Programmez sur **Atari ST** 30 ans après

Frédéric Sagez
Codeur du groupe
NoExtra-Team
fsagez@gmail.com

Fan de lignes de Punchs écrites en GFA-BASIC ou de codes assembleurs prodigués par un Jedi pour faire de beaux effets ? Vous avez eu la fièvre de la programmation sur votre ST lors de votre adolescence, il est toujours temps de s'y remettre. « Chérie, tu as mis où les ST Mag ? ». Euh…

Mais avant d'en arriver là, replongeons-nous quelques instants dans le code assembleur tout en étant aidé par un Framework, regardez comment est agencé un programme, faire tourner des effets, juste histoire de s'y remettre un peu plus facilement…

Et pour commencer...

Ce dont nous avons besoin dans le cas idéal : un Atari STF avec 512 ko ou 1 Méga de mémoire vive avec un écran couleur et une disquette contenant le programme DevPac ST pour compiler nos programmes.
A défaut d'être équipé du vrai hardware, il vous faudrait pour votre machine actuelle utiliser un émulateur tel que **Steem SSE** (https://sourceforge.net/projects/steemsse/) ou **HATARI** (https://hatari.tuxfamily.org/). La dernière version de Steem SSE permet d'émuler un Atari STF ou STE avec pas mal d'options avec une meilleure gestion du Shifter – partie graphique du ST – qui permet notamment de visualiser des démos ou des jeux utilisant le mode OVERSCAN du Hardware. Cet émulateur fonctionne uniquement sur Windows et peut utiliser des plugins. HATARI est un émulateur qui a fait du chemin, et qui, aujourd'hui, est arrivé à maturité ; fonctionnant sur MAC et PC, il offre aussi une foule d'options concernant la partie graphique est sonore plus la gestion de disque dur. Point important avant de commencer : la version du TOS (Operating System de l'Atari) est primordiale suivant que vous allez coder sur un Atari STF ou STE. Récupérez sur Internet une version 1.4FR pour écrire des programmes fonctionnant sur un Atari STF et une version 1.62FR spécifiquement sur STE. Il existe d'autres versions du TOS en cours de développement comme EmuTOS (http://emutos.sourceforge.net/) qui est un clone Open Source utilisé par défaut dans l'émulateur HATARI.

L'assembleur ? Kezako ?

L'assembleur est un langage de programmation constitué de jeux d'instructions représentés par des labels, mnémoniques, opérandes et commentaires qui sera compilé / traduit en langage machine pour être exécuté sur la machine cible. Pour écrire un programme en assembleur, vous avez besoin de connaître les différents compartiments ou mettre le code et les éléments pour que le programme se compile. Nous avons trois sections apparentes à indiquer : TEXT, DATA et BSS. La partie TEXT contient le code en assembleur. Elle est située au début du fichier. La partie DATA contient toutes les données « non modifiables » du programme (image, musique, etc.). La partie BSS sera la partie mémoire d'accès direct en lecture/écriture. Pour créer un exécutable et l'exécuter sur le bureau GEM, nous aurons besoin d'un compilateur, outil qui va convertir notre programme (appelé code source) en un exécutable binaire. Les codes sources utilisés sont des fichiers avec l'extension « .S ».
Dans notre cas nous allons utiliser le logiciel DEVPAC ST qui sert à la fois d'éditeur de texte, à compiler notre fichier en programme binaire ou à l'exécuter / déboguer dans une partie de la mémoire vive de la machine. Récupérer le répertoire DEVPAC et son contenu dans le repository et copiez-le à la racine du lecteur C de votre émulateur préféré. Pour l'exécuter, il faudra lancer le programme *DEVPAC.PRG* pour l'utiliser.

Remarque : il existe bien sûr d'autres compilateurs 68k que vous pouvez aussi utiliser sur différentes plateformes tel que VASM.

A propos du framework

Nous avons besoin d'avoir un programme qui puisse s'exécuter normalement sous GEM - le bureau de l'Atari - et le Framework que nous allons utiliser permet de répondre à différents critères.
Son mode de fonctionnement est assez simple d'utilisation et est agencé comme ceci :

- Il réserve de la mémoire et l'alloue automatiquement tout en adaptant tout le contenu du programme qui sera copié au chargement et exécuté en mémoire lors de son lancement,
- Nous utilisons le mode SUPERVISEUR pour utiliser certaines fonctions du TOS et accéder à des zones de mémoires spécifiques du ST sans provoquer de plantage machine avec des bombes à l'affichage.
  Nous allons très peu utiliser d'appel système jusqu'à la fin du programme. (Appel de type TRAP #1 soit un appel GEMDOS)
- Nous préparons la stack BSS dans un premier temps, puis nous allons initialiser le Hardware du ST : nous sauvegardons l'état des Timers et des Interruptions du ST tout en les désactivant afin de démarrer notre pro-

gramme « proprement », on sauvegarde la palette d'origine - soit seize couleurs - puis nous initialisons les états des Timers ainsi que les Interruptions au gré de nos besoins. Et enfin nous sauvegardons l'état des adresses vidéo d'origine, on supprime la souris, le son du clavier et on prépare l'ACIA pour gérer les flux entrants / sortants du clavier.

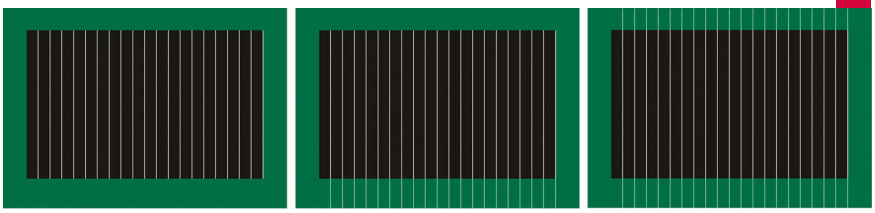
- Côté graphique, nous allons préparer deux écrans « physiques » et « logiques » que nous allons créer dans notre zone BBS en full accès mémoire afin de permettre un clipping parfait à chaque fois que le ou les effets seront exécutés dans la boucle principale *default_loop,* tout en étant synchronisés avec la **VBL**. Nous pouvons appliquer un masque à nos écrans avec un *PATTERN* pour avoir une visualisation complète des écrans utilisés, mais il permet d'abord d'effacer proprement nos écrans. A noter que parfois nous n'avons pas forcément besoin de deux écrans, suivant les accès mémoire et le plan - bitplane - utilisé par le ou les effets. C'est pourquoi nous pouvons choisir le nombre d'écrans à utiliser, soit un ou deux par défaut. Mais il ne faut pas négliger la mémoire utilisée au final car un écran représente une zone mémoire de 160 x 200, soit 32kb.
- Nous initialisons enfin la musique, puis nous lançons l'exécution de la VBL qui va nous permettre de synchroniser tous les évènements de notre programme : jouer de la musique au format Soundchip en boucle et de gérer les différents Timers côté **MFP**.
- Puis nous avons une boucle principale qui permet d'échanger nos deux écrans physiques et logiques tout en incluant des effets que l'on pourra exécuter à l'intérieur de celle-ci. On pourra sortir de la boucle en appuyant la touche ESPACE ou on pourra visualiser le temps machine restant via la touche ALTERNATE de gauche du clavier.
- Et au final nous rendons l'état original au système tout en passant le mode standard du ST, le mode USER.

Description du Framework

Nous allons utiliser des directives d'assembleur comme paramètres pour paramétrer notre environnement de démonstration :

- Les paramètres *BOTTOM_BORDER*, *TOPBOTTOM_BORDER* permettent de passer en mode Overscan bas ou haut plus bas tout en jouant avec les Timers du ST ou en étant en mode standard avec l'option *NO_BORDER*. A la base, l'écran du ST aura une résolution 320 par 200 avec 16 couleurs, soit le mode LOW RESOLUTION à 50 Hertz. Pour paramétrer le type d'écran voulu, il suffit de choisir son mode d'affichage en mettant à "0" celui que l'on désire utiliser et de mettre à "1" les deux autres dont on n'a pas besoin. [1]

1



NO_BORDER BOTTOM_BORDER TOPBOTTOM_BORDER

La zone de couleur verte correspond à la zone non modifiable de l'écran d'un ST.

- Concernant la gestion des écrans, nous utiliserons deux écrans par défaut (physiques et logiques) pour éviter des effets de scintillement via le paramètre *NB_OF_SCREEN*. Il doit être initialisé avec le paramètre *ONE_SCREEN* si on utilise un écran ou *TWO_SCREENS* pour deux écrans. Le paramètre *PATTERN* est notre masque pour remplir nos écrans, on peut mettre des valeurs à partir de $0 jusqu'à $ffffffff. Mais il doit toujours être initialisé car il sert principalement à les effacer.
- Le paramètre *SEEMYVBL* permet de visualiser le temps machine qui est utilisé par le ou les effets.
- Le paramètre *ERROR_SYS* capture le type d'erreur. Au lieu d'afficher un nombre de bombes accompagné d'un reset Hard, on indique via un code couleur où se situe notre erreur. (Bus de données, Bus d'adresse, etc.).

La gestion des erreurs a été écrite par Dbug (Mickaël Pointier - http://www.defence-force.com)

- Le paramètre *FADE_INTRO* permet d'effectuer un Fade de couleurs à partir de la couleur blanche provenant du GEM vers la couleur noire.
- Le paramètre *TEST_STE* détecte si le type d'ordinateur utilisé est un STE.
- Le paramètre *STF_INITS* utilisé par défaut, permet au programme de rester compatible avec les autres gammes Atari tel que le ST jusqu'au FALCON avec carte accélératrice lors de son exécution. Source écrit par Leonard (Arnaud Carré - http://leonard.oxg.free.fr/SainT/saint.html).

Grâce à ce petit programme écrit en assembleur nous allons écrire notre première démonstration qui sera constituée des effets de RASTERS et d'un SCROLLTEXT.

Dépassons les capacités de l'Atari STF

En utilisant le Timer B, nous allons changer la couleur du fond de l'écran, et ce, permettre d'afficher plus de 16 couleurs ! Nous pouvons par défaut utiliser 199 lignes de couleurs sur la partie de l'écran - visible - standard du ST, et pour chaque ligne nous pouvons afficher une couleur qui s'étalonne entre $000 et $777. Pour cela nous allons utiliser les Timers A et B via les interruptions $fffffa07.w et $fffffa13.w. Nous allons sélectionner à partir de quelles lignes on va afficher nos Rasters en passant par l'adresse $fffffa21.w, initialiser notre routine **HBL** en la plaçant sur l'adresse du vecteur $120.w, puis lancer notre Timer via l'interruption $fffffa1b.w. Notre routine exécutée dans la **HBL** permet de temporiser / synchroniser le Timer B et l'adresse vidéo du ST (adresse $ffff8209.w) pour ensuite envoyer notre palette de couleur que l'on veut afficher via l'adresse de la couleur de notre fond d'écran qui est à l'adresse $ffff8240.w. (40 couleurs au total vont être utilisées pour cette palette non standard).

Une fois la synchronisation faite et les variables implémentées, nous utilisons une boucle pour afficher 160 lignes de Rasters en forme d'escalier. Nous envoyons les couleurs de notre palette *PAL_RAST* via le registre d'adresse A0 sur notre couleur de Background sur le registre d'adresse A1 via l'instruction MOVE.W qui transmet des données au format WORD. Nous obtenons un effet improbable qui est en fait « customisé » par un nombre de NOP (instruction n'effectuant rien) bien « placé » tout en jouant avec les Timers !

Voici le code utilisé pour cet effet, ne pas oublier de protéger les registres utilisés dans notre **HBL** afin de ne pas impacter tout la démonstration : [2]

Petit rappel:

- Toutes les instructions en dehors des Overscans sont incluses dans la directives *NO_BORDER* (initialisation et lancement) ;
- La Vertical Blank Line utilise le vecteur d'interruption $70.w, c'est elle qui lance /gère des Timers en attendant le retour de l'écran, on la synchronise à chaque balayage dans la

```
Hbl_ligne:
    clr.b     $fffffa1b.l

    movem.l   a0-a1/d0-d1,-(a7)

    moveq     #$e,d0
    move.w    #9,d1
.loop:
    dbf       d1,.loop              ; Tempo !
.sync:
    move.b    $ffff8209.w,d1        ; We have a video adress available ?
    beq.s     .sync
    sub.b     d1,d0
    rol.w     d0,d0                 ; Set done !
    lea       $ffff8240.w,a1        ; Color of the Backgound
    lea       PAL_RAST,a0           ; Colors of the Rasters
    move.w    #9,d0
.tempo:
    dbf       d0,.tempo             ; Tempo !

    move.w    #80,d0                ; Rasters Loop
.DoRaster:
    dcb.w     40,$3298              ; 40 instructions MOVE.W (A0)+,(A1)
    dcb.w     11,$4e71              ; 11 NOP
    dcb.w     40,$3298              ; 40 instructions MOVE.W (A0)+,(A1)
    dcb.w     8,$4e71               ; 8 NOP
    dbf       d0,.DoRaster

    movem.l   (a7)+,a0-a1/d0-d1

    bclr      #0,$fffffa0f.w        ; Stop Timer A & B
    move.b    #0,$ffff8240.w        ; Black color after Rasters
    rte
```

2

```
Scrolling:
    move.l    physique(pc),a0       ; Physique screen selected for display the effect
    lea       160*100(a0),a0        ; And put the Scrolltext in the middle of the screen
    lea       Buffer_Scroll,a2      ; Buffer of the character font
    moveq     #15,d7                ; 16 lines to display
.scroll:
    roxl      (a2)+                 ; Scroll each pixel of the character Buffer 16 times
i set 160-8
 rept 20
    roxl      i(a0)                 ; Scroll the right to the left part of the Screen
i set i-8
 endr
    lea 160(a0),a0                  ; Next line
    dbf d7,.scroll

    addq      #1,sequence           ; We need to ROXL 16 pixels of the character
    cmpi      #16-1,sequence        ; And we need to count 16 times
    ble       nocar
    clr       sequence
restart:
    move.l    pt_text,a0            ; Next character of the text
    move.b    (a0)+,d0
    tst.b     d0
    bne.s     .noendtxt
    move.l    #text,pt_text         ; Or restart the text
    bra.s     restart
.noendtxt:

    move.l    a0,pt_text            ; Test character in our ASCII table
    lea       list_chr,a2
    moveq     #-$1,d1
.search:
    addq      #$1,d1
    cmp.b     (a2)+,d0
    bne.s     .search
    lea       fonte,a0              ; Find Font character to display
search0:
    cmpi      #20,d1
    bmi.s     .search1
    lea       640(a0),a0
    subi      #20,d1
    bra.s     search0
.search1:
    add       d1,d1
    lea       0(a0,d1.w),a0         ; Find it !

    lea Buffer_Scroll,a2            ; Recopy Font character to the Buffer character
    moveq     #16-1,d0
.recopy:
    move      (a0),(a2)+
    lea       40(a0),a0
    dbf       d0,.recopy
nocar:
    rts
```

3

boucle principale avec la fonction *Wait_Vbl* ;

- La Horizontal Blank Line balaye l'écran ligne par ligne et utilise le vecteur d'interruption $120.w ;
- Ses deux interruptions matérielles sont reconnaissables car elles finissent par un RTE (Return from Exception) ;
- La Multi Fonctional Peripheral - appelé **MFP** - gère les interruptions matérielles que nous avons initialisées juste avant d'installer les deux vecteurs d'interruption.

Le scrolltext

Pour faire un scrolltext nous allons utiliser la résolution standard du ST qui est de 320 pixels par 200 lignes. Chaque pixel est composé d'une valeur entre 0 to 15 (1 pixel est égale à 4 bits), soit 160 bytes affichées par ligne ? Prenons exemple avec la valeur $FFFF0000, notre byte, notre pixel donc équivaut à la couleur numéro 6 de notre palette qui se situe au 3ème plan sur quatre que peut utiliser notre écran. Pour faire bouger un élément sur notre écran physique du ST, même en utilisant le 1er plan qui contient une seule couleur, nous devons modifier en temps réel le ou les pixels, ce qui est très consommateur pour une machine architecturée à base d'un Motorola 68000 à 8Mhz. Pour pallier ce problème, nous devons calculer les décalages requis pour chaque déplacement du bloc de pixels via l'instruction Rotate Left with Extend !

Donc pour afficher un Scrolltext de 16 par 16 pixels, soit une ligne continue de 320 pixels sur 16 lignes, nous n'allons pas utiliser un scrolling standard avec l'instruction MOVE.W mais plutôt l'instruction ROXL.W qui permettra de gérer le déplacement à gauche de la partie graphique tout en utilisant le registre étendu "X" à chaque décalage. De plus nous utiliserons un Buffer pour chaque caractère à afficher, en fait cela nous permettra de bufferiser notre séquence dans une partie de la mémoire et de la copier ensuite sur l'écran ; ceci afin d'éviter de tout le temps solliciter l'adresse vidéo pour chaque instruction effectuée lors du Scrolling. Nous écrirons un sous-programme *Scrolling* que nous appellerons via un BSR dans la boucle principale.

Voici le code du scrolling de 1 pixel par frame : [3]

Avec le code des Rasters dans la VBL et l'appel du Scrolling dans la boucle principale, voici le résultat final haut en couleur : [4]

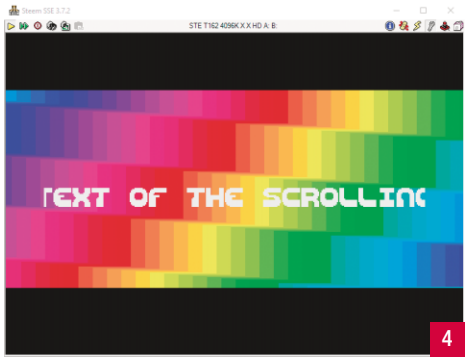
Vous trouverez le code source assembleur complet sur la branche suivante : https://github.com/NoExtra-Team/framework/tree/master/sources/RASTERS.

Ressources

Le Framework est disponible à l'adresse suivante https://github.com/NoExtra-Team/framework et rien ne vous empêche de contribuer à ce petit bout de code pour le faire évoluer ou de créer d'autres exemples. Pour plus d'informations lisez le fichier A_LIRE.TXT. Le programme DevPac ST de l'éditeur HiSoft avec lequel on va compiler nos fichiers écris en assembleur est disponible sur une autre branche https://github.com/NoExtra-Team/framework/tree/master/utils. La documentation de DevPac est disponible au format PDF sur le site Internet http://devlynx.ti-fr.com/ST/, vous pourrez ainsi connaître tous les secrets de la compilation sur ST avec ce logiciel.

Documentations

- Les livres du développeur TOME 1 et 2 pour tout comprendre sur la technologie des Atari modèles STF et STE



4

- Une foule de scans de livres concernant l'Atari ST sont hébergés sur le site http://devlynx.ti-fr.com/ST/
- Plus de codes sources et de tutos ? Visitez le Wiki *Atari-forum* sur http://www.atari-wiki.com/?title=Assembly_language_tutorials
- Tous les utilitaires pour Atari ST sont disponibles sur le site *Essential software List* à l'adresse https://sites.google.com/site/stessential/
- Fan du magazine *ST Magazine*, tous les disques sont archivés et disponibles sur http://stmagazine.free.fr/archives/

Pour finir

Nous verrons dans un prochain article la gestion d'un menu avec le chargement et l'exécution de programme, en fait nous verrons comment sont agencés les Menus EXTRA par le groupe de démo NoExtra-Team. Nous parlerons aussi du Blitter sur STE et comment jouer un Module quatre voix sur un Atari STF et sur STE. Bref, tout plein de bonnes choses pour la prochaine fois....