/* Remarque: Vous êtes vivement prié de ne pas vous servir de
/* cet utilitaire de grande classe afin de répandre
/* d'odieus virus coquilles tout partout....
/*
/* Utilisation: Greffeur <executable> <greffe> <résultat>
/*
/* Ecrit en SAS 5.10
/*
/* Compiler et linker d'un seul coup par:
/*
```

```
/* lc -d0 -m0t -v -y -Ln greffeur */
/*
/* VIVE LES POINTEURS !
/*
/* *****
#include <exec/types.h>
#include <exec/memory.h>
#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include <stdio.h>

struct FileLock *LockGrefe; /* because longword alignment */
struct FileLock *LockSource, *LockTarget;
struct FileInfoBlock *InfoGrefe, *InfoSource, *InfoTarget;
LONG *Source, *Grefe, *Target, *Fixe, *FixeSource, *FixeGrefe;
BPTR source, greffe, target;

VOID CleanUp();
LONG *Reloc(LONG *,int), *Code(LONG *), *Debug(LONG *);
LONG *Symbol(LONG *), *Data(LONG *), *Bss(LONG *);
LONG *Overlay(LONG *);

VOID main(int argc, char **argv)
{ int i, nbr_hunk = 0, compteur;
  LONG *pointeur, Test;

  printf("\nGreffeur de Programmes !!!\n");
  printf("Par F Mazué, L'incroyable HUNK !\n");

  if(argc != 4)
  { printf("La ligne doit avoir comme paramètres 3 noms de
    fichiers\n");
    printf("D'abord le fichier source qui doit être
    exécutable\n");
    printf("Ensuite le fichier à greffer sur le fichier
    source\n");
    printf("ce fichier doit aussi être
    exécutable\n");
    printf("Enfin le fichier cible\n");
    CleanUp();
  }

  InfoGrefe=(struct FileInfoBlock *)
  AllocMem(sizeof(struct FileInfoBlock),MEMF_PUBLIC|MEMF_CLEAR);
  if(!InfoGrefe) CleanUp();

  InfoSource=(struct FileInfoBlock *)
  AllocMem(sizeof(struct FileInfoBlock),MEMF_PUBLIC|MEMF_CLEAR);
  if(!InfoSource) CleanUp();

  InfoTarget=(struct FileInfoBlock *)
  AllocMem(sizeof(struct FileInfoBlock),MEMF_PUBLIC|MEMF_CLEAR);
  if(!InfoTarget) CleanUp();

  LockSource = (struct FileLock *)Lock(argv[1],SHARED_LOCK);
  Examine(LockSource,InfoSource);

  if(InfoSource->fib_DirEntryType >0)
  { printf("Nom de fichier ou de répertoire incorrect\n");
    CleanUp();
  }

  LockGrefe = (struct FileLock *)Lock(argv[2],SHARED_LOCK);
  Examine(LockGrefe,InfoGrefe);
  if(InfoGrefe->fib_DirEntryType >0)
  { printf("Nom de fichier ou de répertoire incorrect\n");
    CleanUp();
  }

  Source=(LONG *)AllocMem(InfoSource->fib_Size,
    MEMF_PUBLIC|MEMF_CLEAR);
  if(!Source) CleanUp();

  FixeSource=Source;
  source=Open(argv[1],MODE_OLDFILE);
  Read(source,Source,InfoSource->fib_Size);

  Greffe=(LONG *)AllocMem(InfoGrefe->fib_Size,
    MEMF_PUBLIC|MEMF_CLEAR);
  if(!Greffe) CleanUp();

  FixeGreffe=Greffe;
  greffe=Open(argv[2],MODE_OLDFILE);
  Read(greffe,Grefe,InfoGrefe->fib_Size);

  Target=(LONG *)AllocMem(InfoGrefe->fib_Size+
    InfoSource->fib_Size,
    MEMF_PUBLIC|MEMF_CLEAR);
  if(!Greffe) CleanUp();

  Fixe=Target;
  target=Open(argv[3],MODE_NEWFILE);
  if(!target) CleanUp();

  LockTarget = (struct FileLock *)Lock(argv[3],ACCESS_WRITE);
  if(*Source != 0x3F3 || *Grefe != 0x3F3)
```



```

{ printf("\nIl faut des fichiers exécutables !!\n");
  CleanUp();
}

pointeur = Greffe+2;
if(*pointeur !=2)
{ printf("\nLa Greffe ne remplit pas les conditions
nécessaires\n");
  printf("Opération terminée... désolé\n");
}

*Target++=*Source++; /* 3F3 */
*Target++=*Source++; /* Nom du HUNK (à priori NULL) */
nbr_hunk=*Source;
printf("\nLe programme Source contient %d HUNKS\n",nbr_hunk);

*Source +=1; /* un HUNK de plus */
*Target++=*Source++; /* Nbre de HUNK placé */
*Target++=*Source++; /* N° du premier HUNK chargé */

compteur = *Source;
compteur +=1;

if(compteur != nbr_hunk)
{ printf("\n le nombre de HUNKS de la source et le nombre de
HUNKS\n");
  printf("effectivement chargés ne se correspondent pas
!\n");
  printf("\nOn essaie quand même ! [RETURN]\n");
  getchar();
}

*Source +=1; /* Dernier HUNK à charger plus 1 */
*Target++=*Source++;

/* Source pointe sur le premier identificateur de la source */
pointeur=Greffe+5; /* Taille du HUNK à ajouter */
printf("\nLa Taille du Hunk greffé est %d octets\n",
*pointeur*4);

*Target++=*pointeur++;
for(i=0;i<compteur;i++)
{ *Target++=*Source++; /* Transfert des longueurs de hunk de
la source */
}

/* Source pointe sur le premier identificateur de la source */
pointeur=Greffe+7;

/***** TRAITEMENT DE LA GREFFE *****/
while(*pointeur != 0x3F2) /* Fin d'un HUNK ? */
{ switch(*pointeur) /* Quel HUNK ? */
{ case 0x3EC:
  pointeur=Reloc(pointeur,0);
  printf("Relogement 32 bits de la greffe effectué\n");
  break;

  case 0x3ED:
  pointeur=Reloc(pointeur,0);
  printf("Relogement 16 bits de la greffe effectué\n");
  break;

  case 0x3EE:
  pointeur=Reloc(pointeur,0);
  printf("Relogement 8 bits de la greffe effectué\n");
  break;

  case 0x3E9:
  pointeur=Code(pointeur);
  printf("Transfert du code de la greffe effectué\n");
  break;

  case 0x3F0:
  pointeur=Symbol(pointeur);
  printf("Table de symboles de la greffe traitée\n");
  break;

  case 0x3F1:
  pointeur=Debug(pointeur);
  printf("Table de débogage de la greffe traitée\n");
  break;

  default:
  *Target=*pointeur++;
  break;
}
}

*Target++=*pointeur++;

/***** TRAITEMENT DE LA SOURCE *****/
printf("\n\n*** Traitement de la source ***\n\n");

/* *Source 1 0x0FFFFFFF pour filtrer les HUNK chip-ram etc */
while((Test=(*Source & 0x0FFFFFFF)) != 0x3F2 || compteur !=0)
{ switch(Test)
{ case 0x3EA:
  Source=Data(Source);
  printf("\nHunk Data traité\n");
  compteur -=1;
  break;

  case 0x3EB:
  Source=Bss(Source);
  printf("\nHunk Bss traité\n");
  compteur -=1;
  break;

  case 0x3EC:
  Source=Reloc(Source,1);
  printf("Relogement 32 bits effectué\n");
  break;

  case 0x3ED:
  Source=Reloc(Source,1);
  printf("Relogement 16 bits effectué\n");
  break;

  case 0x3EE:
  Source=Reloc(Source,1);
  printf("Relogement 8 bits effectué\n");
  break;

  case 0x3E9:
  Source=Code(Source);
  printf("\nHunk Code traité\n");
  compteur -=1;
  break;

  case 0x3F0:
  Source=Symbol(Source);
  printf("Hunk Symbol traité\n");
  break;

  case 0x3F1:
  Source=Debug(Source);
  printf("Hunk debug traité\n");
  break;

  case 0x3F5:
  printf("\nAttention, la source contient des Overlays\n");
  printf("Le résultat n'est pas garanti...\n");
  Source=Overlay(Source);
  printf("Overlay traité\n");
  break;

  default:
  *Target++=*Source++;
  break;
}
}

*Target++=*Source;

Write(target,Fixe,(int)Target-(int)Fixe); /* Et voilà ! */
printf("\n\nOPERATION TERMINEE F Mazué vous salue\n\n");
CleanUp();
}

VOID CleanUp()
{
  if(FixeSource) FreeMem(FixeSource,InfoSource->fib_Size);
  if(FixeGreffe) FreeMem(FixeGreffe,InfoGreffe->fib_Size);

  if(LockSource) UnLock(LockSource);
  if(LockGreffe) UnLock(LockGreffe);
  if(LockTarget) UnLock(LockTarget);

  if(source) Close(source);
  if(greffe) Close(greffe);
  if(target) Close(target);

  if(InfoSource) FreeMem(InfoSource,
    sizeof(struct FileInfoBlock));
  if(InfoGreffe) FreeMem(InfoGreffe,
    sizeof(struct FileInfoBlock));
  if(InfoTarget) FreeMem(InfoTarget,
    sizeof(struct FileInfoBlock));
  exit(0); /* Fin du match ! */
}

LONG *Reloc(LONG *pointeur,int déplacement)
{ int i, encore=1, limite;

  *Target++=*pointeur++; /* 0x3EC est mis*/
  while(encore)
  { limite=*pointeur;
    *Target++=*pointeur++; /* nbr de correction placé */
    *pointeur +=déplacement; /* HUNK corrigé */
    *Target++=*pointeur++; /* et placé */

    for(i=0;i<limite;i++)
      *Target++=*pointeur++;

    if(*pointeur==0)

```



```

    { *Target++=*pointeur++; /* pour le 0 final */
      break;
    }
  }
  return(pointeur);
}

LONG *Code(LONG *pointeur)
{ int limite, i;

  *Target++=*pointeur++; /* 0x3E9 est mis*/
  limite = *pointeur; /* nombre de longs mots à copier */
  *Target++=*pointeur++; /* Ce nombre lui-même copié */

  for(i=0;i<limite;i++)
    *Target++=*pointeur++;

  return(pointeur);
}

LONG *Debug(LONG *pointeur)
{ int limite, i;

  *Target++=*pointeur++; /* 0x3F1 est mis */
  limite = *pointeur; /* nombre de longs mots à copier */
  *Target++=*pointeur++; /* Ce nombre lui-même copié */

  for(i=0;i<limite;i++)
    *Target++=*pointeur++;

  return(pointeur);
}

LONG *Symbol(LONG *pointeur)
{ int perpet=1;
  LONG Test;

  *Target++=*pointeur++; /* 0x3F0 est mis */
  while(perpet)
  { switch(Test=(*pointeur & 0xFFFFFFFF))
    { case 0x3E9: /* cf article */
      case 0x3EA:
      case 0x3EB:
      case 0x3EC:
      case 0x3ED:
      case 0x3EE:
      case 0x3F1:
      case 0x3F2:
      case 0x3F5:
        return(pointeur);

      default:
        *Target++=*pointeur++;
        break;
    }
  }

  *Target++=*pointeur++; /* 0x3EA est mis*/
  limite = *pointeur; /* nombre de longs mots à copier */
  *Target++=*pointeur++; /* Ce nombre lui-même copié */

  for(i=0;i<limite;i++)
    *Target++=*pointeur++;

  return(pointeur);
}

LONG *Bss(LONG *pointeur)
{
  *Target++=*pointeur++; /*0x3EB mis */
  *Target++=*pointeur++; /* Longueur mise */
  return(pointeur);
}

LONG *Overlay(LONG *pointeur)
{ int limite, i;

  *Target++=*pointeur++; /* 0x3F5 est mis*/
  limite = *pointeur; /* nombre de longs mots à copier */
  *Target++=*pointeur++; /* Ce nombre lui-même copié */
  *Target++=*pointeur++; /* Niveau d'Overlay */

  for(i=0;i<limite;i++)
    *Target++=*pointeur++;

  *Target++=*pointeur++; /* fin d'overlay */
  return(pointeur);
}

```

Listing 2 : Cls.s

```

;
;      *** CLS ***
;
; Programme d'effacement de la fenetre

```

```

; CLI ou SHELL courante par F MAZUE

; offsets d'EXEC.LIBRARY
OldOpenLibrary = -408 ;une fois n'est pas coutume !
CloseLibrary   = -414

; offsets de DOS.LIBRARY
Output          = -60
Write           = -48

section 1,code

movem.l d0-d7/a0-a6,-(sp)
tst.l d0
beq erreur

move.l $04,a6
lea dosname(pc),a1
jsr OldOpenLibrary(a6)
beq erreur
move.l d0,a6
jsr Output(a6)
lea debut(pc),a0
move.l a0,d2
moveq #(fin-debut),d3
jsr Write(a6)
move.l a6,a1
move.l $04,a6
jsr CloseLibrary(a6)
erreur
movem.l (sp)+,d0-d7/a0-a6
jmp suite

dosname
dc.b 'dos.library',0
even

debut
dc.b $9b,";H" ; ATTENTION, respecter les majuscules !
dc.b $9b,"J"
fin

section 2,code
suite
nop

```

Listing 3 : CLI_Regs.s

```

*****
* prog: reg_cli.s
* Auteur: F Mazué pour ANT le 04/11/1991
*
* Programme qui initialise les registres A2,A5,A6 afin qu'il
* soit possible de tracer sous le debogueur de votre choix la
* commande CLI sur laquelle il sera greffé.
* Necessite le programme 'Greffeur' du même auteur
*****

incdir "include/" ;mettez ici votre path personnel
include "exec/exec.i"
include "exec/exec_lib.i"
include "exec/execbase.i"
include "libraries/dos.i"
include "libraries/dosextens.i"

section 1,code

movem.l d0/a0,-(sp) ; Obligatoirement INDISPENSABLE !
move.l 4,a6 ; execbase
move.l LibList(a6),a0
lea dosname,a1
CALLEXEC FindName ; chercher et trouver la dos.library et
                  ; obtenir la valeur de DOSBASE
                  ; c'est ici une solution préférable
                  ; à ouvrir puis fermer la librairie

move.l d0,a0 ; cf RMK 'libraries/dosextens.h (ou .i)
move.l d1_A2(a0),a2 ;
move.l d1_A5(a0),a5 ;
move.l d1_A6(a0),a6 ;

movem.l (sp)+,d0/a0
jmp dummy

dosname DOSNAME
even

section 2,code
dummy
rts ; n'importe quoi ou autre chose ....

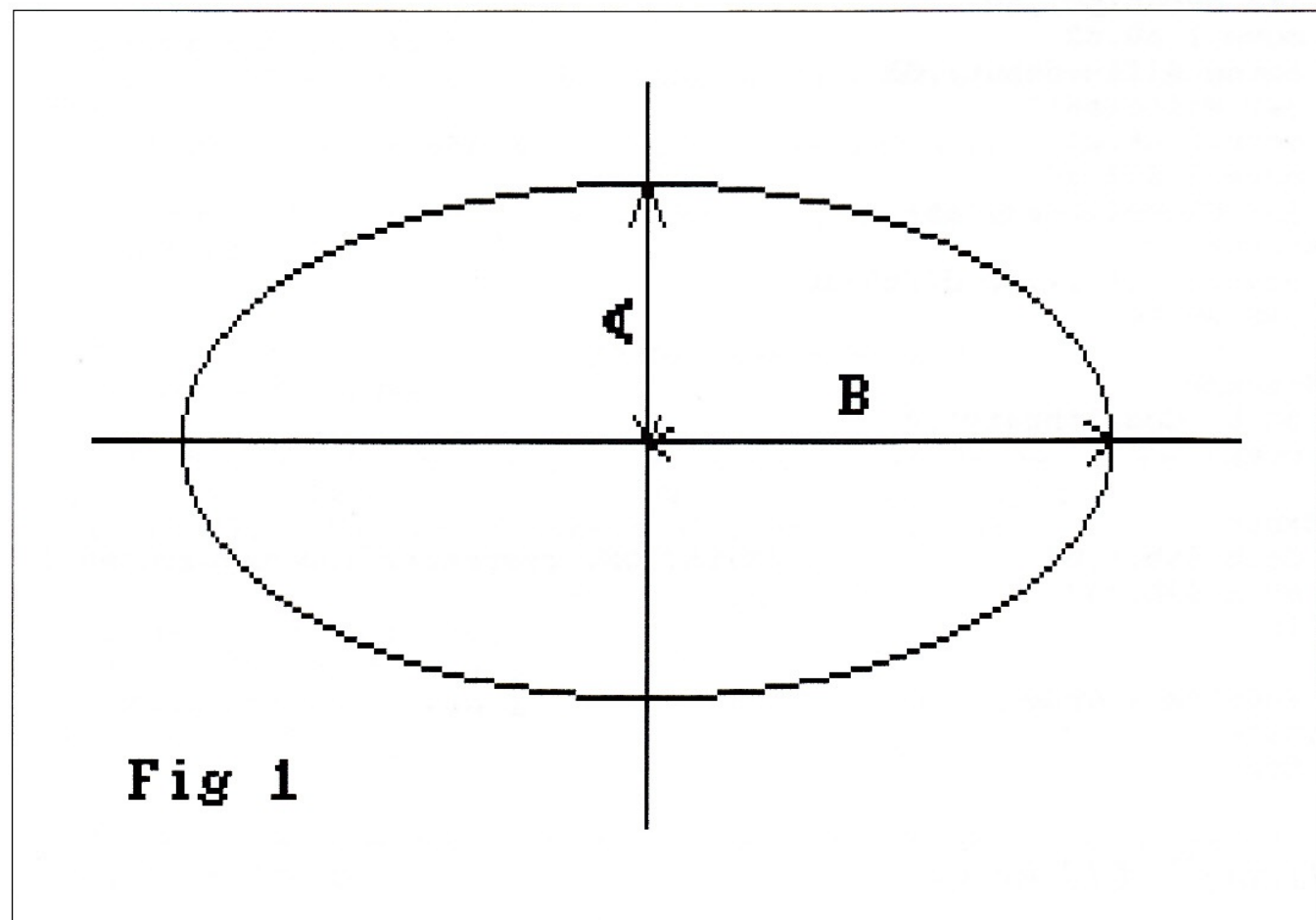
```

Par Frédéric Mazué

DEMOS : LES ELLIPSES

Aujourd'hui, je vais vous parler d'un sujet qui me tient à coeur, les ellipses et les cercles, ces derniers n'étant rien d'autre qu'un cas particulier des premières.

Hé oui, je vais vous expliquer comment afficher de magnifiques ellipses à l'écran, et ce de manière extrêmement rapide. Hélas, petite contrainte : nos ellipses seront parallèles aux axes ! Pour mieux comprendre, regardez le schéma 1 ci-dessous, vous y trouverez toutes les informations nécessaires.



Une ellipse est donc définie par son centre (xc,yc) et ses rayons a et b. Si a=b on obtient un cercle. Cette ellipse est définie mathématiquement par l'équation

$$x^2/a^2 + y^2/b^2 = 1.$$

Une fois obtenus x et y vérifiant cette équation, il ne reste plus qu'à additionner les coordonnées du centre xc et yc à l'ensemble des coordonnées des points.

Pour tracer notre ellipse, nous allons dessiner un point par ligne d'écran et ses symétriques, puis remplir le tout à l'aide du Blitter en mode remplissage. Il y a en effet deux symétries dans une ellipse de ce type : une horizontale et une verticale. Il suffit donc de calculer un seul quart de l'ellipse puis d'afficher les autres points par symétrie.

Nous partons du centre de l'ellipse, du point de coordonnées (x=a,y=0), qui vérifie évidemment l'équation de l'ellipse. Puis nous cherchons le point suivant d'abscisse $x_l = x + \text{delta}$ et d'ordonnée $y_l = y + 1$. Tout le problème est de calculer delta. La démonstration est un peu longue pour tenir dans la place qui m'est impartie, mais on arrive au résultat suivant.

Pour calculer delta, on effectue la somme $(2*y-1)*a*a/b*b$ puis on ajoute $-2*x+1$, en décrémentant x à chaque itération jusqu'à ce que cette somme devienne négative. Je m'explique un peu : cette valeur delta est représentative de la différence entre l'ellipse affichée et l'ellipse théorique. Si delta est positif, cela veut dire que le point (x,y) est à l'extérieur de la surface de l'ellipse, et si delta est négatif ce point est à l'intérieur de l'ellipse. Si enfin, delta est nul, alors (x,y) est sur l'ellipse. Plus delta est petit, plus le point de coordonnées (x,y) est proche de l'ellipse. C'est vrai, ce n'est pas évident à comprendre, mais une ellipse est un peu plus compliquée qu'une droite !

Nous avons donc le point (x,y) que nous affichons, ainsi que ses symétriques. Une fois que cela est fait pour tous les points de l'ellipse,

il ne reste plus qu'à donner un petit coup de Blitter pour remplir le tout.

Quelques remarques : il n'y a pas de clipping, c'est-à-dire que si l'ellipse vient à sortir de l'écran, elle risque fort d'aller embêter la mémoire des petits copains ! De plus, ce listing trace une ellipse pleine. Pour ne dessiner que le contour, il suffit d'enlever le remplissage au Blitter et d'afficher un point à chaque itération du calcul de delta.

Le listing ci-dessous est l'application de cet algorithme. Il est bien entendu sous Devpac, car je pense que toute personne sensée à définitivement cessé d'utiliser Seka. Vous pouvez changer les paramètres A et B (rayons de l'ellipse) en manipulant la souris et déplacer le centre de l'ellipse à l'aide du joystick. Cette dernière opération est à utiliser avec modération pour ne pas faire sortir l'ellipse de l'écran.

Voilà, j'espère que tout le monde n'a pas abandonné la lecture ! La prochaine fois, j'essaierai de trouver quelque chose de plus simple.

```
; *****
; * Ellipse.s
; * Auteur: Thomas LANDSPURG
; *
; * A assembler avec Devpac 2
; *****

OPT      O1+,O2+,W-,A-,C-

NB_BYTE_LIGNE=40 ; Taille sur x de l'écran en octets
NB_LIGNE=200      ; Hauteur de l'écran en lignes
prof=1           ; profondeur de l'écran

CUSTOM=$dff000

incdir "include:"
include exec/exec_lib.i
include graphics/graphics_lib.i
include hardware/custom.i

; DEBUT DU PROGRAMME
start    move.l #graphname,a1 ; ouverture de graphics.library
        CALLEXEC OpenLibrary
        move.l d0,_GfxBase
        move.l d0,a0
        move.l $32(a0),oldcop ; Ancienne copperlist

        bsr allocscr
        tst.l abtplane0 ; en cas d'erreur memoire,
        beq.s error_mem ; on sort
        CALLGRAF OwnBlitter
        CALLEXEC Forbid
        move.b #10000111,$bfd100 ; Arrete le drive (BEURK!)
        lea CUSTOM,a5
        move.w #$20,dmacon(a5) ; Desactive DMA sprites

out:     bsr main_pg ; pg principal
        lea CUSTOM,a5
        move.w #$8020,dmacon(a5); reactive DMA sprites
        CALLEXEC Permit ; reautorise le multitache
        CALLGRAF DisownBlitter ; et l'utilisation du blitter
        bsr freescr ; liberation de l'écran

        move.l _GfxBase(pc),a0 ; restore la copperlist
        move.l oldcop,$32(a0)
        move.l $26(a0),CUSTOM+cop1lc

error_mem:
        move.l _GfxBase(pc),a1
        CALLEXEC CloseLibrary ; referme les librairies
        moveq.l #0,d0
        rts

; *****
; * PG PRINCIPAL *
; *****

main_pg  bsr sync ; attente de la synchro
        bsr switch ; switching de page
        bsr bltclear ; effacement de l'écran
        bsr waitblit
        lea CUSTOM,a5
        move.w joy0dat(a5),d4 ; recupere coord de la souris
        move.w d4,d5
        lsr.w #8,d5
        and.w #$7f,d4
        move.w d4,rayona ; A de l'ellipse
        lsr.w #2,d5
        move.w d5,rayonb ; B de l'ellipse
        bsr joyst
        bsr draw_ellipse ; affichage de l'ellipse

        btst #6,$bfe001 ; test bouton droit de la souris
        bne.s main_pg
        rts

; switching de page
switch  move.l abtplane0(pc),a5
        move.l bbtplane0(pc),a4
        tst.b flagcop ; gestion switching de page
        bne copb
        exg a5,a4 ; on inverse les deux buffers
```



```

copb    not.b    flagcop
        move.l   a4,bitplane0    ; buffer de travail
        bsr      initbrush       ; on change la copperlist
        move.l   _GfxBase(pc),a0 ; on reaffiche la copperlist
        move.l   #mycop,$32(a0)
        lea      CUSTOM,a5
        move.l   #mycop,copllc(a5)
        clr.w    copjmp1(a5)
        rts

; ATTENTE DE LA SYNCHRO
sync     btst     #0,vposr+1(a5)
        beq      sync
sync2    cmp.b    #20,vhposr(a5)
        bne      sync2
        rts

; JOYSTICK
joyst    move.w   joyldat(a5),d0
        btst     #1,d0
        beq      pasdroite
        add.w    #1,circlex
pasdroite:
        btst     #9,d0
        beq      pasgauche
        sub.w    #1,circlex
pasgauche:
        move.w   d0,d1
        asr.w    #1,d1
        eor.w    d1,d0
        btst     #8,d0
        beq      pashaut
        add.w    #1,circley
pashaut  btst     #0,d0
        beq      pasbas
        sub.w    #1,circley
pasbas:
out_joyrts

; Allocation memoire pour les buffers ecran
allocscr:
        move.l   #taillescr*2,d0 ; 2 ecrans car double buffering
        move.l   #10002,d1
        CALLEXEC AllocMem
        move.l   d0,abitmap0     ; premier buffer
        move.l   d0,bitplane0
        add.l    #taillescr,d0
        move.l   d0,bbitmap0     ; deuxieme buffer
        rts
; desallocation memoire des buffers
freescr  move.l   abitmap0(pc),a1
        move.l   #taillescr*2,d0
        CALLEXEC FreeMem
        rts

; MET DANS LA COPPER LIST LES ADRESSES DES BITPLANES
initbrush:
        lea      p_bitpl,a0
        move.w   #00e0,d1        ; d0 = l'adresse du prem plan
        move.w   #prof-1,d2      ; adresses de bitp0h
ibrush   move.l   a5,d0
        bsr      metadr
        bsr      metadr
        lea      NB_BYTE_LIGNE(a5),a5
        dbf      d2,ibrush
        rts

metadr    swap     d0             ; c'est ici qu'on le met dans la coplist
        move.w   d1,(a0)+
        move.w   d0,(a0)+
        addq.w   #2,d1
        rts

; Le coeur du programme: la routine de trace d'ellipses
; utilise les variables rayona,rayonb,circlex et circley
; ainsi que bitplane 0
draw_ellipse:
        move.w   rayona(pc),d6
        and.w    #$ff,d6 ; Pour eviter les ellipses trop grandes
        beq      out_ellipse ; si A=0 on sort
        move.w   rayonb(pc),d7
        and.w    #$ff,d7 ; pour les ellipses trop grandes!
        beq      out_ellipse ; si B=0 on sort
        clr.l    d4
        move.w   d6,d4 ; d4=A
        lsl.l    #8,d4
        divu     d7,d4 ; d4=A/B
        mulu     d4,d4 ; d4=(A/B)carre
        move.l   d4,a0 ; d4=alpha
        neg.l    d4
        add.l    a0,a0
        clr.l    d0
        move.w   d6,d0 ; d0=a=x
        move.l   d0,d1
        add.w    d1,d1 ; d1=delta=2*x-1
        swap     d1
        move.l   bitplane0(pc),a1 ; pointeur sur le buffer ecran
        move.w   circley(pc),d2 ; d2 contiendra la ligne haute
        move.w   d2,d5
        move.w   d2,d3 ; d3 la ligne basse
        subq.w   #1,d3
        mulu     #NB_BYTE_LIGNE,d5 ; d5 contient l'offset en octets
        add.l    d5,a1 ; a1 pointe sur la ligne du bas
        move.l   a1,a2 ; a2 pointe sur la ligne du haut
        lea      -40(a1),a1
        subq.w   #1,d7 ; a cause du dbf!
        move.w   circlex(pc),a5 ; a5 contient coord x du centre
loop_ellipse2:
        add.l    a0,d4 ; alpha=alpha+2*(a*a)/(b*b)
        sub.l    d4,d1 ; delta=delta-alpha
        bpl.s    delta_pos2 ; delta>0 on affiche le point
        swap     d1 ; sinon on itere
        delta_neg2:

```

```

        subq.w   #1,d0 ; x=x-1
        bmi.s    out_ellipse_before2 ; x<0 ->fin de l'ellipse
        add.w    d0,d1 ; \
        add.w    d0,d1 ; --delta+2*x
        subq.w   #1,d1 ; delta+2*x-1
        bmi.s    delta_neg2 ; tant que delta<0
        swap     d1 ; d1=d1*65536
delta_pos2:
        move.w   a5,d5 ; d5=centrex
        sub.w    d0,d5 ; d5=centrex-x
        move.w   d5,d6 ; d6 contient cette position
        lsr.w    #3,d6 ; on se prepare a l'affichage
        and.b    #7,d5 ; de ce point
        move.b    #80,d3
        lsr.b    d5,d3
        eor.b    d3,(a1,d6.w) ; on l'affiche en haut a gauche
        eor.b    d3,(a2,d6.w) ; et son symetrique bas/gauche
        move.w   a5,d5 ; d5=centrex
        add.w    d0,d5 ; d5=centrex+x
        move.w   d5,d6 ; d6 contient cette position
        lsr.w    #3,d6 ; on se prepare la aussi
        and.b    #7,d5 ; a l'affichage de ce point
        move.b    #80,d3
        lsr.b    d5,d3
        eor.b    d3,(a1,d6.w) ; on l'affiche en haut a droite
        eor.b    d3,(a2,d6.w) ; et en bas a droite
        lea      -40(a1),a1 ; prochaine ligne haut
        lea      40(a2),a2 ; prochaine ligne bas
        dbf      d7,loop_ellipse2 ; tant qu'il y aura des lignes...
out_ellipse_before2:
        bsr      blit_circle ; un petit coup de blitter
out_ellipse:
        rts

; Remplissage au blitter. On remplit tout l'ecran, même
; si ce n'est pas necessaire.
blit_circle:
        move.l   bitplane0(pc),a0 ; pointeur sur l'ecran
        ; on se place a la fin de l'ecran (mode 'descendant')
        add.l    #NB_LIGNE*prof*Nb_BYTE_LIGNE,a0
        lea      CUSTOM,a5
        bsr      waitblit
        move.l   a0,bldpt(a5) ; source
        move.l   a0,bltapt(a5) ; destination
        move.l   #-1,bltafm(a5) ; masque
        move.w   #09f0,bltcon0(a5) ; D=A
        move.w   #000a,bltcon1(a5) ; mode remplissage et descente
        clr.w    bltdmod(a5) ; modulo=0
        clr.w    bltamod(a5) ;
        move.w   #NB_LIGNE*prof<<6+Nb_BYTE_LIGNE/2,bltsize(a5)
        rts

rayona:dc.w    100
rayonb:dc.w    80
circlex:dc.w   160
circley:dc.w   100

; EFFACEMENT AU BLITTER D'UN BITMAP
bltclear:
        bsr      waitblit
        move.l   bitplane0(pc),bldpt(a5)
        move.w   #01f0,bltcon0(a5) ; D=A en desactivant A
        clr.w    bltcon1(a5)
        clr.w    bltadat(a5) ; donnee=0
        clr.w    bltdmod(a5) ; modulo=0
        move.w   #NB_LIGNE*prof<<6+Nb_BYTE_LIGNE/2,bltsize(a5)
        rts

waitblit:
        btst     #14,dmaconr(a5)
        bne.s    waitblit
        rts

graphname    GRAFNAME
even
_GfxBase:    dcb.l    1
oldcop:      dc.l     0
bitplane0:   dcb.l    1
abitmap0:    dcb.l    1
bbitmap0:    dcb.l    1
flagcop:     dc.b     0

tailleplan=Nb_BYTE_LIGNE*40
taillescr=tailleplan*prof

mycop        section coplist,DATA_C
dc.w          $1081,$fffe
dc.w          $180,0,$182,$ff
dc.w          $008e,$2581
dc.w          $0090,$40c1
dc.w          $0092,$0038
dc.w          $0094,$00cc
dc.w          $102,0
dc.w          $108,(prof-1)*40
dc.w          $10a,(prof-1)*40
dc.w          $009c,$8010
dc.w          $3101,$fffe
dc.w          $0100,prof<<12

p_bitpl      dcb.w    prof*4,0
dc.l          $f0dffffe
dc.l          $01000000
dc.l          $fffffffe

fincop4:
END

```

Par Thomas Landspurg

DEMOS : LES ETOILES 3D

Salut, coucou et bonjour ! Pour bien commencer l'année, vous allez avoir droit au programme qui a permis à beaucoup de faire de très jolis effets avec les étoiles, autrement dit vous avez compris : nous allons faire des étoiles en 3D.

Le principe est simple : on choisit des points et on les fait bouger (ha, ha, mooorrrreeee de rire ! (NLDR : je sais, Jérôme, je sais : mort s'écrit avec un 't', pas un 'e', mais bon, si ça peut te faire plaisir...)). Pour les faire bouger, il faut se servir des touches du pavé numérique : les touches de gauche servent à décrémenter la vitesse, les touches du milieu à la stopper et les touches de droite à l'incrémenter. La première ligne de chiffres contrôle la composante X de la vitesse, la seconde, la composante Y et la troisième, la composante Z.

Bon allez, sérieusement, la première étape consiste à choisir une centaine d'étoiles au hasard (sous entendu, les coordonnées X, Y et Z) dans un cube de côté 1024. D'ailleurs, vous trouverez dans le source une routine de hasard à peu près convenable (la technique n'est pas de moi, mais j'ai oublié de qui c'était. De toute façon, je vous expliquerai la routine ainsi que le principe plus tard). Donc, l'observateur se situe perpendiculairement à l'une des faces du cube et la ruse consiste à bouger les étoiles elles-mêmes, et non pas le point d'observation. Avantage : la gestion en est grandement simplifiée.

En effet, si vous allez vers l'avant, les étoiles vont avancer vers vous et vous allez fatalement rencontrer la face opposée du cube. Conclusion, vous allez voir de moins en moins d'étoiles au fur et à mesure que vous avancerez vers la face. Il faut donc éviter ce problème, sinon ça fait pas très beau. D'où l'avantage de la technique : en effet, les étoiles qui vont nous dépasser et donc se trouver derrière nous, vont être remises de l'autre côté. Exemple sur une droite : je suis en 1, l'étoile est en 4 ; quand j'avance, l'étoile vient en 3, puis en 2, en 1, puis en 0. Au moment où elle atteint le 0, on la remet en 4 et donc je vois toujours une étoile arriver vers moi. En faisant cela pour les 150 étoiles, vous obtenez le mega extra super d'enfer hyper géant... (à compléter) effet d'univers 3D alors qu'en fait, vous restez bêtement devant une vulgaire boîte de 1024 de côté.

Vous devez vous demander pourquoi j'ai choisi 1024 comme taille du cube ? Oui, c'est vrai : pourquoi, Maître ? Hé bien, Disciple, pour deux raisons principales (les 36037 autres étant mineures, quoi que la 8342 soit limite) : la première est que 1024 a l'immense privilège d'être une puissance de 2 (2 puissance 6 exactement) et qu'en informatique, tout ce qui est relié aux puissances de 2 est d'une grande facilité de gestion. Dans notre cas, nous avons pu, grâce à ce chiffre particulier, ne pas tester si le point sort ou non du cube : en effet, nous n'avons qu'à faire sur chaque coordonnée une simple opération logique de type AND 1023, et la méchante coordonnée qui s'aventure en 1027 par exemple est immédiatement remise en 3. La seconde raison est que 150 étoiles est juste le nombre qu'il faut dans une cube de 1024 de côté, car si on avait fait un cube de 2 puissance 32 de côté, on aurait vu une étoile tous les 3 ans, ce qui aurait nettement atténué l'effet du déplacement.

Je vais quand même vous expliquer la 8342ième raison, car elle a son importance. En effet, quand on fait joujou avec la 3D, l'échelle des coordonnées n'est pas du tout imposée par le calcul ou un autre facteur, elle est choisie selon le goût du programmeur lors de la projection (transformation des coordonnées 3D en 2D). J'aurais très bien pu choisir un carré de 2 puissance 24 de côté et adapter les coordonnées des points en conséquence (NLDR : et augmenter le nombre d'étoiles, ça n'aurait pas marché ?).

Pour ce qui est de la gestion de la couleur en fonction de la distance, cela se révèle assez simple : on se sert de la coordonnée Z de chaque étoile pour connaître sa distance d'avec l'observateur. Les puristes auraient peut-être préféré faire le calcul suivant $x + y + z$. En effet,

cela aurait donné un chiffre proportionnel à la distance exacte entre chaque étoile et le centre de l'écran, c'est-à-dire l'observateur... En utilisant la coordonnée Z, on obtient la distance entre chaque étoile et le plan représenté par l'écran. L'erreur est donc réelle mais elle est si minime que personne ne s'en aperçoit. De plus, on gagne un temps-machine considérable.

Pour gagner du temps lors de l'affichage (qui prend quand même environ 50% du temps d'exécution total), j'ai fait une routine de point pour chacune des quatre couleurs utilisées. On y perd en souplesse, mais on y gagne en rapidité et dans une démo, le temps machine prime. Autre astuce qui alourdit le programme : chaque étoile doit bien évidemment être affichée puis effacée, et pour disposer de 4 couleurs d'étoiles (plus le fond), il faut 3 bitplans. Pour gagner un peu de temps, j'ai défini 4 bitplans et j'installe la palette de telle sorte que chacune des 4 couleurs corresponde à un bitplan différent. Grâce à cela, je trace et efface chaque étoile sur un seul plan, d'où un gain relativement important. Autre astuce (déjà utilisée des milliers de fois), je pré-calcule les adresses de début de lignes ; cela me permet de ne pas avoir à faire une multiplication à chaque fois que je trace un point.

Pour ceux qui n'ont jamais fait de 3D, je vais expliquer comment on parvient à rendre l'effet de perspective. Tout part d'un principe simple et connu de tous : la perspective est une notion selon laquelle plus un objet est loin de l'observateur, plus il lui apparaît petit. C'est donc cette notion qu'il faut simuler. Dans un premier temps, on attribue à chaque étoile une coordonnée en 3 composantes (X, Y, Z), ce qui permet de la situer dans l'espace. X représente l'horizontale, Y, la verticale et Z la profondeur. On s'aperçoit assez vite qu'il faut faire intervenir Z dans les calculs, car la taille de l'objet dépend directement de la distance qui le sépare de l'observateur. Le calcul que tout le monde utilise est la division de x et y par z (je ne sais pas s'il est juste par rapport à la réalité, mais en tout cas, il correspond suffisamment bien pour que personne ne voie d'erreur).

Prenons un exemple concret : on est dans une voiture roulant en permanence à 50km/h et on regarde un panneau routier se trouvant 300 mètres devant. A 300 m, le panneau grossit lentement, à 100 m, il grossit plus vite et à 20 m encore plus vite, donc le grossissement ne s'opère pas de façon linéaire et seule la division permet de donner l'effet voulu. Detail : il faut avant de diviser x et y, les multiplier par un gros chiffre, sinon on obtiendra un objet de taille minuscule.

La phrase du mois : "Ton meilleur repos est le sommeil. Tu le provoques souvent intentionnellement, cependant tu crains la mort, qui n'est rien de plus."

Aujourd'hui, je me sens généreux, je vais donc vous en donner une deuxième : "I'm not prejudice, I hate everybody."

* AUTEUR: j.etienne
* SUJET: déplacement d'étoiles en pseudo 3d
* ASSEMBLEUR: devpack 2.14
* UTILISATION: Les touches du pave numerique

```
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
SCR_SIZE =       BPL_SIZE*BPL_DEPTH

NB_STAR  =        180

vsync: macro
.wait_vsync@:
    move.l vposr(a5),d0
    and.l  #$1ff00,d0
    cmp.l  #\1*$100,d0
    bne.s  .wait_vsync@
endm

*****  programme principal  *****
move.l   (execbase).w,a6
lea      custom,a5
CALLEXECForbid
```



```

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*192
move.w #$0038,ddfstrt(a5)      * > avec un mot en plus
move.w #$00d0,ddfstop(a5)      * / pour le shift

bsr BUILD_COPLIST

move.l coplist_adr,copllc(a5)  * > run my coplist
clr.w copjmp1(a5)              * /

move.w #$8380,dmacon(a5)      * blitter,copper,bitplane

* init color map
COLOR_1 = $fff
COLOR_2 = $bbb
COLOR_3 = $888
COLOR_4 = $444

DUMMY SET 0
move.w #0,color+DUMMY(a5)
DUMMY SET DUMMY+2
move.w #COLOR_4,color+DUMMY(a5)
DUMMY SET DUMMY+2
rept 2
move.w #COLOR_3,color+DUMMY(a5)
DUMMY SET DUMMY+2
endr
rept 4
move.w #COLOR_2,color+DUMMY(a5)
DUMMY SET DUMMY+2
endr
rept 8
move.w #COLOR_1,color+DUMMY(a5)
DUMMY SET DUMMY+2
endr

move.w #3,SPEED_X              * init the first speed
move.w #3,SPEED_Y
move.w #-4,SPEED_Z
bsr INIT_TAB_STAR              * init the first coords

MAIN_LOOP:
vsync $f
bsr BUILD_COPLIST
* double buffering for tab
move.l LOG_TAB_ADR_POINT,d0
move.l PHY_TAB_ADR_POINT,LOG_TAB_ADR_POINT
move.l d0,PHY_TAB_ADR_POINT
* double buffering for screen
move.l LOG_SCR_ADR,d0
move.l PHY_SCR_ADR,LOG_SCR_ADR
move.l d0,PHY_SCR_ADR

bsr GESTION_KEYBOARD
bsr CLEAR_STAR
bsr AFF_STAR

move.w #$f00,color(a5) * \
move.w #$2f,d0          * > barre de couleur pour
dbf d0,*                * > montrer le temps d'execution
clr.w color(a5)         * /

move.b $bfec01,d0      * \
not d0                  * >capture the key pressed
ror.b #1,d0             * / and put is code RAW in d0
cmp.b #$45,d0          * \
beq INIT_END           * / sort si on press sur esc
btst #6,$bfe001
bne MAIN_LOOP
bra INIT_END

CLEAR_STAR: * clears all the old stars which were on this screen
move.l LOG_TAB_ADR_POINT(pc),a0
tst.l (a0)
beq.s .END_CLEAR_STAR    * \ dont clear the page
move.l #NB_STAR-1,d0     * / the first stars
moveq #0,d1

.LOOP_CLEAR_STAR:
move.l (a0)+,a1
move.b d1,(a1)
dbf d0,.LOOP_CLEAR_STAR
.END_CLEAR_STAR
rts

*****
END_CLEAR_STAR

GESTION_KEYBOARD: * this routine computes the speed according to
the keyboard
ACC = 1
KEY_DEC_X = $3D
KEY_STP_X = $3e
KEY_INC_X = $3f
KEY_DEC_Y = $2d
KEY_STP_Y = $2e
KEY_INC_Y = $2f

```

```

KEY_DEC_Z = $1d
KEY_STP_Z = $1e
KEY_INC_Z = $1f

moveq #0,d0
move.b $bfec01,d0          * \
not d0                     * >capture the key pressed
ror.b #1,d0                * / and put is code RAW in d0

cmp.b #KEY_DEC_X,d0
bne.s .DONT_DEC_X
sub.w #ACC,SPEED_X
.DONT_DEC_X
cmp.b #KEY_STP_X,d0
bne.s .DONT_STP_X
clr.w SPEED_X
.DONT_STP_X
cmp.b #KEY_INC_X,d0
bne.s .DONT_INC_X
add.w #ACC,SPEED_X
.DONT_INC_X
cmp.b #KEY_DEC_Y,d0
bne.s .DONT_DEC_Y
sub.w #ACC,SPEED_Y
.DONT_DEC_Y
cmp.b #KEY_STP_Y,d0
bne.s .DONT_STP_Y
clr.w SPEED_Y
.DONT_STP_Y
cmp.b #KEY_INC_Y,d0
bne.s .DONT_INC_Y
add.w #ACC,SPEED_Y
.DONT_INC_Y
cmp.b #KEY_DEC_Z,d0
bne.s .DONT_DEC_Z
sub.w #ACC,SPEED_Z
.DONT_DEC_Z
cmp.b #KEY_STP_Z,d0
bne.s .DONT_STP_Z
clr.w SPEED_Z
.DONT_STP_Z
cmp.b #KEY_INC_Z,d0
bne.s .DONT_INC_Z
add.w #ACC,SPEED_Z
.DONT_INC_Z
rts

SPEED_X ds.w 1
SPEED_Y ds.w 1
SPEED_Z ds.w 1
*****
END_GESTION_KEYBOARD

AFF_STAR: *****
* This routine computes the coord and displays each star
DELTA_Z = 250
move.l #NB_STAR-1,d7
move.l LOG_TAB_ADR_POINT(pc),a1
lea TAB_STAR(pc),a2
lea SPEED_X(pc),a3
move.w #$03ff,d4
move.w #$01ff,d3

.LOOP_EACH_STAR:
* compute the new coord for each star
movem.w (a2),d0-d2
add.w (a3),d0
add.w 2(a3),d1
add.w 4(a3),d2
and.w d4,d0          * \ quand une etoile sur du cube,
and.w d4,d1          * > on la remet dedans.
and.w d4,d2          * /
move.w d0,(a2)+
move.w d1,(a2)+
move.w d2,(a2)+
* compute the transformation in plane
sub.w d3,d0
sub.w d3,d1
ext.l d0
ext.l d1
asl.l #7,d1
asl.l #7,d0
add.w #DELTA_Z,d2
divs d2,d0
divs d2,d1
add.w #BPL_X/2,d0
add.w #BPL_Y/2,d1
* clip the point: test if the point is in or out of screen
tst.w d0
blt .POINT_NOT_ON_SCREEN
cmp.w #BPL_X-1,d0
bgt .POINT_NOT_ON_SCREEN
tst.w d1
blt .POINT_NOT_ON_SCREEN
cmp.w #BPL_Y-1,d1
bgt .POINT_NOT_ON_SCREEN
* compute the adr and the shift for the point
move.l LOG_SCR_ADR,a0
lsl.w #2,d1
add.l ADR_LINE_ACORDING_TO_Y(pc,d1.w),a0
move.w d0,d1
lsr.w #3,d1
lea (a0,d1.w),a0
and.w #%111,d0

```



```

neg.w    d0
addq.w   #7,d0

* display the point
* the color is chosen according to the Z coord.
* test if this color 1
  cmp.w   #1*1024/4+DELTA_Z,d2
  bgt.s   .NOT_COLOR_1
  lea     3*BPL_WIDTH(a0),a0
  move.l  a0,(a1)+      * save adr to clear point later
  bset    d0,(a0)
  dbf     d7,.LOOP_EACH_STAR
  rts
.NOT_COLOR_1

* test for color 2
  cmp.w   #2*1024/4+DELTA_Z,d2
  bgt.s   .NOT_COLOR_2
  lea     2*BPL_WIDTH(a0),a0
  move.l  a0,(a1)+      * save adr to clear point later
  bset    d0,(a0)
  dbf     d7,.LOOP_EACH_STAR
  rts
.NOT_COLOR_2

* test for color 3
  cmp.w   #3*1024/4+DELTA_Z,d2
  bgt.s   .NOT_COLOR_3
  lea     1*BPL_WIDTH(a0),a0
  move.l  a0,(a1)+      * save adr to clear point later
  bset    d0,(a0)
  dbf     d7,.LOOP_EACH_STAR
  rts
.NOT_COLOR_3

* color 4.
  move.l  a0,(a1)+      * save adr to clear point later
  bset    d0,(a0)
  dbf     d7,.LOOP_EACH_STAR
  rts

.POINT_NOT_ON_SCREEN
  move.l  LOG_SCR_ADR,(a1)+
  dbf     d7,.LOOP_EACH_STAR
  rts

***** END_AFF_STAR
* Ce tableau contient l'adresse de chaque ligne relative au debut
de l'ecran
ADR_LINE ACORDING_TO_Y:
DUMMY SET 0
REPT BPL_Y
dc.l DUMMY
DUMMY SET DUMMY+BPL_WIDTH*BPL_DEPTH
ENDR

INIT_TAB_STAR:
*****
* This routine computes the coords X Y Z of each star
  bsr    AM_INIT_RANDOM
  lea     TAB_STAR(pc),a0
  move.w  #NB_STAR-1,d1

.LOOP_INIT_STAR:
  bsr    RANDOM          * choose X at random
  and.l  #$3ff,d0
  move.w  d0,(a0)+
  bsr    RANDOM          * choose Y at random
  and.l  #$3ff,d0
  move.w  d0,(a0)+
  bsr    RANDOM          * choose Z at random
  and.l  #$3ff,d0
  move.w  d0,(a0)+
  dbf     d1,.LOOP_INIT_STAR
  rts
*****
END_INIT_TAB_STAR

RANDOM:
*****
* OUT: d0 = Random Number 24 bits

* t= a * ( z % q ) - r * int ( z / q )
* if t > 0
*   z=t
* else
*   z=t+m
* endif
* pour plus d'explications, voir st magazine n~45 page 108

RND_A equ $41a7
RND_M equ $7ffffff
RND_Q equ $f31d
RND_R equ $8b14

movem.l d1-d3,-(sp)
move.l  #RND_Q,d0
move.l  RND_Z,d1
moveq   #0,d3
divu    d0,d1
bvc.s   .RESULT
move.l  d1,d2
clr.w   d1
swap    d0,d1
mulu    #RND_A,d0
move.w  d1,d3
swap    d3
mulu    #RND_R,d3
swap    d3
clr.w   d3
mulu    #RND_R,d1
add.l   d1,d3
sub.l   d3,d0
move.l  d0,RND_Z
movem.l (sp)+,d1-d3
rts

.RESULT:
  move.l  d1,d0
  swap    d1
  move.w  d3,d1
  swap    d1
  clr.w   d0
  swap    d0
  mulu    #RND_A,d0
  move.w  d1,d3
  swap    d3
  mulu    #RND_R,d3
  swap    d3
  clr.w   d3
  mulu    #RND_R,d1
  add.l   d1,d3
  sub.l   d3,d0
  move.l  d0,RND_Z
  movem.l (sp)+,d1-d3
  rts

AM_INIT_RANDOM:
*****
* cette routine initialise la semence de RANDOM
* Il est imperativement necessaire de lancer cette routine avant
* de se servir de RANDOM sinon RANDOM donnera toujours la meme
* suite de chiffres a chaque fois que vous lancerez le programme
  move.l  VPOS+CUSTOM,d0
  and.l   #RND_M,d0
  cmp.l   #RND_M-1,d0
  blt.s   .SEMENCE_LT_RANDOM_M
  subq.l  #2,d0
.SEMENCE_LT_RANDOM_M:
  addq.l  #1,d0
  move.l  d0,RND_Z
  rts

RND_Z: ds.l 1          * BUG SI ON PLACE CETTE LIGNE PLUS HAUT
                          * OPT o+ code en absolu court !!

END_AM_INIT_RANDOM

INIT_END: * reactivation de l'ancienne coplist
  lea     GfxName(pc),a1 * nom de la library ds a1
  moveq   #0,d0          * version 0 (the last)
  CALLEXEC OpenLibrary   * 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   copjmpl(a5)    * l'old coplist et lancement
  move.w  #$83e0,dmacon(a5) * activation des canaux dma
  CALLEXEC Permit        * multi switching autorise
  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

TAB_STAR:
LOG_TAB_ADR_POINT: ds.w 3*Nb_STAR
PHY_TAB_ADR_POINT: dc.l TAB_ADR_POINT_1
TAB_ADR_POINT_1:   ds.l Nb_STAR
TAB_ADR_POINT_2:   ds.l Nb_STAR

***** datas
* variables
LOG_SCR_ADR: dc.l ecran1
PHY_SCR_ADR: dc.l ecran2
COPLIST_ADR: dc.l coplist

section ZONE_CHIP,BSS_C

ecran1 ds.l (scr_size*2)/4
ecran2 ds.l (scr_size*2)/4
coplist ds.l 1000*4 * taille inconnue, on prevoit gros
end

```

```

divu    d0,d1
move.w  d1,d3
move.w  d2,d1
divu    d0,d1

.RESULT:
  move.l  d1,d0
  swap    d1
  move.w  d3,d1
  swap    d1
  clr.w   d0
  swap    d0
  mulu    #RND_A,d0
  move.w  d1,d3
  swap    d3
  mulu    #RND_R,d3
  swap    d3
  clr.w   d3
  mulu    #RND_R,d1
  add.l   d1,d3
  sub.l   d3,d0
  move.l  d0,RND_Z
  movem.l (sp)+,d1-d3
  rts

AM_INIT_RANDOM:
*****
* cette routine initialise la semence de RANDOM
* Il est imperativement necessaire de lancer cette routine avant
* de se servir de RANDOM sinon RANDOM donnera toujours la meme
* suite de chiffres a chaque fois que vous lancerez le programme
  move.l  VPOS+CUSTOM,d0
  and.l   #RND_M,d0
  cmp.l   #RND_M-1,d0
  blt.s   .SEMENCE_LT_RANDOM_M
  subq.l  #2,d0
.SEMENCE_LT_RANDOM_M:
  addq.l  #1,d0
  move.l  d0,RND_Z
  rts

RND_Z: ds.l 1          * BUG SI ON PLACE CETTE LIGNE PLUS HAUT
                          * OPT o+ code en absolu court !!

END_AM_INIT_RANDOM

INIT_END: * reactivation de l'ancienne coplist
  lea     GfxName(pc),a1 * nom de la library ds a1
  moveq   #0,d0          * version 0 (the last)
  CALLEXEC OpenLibrary   * 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   copjmpl(a5)    * l'old coplist et lancement
  move.w  #$83e0,dmacon(a5) * activation des canaux dma
  CALLEXEC Permit        * multi switching autorise
  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

TAB_STAR:
LOG_TAB_ADR_POINT: ds.w 3*Nb_STAR
PHY_TAB_ADR_POINT: dc.l TAB_ADR_POINT_1
TAB_ADR_POINT_1:   ds.l Nb_STAR
TAB_ADR_POINT_2:   ds.l Nb_STAR

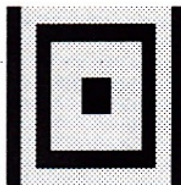
***** datas
* variables
LOG_SCR_ADR: dc.l ecran1
PHY_SCR_ADR: dc.l ecran2
COPLIST_ADR: dc.l coplist

section ZONE_CHIP,BSS_C

ecran1 ds.l (scr_size*2)/4
ecran2 ds.l (scr_size*2)/4
coplist ds.l 1000*4 * taille inconnue, on prevoit gros
end

```

Par Jérôme Etienne



LES ROUTINETTES DE FRANCOIS

Ce mois-ci, quatre routinettes pour le prix d'une seule : l'ANT vous en donne toujours plus !

Les Basics (GFA et AMOS) offrent une belle fonction : `=Exist()`, qui permet de savoir si un fichier existe réellement avant de le charger. Je vous propose donc aujourd'hui la version routinette de cette fonction : vous lui donnez un nom de fichier, et elle retourne vrai si ce fichier existe, ou faux s'il n'existe pas.

La routinette détourne pour un bref instant le requester du système : on peut ainsi demander la présence d'un nom de disque, ou d'un device. Ultra-pratique n'est-il pas ?

Plusieurs points d'entrée sont à votre disposition :

* `_IntOpen` : cette routine ouvre l'intuition.library en positionnant un flag (Wb2.0). Ce flag n'est à -1 que lorsque l'Amiga utilise le Workbench 2.0 (ou une version postérieure).

* `_Exist` : la fonction elle-même.

* `_ReqOff` : arrêt du system-request et passage en mode Forbid. Veillez à ne pas rester dans ce mode trop longtemps.

* `_ReqOn` : retour au requester normal et au mode Permit.

Pour appeler ces routinettes, vous devez comme d'habitude mettre l'adresse de la page de données en A5. L'exemple d'utilisation de ces routinettes est une nouvelle fonction CLI, recherchant le fichier (ou le disque) donné en argument. Le résultat est affiché dans la fenêtre courante.

On se retrouve le mois prochain pour de nouvelles routinettes, nettes.

```
*-----
*      ROUTINETTES numéro 3
*
*      _Exist: recherche un fichier sans requester
*      _ReqOff: plus de requester
*      _ReqOn:  requester de nouveau là
*      _IntOpen: ouvre intuition.library avec détection de
*                  version
*
*      Par François Lionet, paru dans l'A.N.T. du 1er Février 1992
*-----
*----- Definition de variables système
*
* Appel du dos
DosCall MACRO
    move.l    a6, -(sp)
    move.l    DosBase-Dt(a5), a6
    jsr       \1(a6)
    move.l    (sp)+, a6
ENDM

* Appel de fonction exec
ExeCall MACRO
    move.l    a6, -(sp)
    move.l    $4.w, a6
    jsr       \1(a6)
    move.l    (sp)+, a6
ENDM

* Offsets des fonctions dos
_LVOLock      equ    -84
_LVOUNLock    equ    -90
_LVOWrite     equ    -48
_LVOWOutput   equ    -60

* Offset des fonctions EXEC
_LVOPenLibrary equ    -552
_LVOCloseLibrary equ    -414
_LVOSetFunction equ    -420
_LVOForbid     equ    -132
_LVOPermit     equ    -138

* Offset des fonctions intuition
_LVOAutoRequest equ    -$15c
_LVOEasyRequest equ    -$24c

*----- Demonstration
Start

* Adresse de la base des données
lea         Dt(pc), a5
```

```
* Copie la ligne de commande dans un buffer
subq.w      #2, d0                ; Attention, une ligne de
                                   ; commande est terminée pas
                                   ; $0A, ne pas le copier !

bmi         .Err2
lea         Buffer-Dt(a5), a1
.Loop       move.b    (a0)+, (a1)+
dbra        d0, .Loop
clr.b       (a0)+

* Ouvre intuition
bsr         _IntOpen
beq.s       .Err2

* Ouvre le dos
lea         DosName-Dt(a5), a1
moveq       #0, d0
ExeCall     _LVOPenLibrary
move.l      d0, DosBase-Dt(a5)    ; Une erreur est très
beq.s       .Err1                ; improbable!

* Appelle la fonction
lea         Buffer-Dt(a5), a0
bsr         _Exist

* Imprime la réponse sur le canal standart
lea         Buffer-Dt(a5), a1
lea         Existe-Dt(a5), a0
moveq       #LExiste, d3
tst.l       d0
bne.s       .Skip2
lea         ExistePas-Dt(a5), a0
moveq       #LExistePas, d3
.Skip2      move.l     a0, d2
DosCall     _LVOWOutput           ; Demande le canal de sortie
move.l      d0, d1
DosCall     _LVOWrite

* Ferme le dos
move.l      DosBase-Dt(a5), a1
ExeCall     _LVOCloseLibrary

* Ferme intuition
.Err1      move.l     IntBase-Dt(a5), a1
ExeCall     _LVOCloseLibrary

* Sortie
.Err2      moveq       #0, d0
rts

*----- Routinette 1: =Exist
* Entree
*
*      A0= Nom du fichier
*
* Sortie
*
*      D0= Faux (0) si fichier inexistant
*      D0= Vrai (-1) si fichier trouve
_Exist
    movem.l    d1/a0-a1, -(sp)

* Plus de requester
bsr         _ReqOff

* Essai de faire un lock sur le fichier
move.l      a0, d1
DosCall     _LVOLock
move.l      d0, d2

* Remet le requester, sans attendre
bsr         _ReqOn

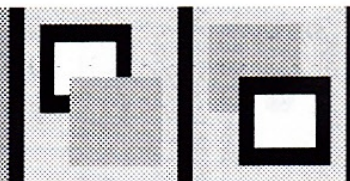
* Enleve le lock si il existe
move.l      d2, d1
beq.s       .NoLock
DosCall     _LVOUNLock
.NoLock

* Retourne la réponse
move.l      d2, d0
beq.s       .Fin
moveq       #-1, d0
.Fin        movem.l    (sp)+, d1/a0-a1
rts

*----- Routinette 2: No Requester
* Entree:
*
*      Néant
*
* Sortie:
*
*      Néant
*
_ReqOff
    movem.l    a0-a1/d0-d1, -(sp)

* Inhibe le multitache
ExeCall     _LVOForbid

* Branche le faux requester
lea         _ReqNul(pc), a0
```

```

        bsr.s    _SetReq
        move.l   d0,ReqSave-Dt(a5)

        movem.l (sp)+,a0-a1/d0-d1
        rts

*      Faux requester: ramene toujours "CANCEL"
*
_RegNul
        moveq    #0,d0
        rts

*----- Routinette 3: Remet le requester
* Entree:
*
* Sortie:
*
*      Néant
*
_RegOn
        movem.l  a0-a1/d0-d1,-(sp)

* Remet l'ancien requester
        move.l   ReqSave-Dt(a5),a0
        bsr.s    _SetReq

* Re-autorise le multitache.
        ExeCall  _LVOPermit

        movem.l  (sp)+,a0-a1/d0-d1
        rts

*----- Routinette 4: Ouvre intuition
*      en trouvant la version
* Entree:
*
* Sortie:
*
*      Néant
*
*      D0= 0 si ouverture ratée
*      D0<>0 si réussie
*
_IntOpen
        movem.l  a0-a1/d1-d2,-(sp)

* Essai l'ouverture de la version 2.0
        moveq    #-1,d2
        moveq    #36,d0
        lea      IntName-Dt(a5),a1
        ExeCall  _LVOOpenLibrary
        tst.l    d0
        bne.s    .Skip

* Ca n'a pas marche: WB 1.3!
        moveq    #0,d2
        moveq    #0,d0
        lea      IntName-Dt(a5),a1
        ExeCall  _LVOOpenLibrary

* Stocke le flag et l'adresse de base
.Skip   move.w   d2,WB2.0-Dt(a5)
        move.l   d0,IntBase-Dt(a5)
        movem.l  (sp)+,a0-a1/d1-d2
        rts

***** Routine interne: fait le changement de vecteur
*      et retourne l'ancien...
_SetReq
        move.l   a0,d0
        move.l   IntBase-Dt(a5),a1
        lea      _LVOAutoRequest,a0
        tst.w    WB2.0-Dt(a5)
        beq.s    .Skip
        lea      _LVOEasyRequest,a0
.Skip   ExeCall  _LVOSetFunction
        rts

* La base des acces à la zone de donnee
Dt
WB2.0      dc.w    0
ReqSave    dc.l    0

DosBase    dc.l    0
IntBase    dc.l    0
DosName    dc.b    "dos.library",0
IntName    dc.b    "intuition.library",0

Existe     dc.b    "Existe !",10
ExistePas  dc.b    "N'existe pas !",10
Finito     dc.b    0

Buffer     ds.b    128

LExiste    equ     ExistePas-Existe
LExistePas equ     Finito-ExistePas

        even

```

DE LA BONNE VOLONTE, MAIS... PEUT MIEUX FAIRE !

Tonton François serait-il un adepte du vieil adage : "pourquoi faire simple quand on peut faire compliqué" ? Pour interdire les Requesters "Please insert volume XXXXX: in any drive" de l'AmigaDOS, il y a beaucoup plus simple. Et surtout, plus "Amiga Friendly".

Il suffit en effet de jeter un oeil à la structure Process que tout programme, à l'exception des tâches simples créées par *CreateTask()* ou équivalente, possède : à l'offset \$B8 (184 décimal) de cette structure, se trouve un champ nommé *pr_WindowPtr*, qui contient un pointeur sur la fenêtre dans laquelle AmigaDOS affichera au besoin ses Requesters. Ce champ contient normalement la valeur NULL, indiquant que c'est la fenêtre (et donc l'écran) du Workbench qu'il faudra utiliser. En mettant ici l'adresse d'une fenêtre à soi, on pourra faire en sorte que ces Requesters apparaissent dans le même écran que cette fenêtre, par exemple dans un Custom Screen.

Mais le plus intéressant est qu'en initialisant *pr_WindowPtr* à -1, aucun Requester ne sera affiché du tout, et AmigaDOS se comportera exactement comme si l'utilisateur avait cliqué sur Cancel. Il faudra bien entendu pas oublier de remettre ce champ dans l'état dans lequel on l'avait trouvé en arrivant (normalement NULL, donc, mais on ne sait jamais : autant sauver la valeur originale quelque part).

L'avantage de cette méthode, outre que c'est celle recommandée par les Rom Kernel Manuals, est qu'on n'est pas obligé d'interdire le multi-tâche et que les autres programmes essayant de cohabiter ne seront pas influencés : leurs Requesters à eux apparaîtront bel et bien... A moins qu'ils n'aient eux aussi usé du même truc.

Max

```

;
; NoReqAussiEtPlusPropre.s
; Par Max pour Amiga NewsTech
;
        INCDIR  "include:"
        INCLUDE "exec/exec_lib.i"
        INCLUDE "libraries/dos.i"
        INCLUDE "libraries/dosexterns.i"
        INCLUDE "libraries/dos_lib.i"

; Pour ceux qui aiment pas les includes : pr_WindowPtr = $B8
;
CALL    MACRO
        jsr    _LVO\1(a6)
        ENDM

;
Start   lea      dosname(pc),a1 ; Ouvre la dos.library
        moveq    #0,d0
        movea.l  $4.w,a6
        CALL     OpenLibrary
        move.l   d0,d7          ; DOSBase dans d7
        beq.s    NoDos
        suba.l   a1,a1          ; Où suis-je ?
        CALL     FindTask
        movea.l  d0,a4
        move.l   pr_WindowPtr(a4),d6 ; d6 = WindowPtr
        moveq    #-1,d0
        move.l   d0,pr_WindowPtr(a4) ; WindowPtr = -1
        lea      name(pc),a0     ; nom : 'PipiCaca:'
        move.l   a0,d1
        moveq    #ACCESS_READ,d2
        movea.l  d7,a6
        CALL     Lock            ; Y'a peu de chance qu'un
        move.l   d0,d1          ; disque de ce nom existe,
        beq.s    .nolock        ; mais on ne sait jamais !
        CALL     Unlock
        .nolock move.l  d6,pr_WindowPtr(a4) ; Remet WindowPtr
        movea.l  $4.w,a1          ; Referme la dos.library
        exg      a1,a6
        CALL     CloseLibrary
        NoDos    moveq    #0,d0    ; That's all folks !
        rts

;
dosname    dc.b    "dos.library",0
even
name       dc.b    "PipiCaca:",0
even

;
END

```

Par Tonton François Lionet



Cher ANT, je vous écris pour vous poser quelques questions et vous faire part de quelques remarques. Tout d'abord, pourquoi n'y a-t-il pas plus de pages ? Manquez-vous de journalistes ? Si oui, pourquoi ne pas vous adresser à des gens comme Xavier Leclerc ?

Est-ce que des rubriques sur le GFA et le Pascal sont réellement nécessaires ? Le C, l'assembleur et même l'AMOS ne couvrent-ils pas tous les besoins ? Par contre, une rubrique sur l'algorithmie est la bienvenue (enfin !).

Vous demandez l'avis des lecteurs sur les "carrés noirs". Si ça peut faire baisser le prix de la revue, ou augmenter le nombre de pages, ce n'est que bénéfique !

Merci de votre attention, bon courage et bonne année !

Christophe Laratte, 87410 le Palais sur Vienne

Nous avons déjà expliqué ici et à plusieurs reprises pourquoi l'ANT ne fait que 32 pages : nous n'y reviendrons donc pas, le débat est clos. Et non, nous ne manquons pas de journalistes, bien au contraire.

Pascal et GFA sont deux langages de programmation de haut niveau disponibles sur l'Amiga. A ce titre, nous pensons qu'ils méritent leur place dans nos pages, même si le nombre de lecteurs intéressés est forcément moins élevé que celui des fanas du C, de l'assembleur et de l'AMOS. Bien sûr, et comme d'habitude dans l'ANT, c'est avant tout vous qui décidez de l'avenir du magazine : si vous n'aimez pas ces rubriques (ou d'autres !), écrivez-nous en masse, on les arrêtera illico. Quant aux "carrés noirs" (pour la pub), l'idée fait son chemin. Il est évident que si elle aboutit (c'est-à-dire, si des annonceurs sont intéressés - et ils seront triés sur le volet), cela entraînera une augmentation du nombre de pages.

Messieurs, dans le numéro 21 d'avril, vous donnez les explications pour installer le compilateur C DICE. Je me le suis donc procuré et l'ai installé : cela marche, sauf que je n'ai pas les includes Commodore. J'ai donc écrit à Commodore France pour les acquérir et vous joins une photocopie de leur réponse... C'est déjà bien, il y a une réponse, je devrais prendre cela comme un honneur. Malheureusement, je le prends mal.

Lorsque j'ai fait mon chèque pour mon Amiga 2000, extension mémoire, disque dur et imprimante, on ne m'a pas demandé mon statut de développeur. Mon argent était bon. Aujourd'hui, je veux développer avec un logiciel du domaine public, alors là non, il faut être reconnu comme développeur par Commodore, et être béni par eux.

Au fait, si je mets 2800 F dans le commerce pour acheter le Lattice, j'aurai peut-être les includes, mais je serai toujours aussi nul en C.

Bon, Commodore me dit qu'une version des includes existe sur une disquette Fred Fish. Sauriez-vous me dire laquelle ?

Dominique Marquaille, 02000 Etouvelles

Il faut savoir que les includes ne sont pas du domaine public ; Commodore possède un Copyright dessus, et ne les vend qu'accompagnés du pack développeur (RKM, outils de développement, etc.). Leur réponse est donc normale, même si elle vous semble amère. Désolé, je viens de parcourir les 530 premières disquettes Fish (merci Aquarium !) et n'y ai pas trouvé d'autres includes que ceux fournis par Electronics Arts pour le format IFF (FF n°185). Si quelqu'un en a entendu parler, qu'ils nous écrive afin que tout le monde puisse en bénéficier.

Salut à tous ! Je vous écris pour signaler à Jérôme Etienne qu'un Modulo, c'est le reste d'une division, et qu'il est regrettable de voir une routine telle que ça (numéro 27, page 25, scroll "Kick Off") :

```
MOD MACRO
.MODULO_GO_ON\@:
    CMP.W    \2,\1
    BLT.S    .MODULO_IS_FINISH\@
    SUB.W    \2,\1
    BRA.S    .MODULO_GO_ON\@
.MODULO_IS_FINISH\@
ENDM
```

Alors que la vraie façon est :

```
MOD:    MACRO
        EXT.L    \1
        DIVU     \2,\1
        SWAP     \1
        ENDM
```

Je remarque aussi que l'auteur de cette routine devait être fatigué d'utiliser ses propres labels, puisqu'à la fin du listing on a droit à :

```
MOVE.W    #16*BPL_DEPTH*64+1,BLTSIZE(A5)
```

au lieu de :

```
MOVE.W    #BLK_HEIGHT*BPL_DEPTH*64+BLT_WIDTH/16,BLTSIZE(A5)
```

et ce à deux reprises, si bien que si l'on veut changer la taille d'un bloc :

```
BLK_WIDTH = 32
BLT_HEIGHT = 32
```

tout l'affichage déconne !

Autre petit détail : LSL et LSR insèrent des 0 lors des décalages (contrairement à ASR et ROR), donc dans le petit morceau de programme suivant (extrait de la routine BUILD_COPLIST du numéro 26, page 11) :

```
LSL.W    #8,D1
AND.W    #$FF00,D1
```

le masquage AND ne sert à rien ! Votre journal ayant un but d'apprentissage, il est dommage d'apprendre d'aussi mauvaises habitudes (surtout dans une routine qui doit tourner à 50 Hz).

Dernière chose : "source" est un nom féminin, donc on n'écrit pas "un source" mais "une source". Et ne me répondez pas que c'est un terme technique et qu'on en fait ce qu'on veut !

Je pense qu'on pourrait ne plus voir pareilles erreurs de débutant dans votre journal, qui est par ailleurs fort bien conçu. Bye.

Misfit (les Félés)

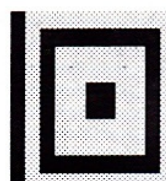
Merci pour toutes ces précisions. Jérôme Etienne a copié 100 fois "le modulo est le reste de la division" et 200 fois "lsl et lsr insèrent des 0". Il s'en souviendra.

"Source" est un nom féminin peut-être, mais lorsque l'on écrit "un source", on sous-entend "un fichier source" (puisque c'est l'expression exacte en informatique). Ce n'est pas un terme technique, c'est une abréviation. Et avant de donner des leçons de Français, vérifiez-donc votre propre orthographe...

Bon, ben voilà qui conclut notre Requester de ce mois. Je ne vous quitterai pas sans vous rappeler que cette rubrique est là pour (essayer de) répondre à tous vos problèmes de programmation, quels qu'ils soient. Pour les problèmes d'ordre pratique (imprimantes, quel disque dur choisir...), merci de plutôt adresser vos courriers à Amiga Revue, ils en feront bon usage. Evitez également de téléphoner, nous n'avons pas envie de passer nos journées suspendus au combiné !

Bonne année à tous !

Par Ok & Cancel 



Transactor

Règlement
et
Proposition d'Article

- 1.) La rubrique *Transactor* de *l'Amiga NewsTech* est la rubrique écrite par les lecteurs, pour les lecteurs. Son but est de leur permettre de s'exprimer librement sur tout sujet qui les passionne, dans quelques langage de programmation que ce soit. Seule restriction : la Rédaction doit avoir la possibilité de tester le bon fonctionnement de tous les programmes fournis, complets ou simples routines d'illustration. Non mais.
- 2.) Tout lecteur désirant participer à la rubrique *Transactor* devra envoyer son article sur disquette au format Amiga, en respectant les quotas définis aux paragraphes 4 et 5. Les disquettes ne seront pas retournées, sauf demande expresse et écrite de l'auteur, et encore.
- 3.) Tout article publié sera rémunéré au tarif forfaitaire de 1, 500 F HT les deux pages ; un article de plus de deux pages sera publié en plusieurs fois, à concurrence de trois épisodes maximum. Le paiement a lieu le mois suivant celui de la parution. Les auteurs des articles retenus pour parution seront directement prévenus par téléphone.
- 4.) Les articles doivent parvenir au format ASCII standard (utilisez un éditeur de textes ou l'option de sauvegarde "text only" ou "ASCII" de votre traitement de texte). Les programmes illustrant les articles, qu'ils soient complets ou de simples routines d'exemple, doivent également être fournis sous forme ASCII, et éventuellement en format "brut" le cas échéant (*Basic*, *AMOS*...). Le(s) fichier(s) exécutable(s) doivent se trouver sur la disquette. L'auteur doit fournir à l'ANT la possibilité de recompiler ou d'interpréter ses programmes sans difficulté (joindre au besoin une version "run-only" du langage utilisé).
- 5.) La taille totale du fichier ASCII donne le nombre de signes de votre article ; une page de l'ANT contient en moyenne 7, 000 signes de texte. Une colonne de listing contient 80 lignes de 65 caractères de large. Un programme de 160 lignes occupe donc une page entière. Les programmes en assembleur particulièrement longs peuvent être publiés en trois colonnes de 40 caractères chaque (240 lignes par page).
- 6.) Toute proposition d'article pour *Transactor* devra être accompagnée du bon ci-dessous, découpé ou photocopié, mais en aucun cas reproduit à la main.
- 7.) Toute proposition d'article pour *Transactor* entraîne l'acceptation par l'auteur que les programmes joints soient placés sur la disquette d'accompagnement de l'ANT. L'auteur doit préciser dans des commentaires en tête de listing s'il place son programme dans le Domaine Public ou s'il désire conserver son copyright. Aucun commentaire implique la mise en Domaine Public.
- 8.) Ni l'ANT, ni *Commodore Revue SARL*, ni les auteurs collaborant à la rubrique *Transactor* ne sauraient être tenus pour responsables en cas de dommages quelconques survenus à l'ordinateur, suite à une mauvaise utilisation des documents (textes et programmes) publiés dans cette rubrique.
- 9.) Toute proposition d'article pour *Transactor* entraîne l'entière acceptation par l'auteur de ce règlement, et sans discuter.

Abonnement à L'ANT et Anciens Numéros

C'est nouveau, ça vient de sortir : dorénavant, si vous êtes déjà abonné, vous pouvez parrainer un nouveau venu dans la grande famille de l'ANT et gagner des cadeaux offerts par CIS.

Si votre abonnement arrive à échéance, n'oubliez pas de le renouveler le plus tôt possible. Ce serait quand même dommage de rater un ou plusieurs numéros !

Ca aussi c'est nouveau, vous pouvez commander les anciens numéros de l'ANT à la pièce, afin de compléter votre collection. Tout est indiqué dans le bon de commande ci-dessous.

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*. 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
- ☐ 1 an (11 numéros) avec disquette 600 FF
- ☐ (cochez cette case pour un ré-abonnement)

• *Je commande les anciens numéros suivants :*

- | | |
|--|--|
| ☐ n°20 (☐ avec disquette) | ☐ n°25 (☐ 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 F (60 F avec disquette), pour un total de

F.

Nom :
Prénom :
Adresse :
Code Postal :
Ville :
Pays :