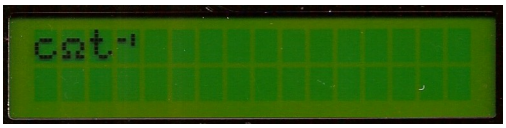
Il est de plus en plus fréquent d'utiliser un afficheur numérique au sein des réalisations électroniques à base de microcontrôleur ou microprocesseur. Néanmoins, on se heurte rapidement à une lacune de taille : les caractères accentués ne sont pas gérés en interne... Une solution existe ; voyons comment l'appliquer.

Le problème

Lors d'un travail de développement, je me suis retrouvé à devoir gérer un de ces petits afficheurs dits « afficheurs intelligents ». Ce nom est d'ailleurs plutôt surestimé car il n'y a aucune intelligence dans ces modules. Les classiques sont gérés par un contrôleur KSS0066 (Samsung), HD44100, HD44780 (Hitachi) ou équivalent. S'il est vrai que ces modules ne sont pas tout jeunes, ils restent largement utilisés, un peu partout où il est nécessaire d'avoir un dialogue machine / homme à peu de frais.

L'utilisation de ces petits modules ne comporte pas de problème particulier ; les documentations des constructeurs étant suffisamment claires, et les informations disponibles sur l'Internet suffisamment nombreuses.

Lors de mon développement, je devais écrire « côté ». Là, stupeur : les codes ASCII de « ô » et de « é » ne sont pas associés, en interne, aux lettres « ô » et « é », mais à la lettre grecque « **Ω** » et au caractère mathématique « $^{-1}$ ».



Après un rapide tour d'horizon de la CGROM (table de caractères interne), force est de constater que ces afficheurs n'ont pas été développés en France...

Il n'est donc pas étonnant que « ô » et « é » ne soient pas présents, car pour mémoire, seule la première partie de la table ASCII (soit sept bits) est normalisée de façon commune, de façon internationale. La deuxième partie (liée au huitième bit) est définie en fonction des différents pays, voire laissée libre à l'utilisateur.

Alors que faire ?

Trois choix se profilaient :

– Ecrire en capitales (ce qui n'est pas très esthétique) ;

– Omettre les accents (ce qui est orthographiquement incorrect) ;

– Ecrire en anglais (ce qui peut poser des problèmes à certains utilisateurs).

Après un rapide sondage auprès de mes collègues, j'ai opté à contrecœur pour la solution sans accents.

Peu après j'ai vu, dans un article d'une revue polonaise consacrée à l'électronique, qu'il était possible de réaliser ses propres caractères... Mais comment ?

La solution choisie était donc remise en question.

Prochaine étape : réaliser mes propres caractères.

La documentation est plutôt vague ; on y apprend que c'est possible, mais après plusieurs essais, je

ne parvenais pas à faire fonctionner l'ensemble correctement. Mes recherches (Internet, ouvrages et documentations) n'ont pas abouti. Tout le monde est d'accord pour dire que c'est faisable, mais personne n'explique vraiment de quelle façon. Il m'a fallu une bonne semaine pour trouver la solution, et un analyseur logique n'a pas été de trop.

Avant de voir la démarche complète ainsi qu'un exemple, apportons quelques précisions, et rappelons la façon dont est organisée la mémoire interne.

Préliminaires et rappels

On se référera ici à un affichage comportant deux lignes de seize caractères, le module le plus classique (quoique le quatre lignes de seize caractères commence à prendre sa part du marché), configuré en fonte 5×7 pixels (ou 5×8 si on inclut le curseur). Nous ne parlerons pas de la fonte 5×10.

De même, on ne verra ici que la gestion en mode 4 bits (le mode 8 bits est aussi disponible).

Le port où est relié l'afficheur (nommé `PortLCD` dans le programme, et défini à l'adresse $ 1062) est câblé de la façon suivante :

E	RS			D3	D2	D1	D0

Notes :

– D0 (LSB) à D3 représentent les données sur 4 bits ;

– RS et E (MSB) seront décrits plus loin ;

– les deux autres bits ne sont pas utilisés ici ;

– la broche R/$\overline{W}$ est physiquement reliée à la masse : c'est une configuration classiquement rencontrée.

D'autre part on rappelle qu'en binaire, LSB signifie *Least Significant Bit*, soit « bit de poids le plus faible » et MSB signifie *Most Significant Bit*, soit « bit de poids le plus fort ».

Généralement, en assembleur, un nombre hexadécimal est précédé d'un « $ », un nombre binaire est précédé d'un « % », et un nombre décimal est précédé d'un « ! ». Un nombre précédé d'aucun symbole est considéré comme décimal.

L'ensemble du programme est écrit en assembleur 68HC11 (Motorola), sous l'environnement Controlboy F1 (http://www.controlord.fr). Bien que le 68HC11 se fasse vieux, il est encore plébiscité par de nombreux électroniciens et largement rencontré. Quoi qu'il en soit, les commentaires permettront à tous d'adapter le code à leur environnement.

Enfin, il est vivement conseillé de se procurer la documentation constructeur du contrôleur et de l'avoir sous la main lors de la lecture de cet article !

Organisation de la mémoire interne

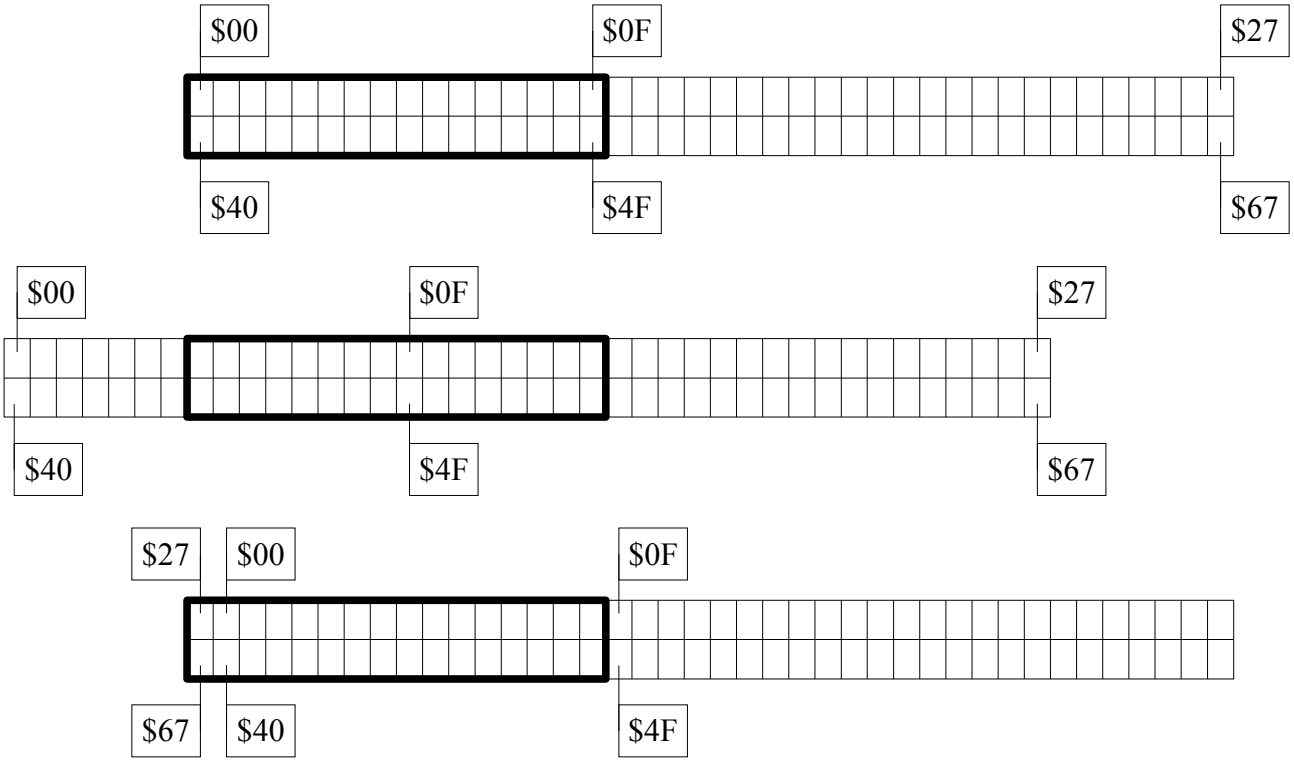
La mémoire interne est scindée en trois parties :

– La DDRAM (*Display Data RAM*) correspond à l'ensemble des caractères potentiellement visibles à un moment donné. Cette mémoire peut contenir 80 caractères (40 par ligne de l'écran). L'écran, lui, est limité à 16 caractères par ligne. Pourquoi la mémoire contient-elle plus de caractères que ne peut en voir l'utilisateur ? Simplement pour pouvoir de façon simple faire défiler du texte. En effet, les adresses en mémoire sont fixes (première ligne de $ 00 à $ 27, et deuxième ligne de $ 40 à $67), et l'écran n'est qu'une sorte de fenêtre sur cette mémoire. Cette pseudo-fenêtre est fixe, et on va faire défiler l'espace-mémoire pour en voir telle ou telle partie.

Sur la figure suivante, on a la pseudo-fenêtre à sa position initiale, ainsi que l'espace-mémoire. Dans le deuxième cas, on a cette même pseudo-fenêtre (qui n'a pas bougé), mais l'espace-mémoire a été

déplacé grâce à l'instruction *Shift left* (décalage à gauche), utilisée sept fois. Autrement dit, le contenu affiché a été décalé à gauche sept fois.

Il est à noter que la mémoire est bouclée sur elle même ; si, depuis la position initiale, on fait un *Shift right* (décalage à droite), les éléments mémorisés en $ 27 et $ 67 se retrouveront à gauche de l'afficheur (c'est comme si les données étaient mémorisées sur un cylindre mobile). Cette spécificité est aussi illustrée sur la figure suivante.



– La CGROM (*Character Generator ROM*) correspond aux caractères intégrés au contrôleur d'affichage. On y trouve la table ASCII classique (ou à peu près) sur sept bits, et une partie étendue (huitième bit) qui comporte divers caractères comme des lettres grecques, des caractères asiatiques, des flèches, etc. Il y a en tout cent soixante ou deux cent huit caractères disponibles en fonte 5×7, qui varient selon le masque de fabrication de la CGROM.

– La CGRAM (*Character Generator RAM*) est celle qui nous intéresse ; on y stocke les caractères personnalisés. La place disponible offre la possibilité de mémoriser huit caractères spéciaux, en fonte 5×7.

L'ensemble de la CGRxM (CGRAM ou CGROM) est adressé sur douze bits (A_0 à A_{11}). Huit bits (A_4 à A_{11}) sont accessibles à l'utilisateur, et permettent de sélectionner un caractère. Les quatre autres bits (A_0 à A_3, les LSB) sont gérés en interne, et représentent chacune des lignes que comporte le caractère. Il est en effet constitué de huit lignes, soient trois bits : sept lignes pour le caractère proprement dit, et une ligne pour le curseur (sous-tiret ou *underscore*). Un bit (A_3) n'est donc pas utilisé en interne ; ce dernier est là pour l'utilisation de la fonte 5×10.

Analyse du stockage d'un caractère

La façon dont est stocké un caractère est la suivante :

Adresses en CGRxM (12 bits)												*Données en CGRxM (8 bits)*							
Utilisateur (8 bits)								**Interne (4 bits)**											
A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
0	1	1	0	0	1	0	1	0	0	1	0	0	0	0	0	1	1	1	0
0	1	1	0	0	1	0	1	0	0	1	1	0	0	0	1	0	0	0	1
0	1	1	0	0	1	0	1	0	1	0	0	0	0	0	1	1	1	1	1
0	1	1	0	0	1	0	1	0	1	0	1	0	0	0	1	0	0	0	0
0	1	1	0	0	1	0	1	0	1	1	0	0	0	0	0	1	1	1	0
0	1	1	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0	0	0

En CGRxM, nous avons nos caractères qui sont définis sur huit bits chacun. Les définitions réelles se font de O_0 à O_4. Un NL1 (niveau logique un) noircit un pixel, tandis qu'un NL0 le laisse transparent. Les données O_5 à O_7 sont au NL0 pour une CGROM (car un industriel peut demander sa propre CGROM), ou peuvent prendre des valeurs quelconques en CGRAM ; elles ne seront de toute façon pas visibles.

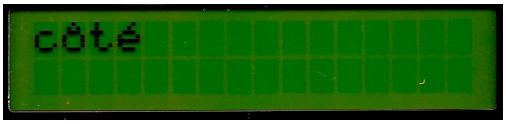
Pour les adresses, il ne faudra spécifier à l'afficheur que les huit bits A_4 à A_{11}, qui sont en fait le code ASCII du caractère voulu. Ici, pour le caractère « e », on enverra donc % 0110 0101. Autrement dit, pour afficher un caractère à l'écran, il suffira de placer l'adresse CGRxM sur huit bits dans la DDRAM ; si la partie de la DDRAM correspondante est comprise dans la pseudo-fenêtre, le caractère s'affichera.

Il est à noter que lorsque l'on travaille en CGRAM (et en CGRAM uniquement), le bit A_7 n'est pas pris en compte ; on peut définir un caractère en % 0000 0000 ou bien en % 0000 1000 ; et le rappeler par le code % 0000 0000 ou % 0000 1000.

De Ą à Ż

Maintenant que l'on voit comment est organisée la mémoire et de quelle façon il va falloir définir le contenu de la CGRAM, passons à une application pratique. Pour cela, nous allons voir de A à Z un petit programme qui ne sert à rien d'autre qu'à la création et l'affichage de caractères spéciaux.

Nous allons pour cela rester en compagnie de notre « côté », car il comporte deux caractères spéciaux distincts, et surtout parce que si l'on ôte les accents, on a un sens tout à fait différent.



La première chose à faire est d'initialiser l'afficheur. Nous ne reviendrons pas sur cette procédure en détail, car elle est très largement décrite dans les documentations ou l'Internet. Seule précision : il faut normalement des temporisations de 15 ms, 4,1 ms et 100 µs. Nous allons dans tous les cas utiliser une seule et unique valeur de 15 ms, pour ne pas surcharger inutilement cet article (et aussi car nous ne sommes pas particulièrement pressés...)

Voici le code :

```
InitLCD   Bsr  Wait15m
          Ldaa      #%00000011     ; Initialisation constructeur
          Bsr       PulseE
          Bsr       Wait15m
          Ldaa      #%00000011     ; Initialisation constructeur
          Bsr       PulseE
          Bsr       Wait15m
```

```
        Ldaa    #%00000011      ; Initialisation constructeur
        Bsr     PulseE
        Ldaa    #%00000010      ; Initialisation constructeur
        Bsr     PulseE
        Ldaa    #%00101000      ; Initialisation du fonctionnement (001)
                                ; - en mode 4 bits (0)
                                ; - deux lignes (1)
                                ; - fonte 5*7 points avec curseur (0)
                                ; Rq : les deux LSB n'ont pas d'importance
        Bsr     SetLCD
        Ldaa    #%00001000      ; Extinction de l'afficheur
        Bsr     SetLCD
        Ldaa    #%00000001      ; Effacement de l'afficheur
        Bsr     SetLCD
        Bsr     Wait15m
        Ldaa    #%00000110      ; Initialisation du mode d'entrée (000001)
                                ; - par incrémentation des adresses (1)
                                ; - sans décalage de l'affichage (0)
        Bsr     SetLCD
        Ldaa    #%00001100      ; Validation et réglage de l'afficheur (00001)
                                ; - afficheur validé (1)
                                ; - curseur éteint (0)
                                ; - sans clignotement du curseur (0)
        Bsr     SetLCD
        Rts                     ; Fin de sous-programme d'initialisation
```

Ah, mais on constate que l'on fait appel à deux ou trois sous-programmes (*via* l'instruction `Bsr` ; *Branch to SubRoutine*)... Que font-ils ? Allez, on les voit un par un...

On a besoin, d'abord, de `Wait15m`, une simple temporisation de 15 ms :

```
Wait15m Ldx     #$1D4B          ; (3 cycles machine) $1D4B = !7499
Label15 Dex                     ; (3 cycles machine)
        Bne     Label15         ; (3 cycles machine)
        Rts                     ; (5 cycles machine)
```

Ensuite, intervient `PulseE`, qui génère une impulsion sur la ligne E de l'afficheur car lors de l'envoi d'une commande à l'afficheur, ce dernier a besoin d'un front descendant pour valider l'ordre. On envoie donc une impulsion de 40 µs, *via* ce sous-programme.

```
PulseE  Oraa    #%10000000      ; E = 1
        Staa    PortLCD
        Bsr     Wait40µ         ; Durée de l'impulsion
        Anda    #%01111111      ; E = 0
        Staa    PortLCD
        Rts                     ; Fin de sous-programme
```

Ce `PulseE`, fait lui aussi appel à un sous-programme : `Wait40µ`, qui génère l'attente de 40 µs. Voici le code de cette temporisation :

```
Wait40µ Ldx     #$13            ; $13 = !19
Label40 Dex
        Bne     Label40
        Rts
```

Enfin, on a des appels à `SetLCD`. Ce sous-programme envoie un mot de réglage de l'afficheur. Ce mot de réglage doit se trouver dans l'accumulateur A avant le lancement de `SetLCD`. Comme on gère ici l'affichage en mode quatre bits, on est contraint d'envoyer le mot de réglage (sur huit bits) en deux fois. D'où la petite gymnastique de rotation et de masquage, qui permet la récupération puis l'envoi de chaque quartet. On voit que ce sous-programme utilise lui aussi `PulseE` pour valider l'envoi desdits quartets.

```
SetLCD  Psha                    ; Récupérer le quartet supérieur de A
        Lsra
        Lsra
        Lsra
        Lsra
        Anda    #%00001111
```

```
        Staa    PortLCD         ; Envoyer le quartet supérieur à l'afficheur
        Bsr     PulseE
        Pula                    ; Récupérer le quartet inférieur de A
        Anda    #%00001111
        Staa    PortLCD         ; Envoyer le quartet inférieur à l'afficheur
        Bsr     PulseE
        Rts                     ; Fin de sous-programme
```

Voilà, on a fait le tour des sous-programmes utilisés. L'initialisation de notre afficheur est à présent terminée. Passons à la chose qui nous intéresse : la génération de nos caractères spéciaux.

Cette génération est faite *via* le sous-programme `GPattern`, qui génère ici deux caractères spéciaux : « ô » et « é ». La capacité de la mémoire CGRAM permet la création de huit caractères, il suffit donc de prolonger ce sous-programme de façon adéquate.

La première chose à faire est de spécifier où l'on veut stocker notre premier caractère. L'espace disponible se trouve en CGRAM, à l'adresse % 00 0000. On va donc envoyer au contrôleur l'ordre d'aller en CGRAM (% 01), immédiatement concaténé avec six des huit bits de l'adresse requise (% 00 0000) ; les deux MSB sont d'office considérés comme étant au NL0. Cet ordre de placement de la CGRAM se fait à l'aide de `SetLCD`, que l'on vient de voir. On utilise `SetLCD` car le fait de spécifier une adresse correspond à un réglage en vue d'un travail futur (transmettre des données).

Ensuite, on envoie nos données les unes après les autres, sans relâche...

On distingue sans peine la forme de nos caractères dans les valeurs envoyées (en rouge pour le « ô », et en vert pour le « é »).

```
GPattrn Ldaa    #%01000000      ; Définition de l'adresse CGRAM
        Bsr     SetLCD
        Ldaa    #%00000100      ; Définition du premier caractère (ô)
        Bsr     SndData
        Ldaa    #%00001010
        Bsr     SndData
        Ldaa    #%00001110
        Bsr     SndData
        Ldaa    #%00010001
        Bsr     SndData
        Ldaa    #%00010001
        Bsr     SndData
        Ldaa    #%00010001
        Bsr     SndData
        Ldaa    #%00001110
        Bsr     SndData
        Ldaa    #%00000000      ; (Curseur)
        Bsr     SndData
        Ldaa    #%00000010      ; Définition du second caractère (é)
        Bsr     SndData
        Ldaa    #%00000100
        Bsr     SndData
        Ldaa    #%00001110
        Bsr     SndData
        Ldaa    #%00010001
        Bsr     SndData
        Ldaa    #%00011111
        Bsr     SndData
        Ldaa    #%00010000
        Bsr     SndData
        Ldaa    #%00001110
        Bsr     SndData
        Ldaa    #%00000000      ; (Curseur)
        Bsr     SndData
        RTS                     ; Fin de sous-programme
```

Pour envoyer une donnée, on utilise un nouveau sous-programme, `SndData`. C'est le même que `SetLCD`, sauf qu'il faut mettre le bit RS au NL1, grâce à une opération de masquage en OU. Ceci car nous n'envoyons plus une adresse, mais une donnée ; nous travaillons avec un autre registre du

contrôleur de l'afficheur. En effet, RS signifie *Register Selector*, soit « sélecteur du registre [de travail] ».

Tout comme pour `SetLCD`, la donnée à envoyer par `SndData` doit se trouver dans l'accumulateur A. Voilà le sous-programme :

```
SndData  Psha                    ; Récupérer le quartet supérieur de A
         Lsra
         Lsra
         Lsra
         Lsra
         Anda      #%00001111
         Oraa      #%01000000    ; Forcer RS au NL1
         Staa      PortLCD       ; Envoyer le quartet supérieur à l'afficheur
         Bsr       PulseE
         Pula                    ; Récupérer le quartet inférieur de A
         Anda      #%00001111
         Oraa      #%01000000    ; Forcer RS au NL1
         Staa      PortLCD       ; Envoyer le quartet inférieur à l'afficheur
         Bsr       PulseE
         Rts                     ; Fin de sous-programme
```

On remarque que dans `GPattern`, on a une seule fois spécifié l'adresse de stockage des caractères. Le contrôleur interne à l'afficheur incrémente lui-même les adresses pour le stockage de la valeur des pixels, ligne à ligne. Du coup, lorsque le premier caractère a entièrement été défini, le contrôleur passe d'office au second caractère.

Si on avait voulu écrire à l'emplacement d'un autre caractère que le premier, il aurait fallu spécifier l'une des adresses suivantes :

Rang du caractère	***Adresse réelle en CGRAM (12 bits)***	***Adresse à spécifier (6 bits)***	***Mot à envoyer avant la définition du caractère***
Premier	% 0000 x000 0000	% 00 x000	`% 0100 x000`
Deuxième	% 0000 x001 0000	% 00 x001	`% 0100 x001`
Troisième	% 0000 x010 0000	% 00 x010	`% 0100 x010`
Quatrième	% 0000 x011 0000	% 00 x011	`% 0100 x011`
Cinquième	% 0000 x100 0000	% 00 x100	`% 0100 x100`
Sixième	% 0000 x101 0000	% 00 x101	`% 0100 x101`
Septième	% 0000 x110 0000	% 00 x110	`% 0100 x110`
Huitième	% 0000 x111 0000	% 00 x111	`% 0100 x111`

Notes :

– en x, il est possible de mettre un NL0 ou un NL1 ; le résultat est strictement identique : ce bit est ignoré en interne ;

– l'adresse n'est spécifiée que sur 6 bits car lors de cette définition, les deux MSB sont considérés comme des NL0, et les quatre LSB aussi (gestion interne ; voir plus haut).

Ca va toujours ?

Bon, on continue.

La phase de création est terminée. On va maintenant envoyer du texte à notre afficheur : le mot « côté », en spécifiant les lettres les unes après les autres. La première chose à faire est, ici aussi, de spécifier à quelle adresse de l'écran (DDRAM) on veut afficher tout ça. Rien de plus simple :

```
         Ldaa      #%10000000
         Bsr       SetLCD
```

Nous sommes en train d'ajuster quelque chose (une adresse), il est donc normal d'utiliser la routine `SetLCD`, et non `SndData`. Le mot envoyé (% 1000 0000), par contre, correspond à quoi ? Au premier caractère de la première ligne. Le MSB (ici au NL1) spécifie que l'on va ajuster une adresse

DDRAM. Les 7 bits qui suivent représentent l'adresse proprement dite ; ici, % 000 0000. Pour rappel, si rien n'a été spécifié de contraire, on a après une initialisation :

– en % 000 0000 ($00) le premier caractère de la première ligne ;
– en % 000 1111 ($0F) le seizième caractère de la première ligne ;
– en % 100 0000 ($40) le premier caractère de la seconde ligne ;
– en % 100 1111 ($4F) le seizième caractère de la seconde ligne.

Une fois cette adresse ajustée, on envoie nos caractères, un à un, à l'aide de la routine `SndData` (ici, c'est bien un envoi de donnée, et non un réglage de l'afficheur). Les adresses de la DDRAM sont incrémentées au fur et à mesure, donc plus besoin de les ajuster. Voilà le code :

```
        Ldaa    #'c'            ; Envoi de "c"
        Bsr     SndData
        Ldaa    #%00000000      ; Envoi du premier caractère créé ("ô")
        Bsr     SndData
        Ldaa    #'t'            ; Envoi de "t"
        Bsr     SndData
        Ldaa    #%00000001      ; Envoi du second caractère créé ("é")
        Bsr     SndData
```

L'envoi d'une lettre classique se fait de façon simple : en l'écrivant entre simples côtes ou en indiquant son code ASCII. L'envoi de l'un de nos propres caractères, lui, se fait en spécifiant son adresse en CGRAM. Le premier (« ô ») a été défini en % 0000 0000, et le second (« é ») en % 0000 0001.

Les caractères classiques peuvent eux aussi être envoyés en binaire : `Ldaa #%01100011` donnera strictement le même résultat que `Ldaa #'c'`. Mais... C'est quand même moins pratique, non ?

A tout cela, il est évident qu'il convient d'ajouter quelques classiques commandes, qui dépendront du compilateur utilisé. Dans notre cas, il faudra avant tout les trois lignes suivantes :

```
PortLCD Equ     $1062           ; Adresse du port LCD
        Org     $8000           ; Origine du programme
        Cli                     ; Activation du debugger
```

De même, il faudra à la fin un `Swi` qui permettra au microcontrôleur de ne pas se retrouver « dans les choux », une fois la tâche exécutée.

```
        Swi                     ; Fin de programme par interruption logicielle
```

L'ensemble du code source est disponible en téléchargement libre sur le site spécifié en pied de page.

Conclusion

Toutes ces démarches ne sont, bien entendu, pas optimisées ; ce type de programmation linéaire permet de bien voir comment séquencer les ordres envoyés. Maintenant, à vous de charger en mémoire vos caractères avant compilation, et de programmer des boucles pour les définir ; à vous de créer plusieurs lots de huit caractères spéciaux, qui pourront être générés à tout moment, et du coup, vous pourrez palier à cette limite de huit. Car avec un peu d'imagination, vous pouvez afficher des *bargraphs*, un indicateur de batterie proportionnel à son niveau réel (mise en œuvre du CAN du processeur), des flèches dans tous les sens, les symboles de lecture, d'enregistrement, de pause pour votre gestion de MP3, etc.

Bonne conception, et faites-moi part de vos réalisations !

maquaire manolo