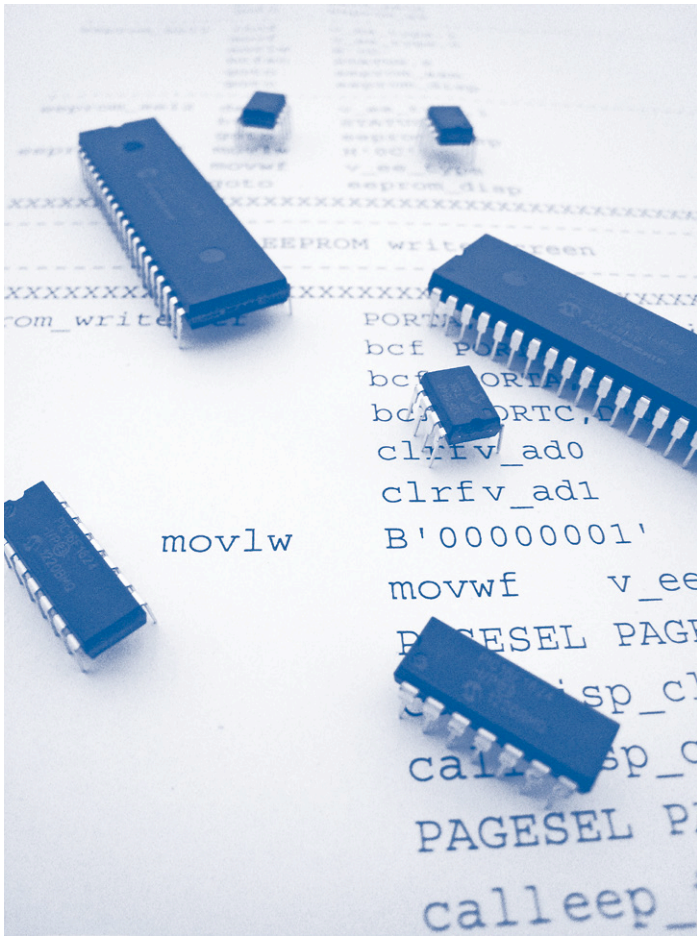

cours intensif d'assembleur (1)

le premier programme

Miroslav Cina (Allemagne), miroslav.cina@t-online.de

De nos jours doit-on encore parler « assembleur » ? Notre réponse est un *oui* (franc et massif), car pour les microcontrôleurs à 8 bits en particulier, optimiser la longueur du code et la rapidité d'exécution peut s'avérer utile. Notre cours intensif autour d'un petit µC PIC allie comme d'habitude théorie et pratique.

L'assembleur est un langage de programmation très proche du matériel, ce qui présente des avantages et des inconvénients.
Avec un langage de programmation de haut niveau, c'est du compilateur que dépend l'efficacité de la compilation du code source. En assembleur, pour obtenir un résultat optimal, il faut s'occuper soi-même tant de la taille du code que de la vitesse d'exécution. La conversion de l'assembleur en langage machine (ou code hexadécimal) est en effet parfaitement définie.
La longueur du programme est un aspect très important en cas d'utilisation d'un *petit* µC (un PIC10F200 par ex. n'a que 256 mots de mémoire de programme). La performance peut jouer un rôle important dans les calculs complexes là où un compilateur C est capable parfois de produire des inepties. Dans de tels cas, il est alors logique de programmer directement en assembleur les parties de programme critiques. Autre avantage : pour les opérations à la chronologie critique, on peut calculer exactement le nombre de cycles d'horloge (c.-à-d. la durée) nécessaire au déroulement d'une routine en assembleur. L'assembleur est aussi le meilleur moyen de découvrir le matériel dans le détail. Les langages de haut niveau cachent (plus ou moins) aux yeux du programmeur ce qui se passe dans les registres & Co.
Cependant, en assembleur, tout est vraiment à faire soi-même – la simple multiplication de deux petits nombres peut requérir une certaine réflexion.

Matériel et logiciel

Comme je travaille depuis longtemps avec les µC PIC de Microchip, j'ai opté ici pour l'un des plus petits d'entre eux, le PIC12F675 [1]. Ce µC d'un euro ou moins fait parfaitement l'affaire pour de nombreuses applications simples.
Pour les expériences proposées, j'ai utilisé un programmateur PICkit2 et une petite carte d'expérimentation (l'ensemble est souvent vendu sous le nom de *PICkit2 Starter Kit*). Dans la prochaine partie du cours, nous verrons comment réaliser soi-même une carte d'expérimentation simple. Rien n'interdit bien entendu d'utiliser pour ce cours le programmateur PICkit3 plus récent.
Le logiciel utilisé est mis à disposition gratuitement. Sous Windows, nous pouvons utiliser un éditeur de texte (Notepad) pour saisir le code source, pour l'assembler ensuite, c.-à-d. le traduire en langage machine, à l'aide du programme MPASM.exe (inclus dans la Suite MPASM de Microchip). La Suite MPASM fait partie de l'IDE (***I**ntegrated **D**evelopment **E**nvironment*) MPLAB. On pourra télécharger l'IDE MPLAB gratuitement chez Microchip [2].
Finalement, le logiciel du PICkit2 assure le « transport » vers le µC du fichier hexadécimal résultant de l'assemblage. Voilà, c'est tout.

Les registres

Commençons par découvrir notre µC et apprendre quelques instructions d'assembleur. Pas de panique – le premier paquet de théorie n'est pas très compliqué ; ensuite nous pourrons démarrer dans la foulée.
Le PIC12F675 est un µC à 8 bits de la famille *midrange* (milieu de gamme) de Microchip. Ce µC est proposé entre autres en boîtier PDIP-8 (***P**lastic **D**ual*

***I**nline **P**ackage*). Six des huit pattes sont des broches de port numérique d'usage général (GP0 à GP5). GP3 ne peut être utilisée qu'en entrée, les cinq autres broches peuvent l'être soit en entrée, soit en sortie. Le PIC dispose de 1 024 mots de mémoire de programme flash, ainsi que de 64 octets de SRAM utilisables librement et de 128 octets d'EEPROM.

Lorsque l'on programme en assembleur, il faut travailler avec des registres ; ce sont, ici, des unités de mémoire de 1 octet qui ont une fonction logique spécifique. Certains registres font partie du noyau ; ils peuvent, par ex. lors de calculs, stocker des valeurs de 8 bits ou donner des informations quant à l'état du µC. D'autres registres sont responsables des blocs périphériques du µC, le registre GPIO qui donne l'état des entrées/sorties par exemple.

Pour les PIC, il y a toujours au moins quatre registres dans le noyau :

- Le registre W
- Le registre de configuration (__CONFIG)
- Le registre des options (OPTION_REG)
- Le registre d'état (STATUS)

La mémoire SRAM de 64 octets utilisable librement, mais aussi la plupart des registres, sont accessibles à des adresses de mémoire bien définies. Une adresse sur 8 bits permet d'accéder à 256 emplacements de mémoire. Un coup d'œil à la cartographie de la mémoire (*memory map*) (**fig. 1**) nous apprend que l'on n'a pas réellement 256 emplacements de mémoire à disposition ; le registre d'état par ex. est accessible aux adresses 03h et 83h.
La cartographie montre que le registre GPIO (nous l'utiliserons bientôt) se trouve à l'emplacement 05h ; nous voyons aussi que nous pouvons accéder à la mémoire d'application (64 octets) sous les adresses de mémoire comprises entre 20h et 5Fh. Les registres sans emplacement de mémoire, comme le registre des options et le registre W, sont des exceptions.
Le registre W (W = *Work*, travail) est le registre principal du µC. Toutes les opérations arithmétiques par ex. se font par le biais du registre W.
Le registre d'état est subdivisé en bits individuels (cf. **fig. 2**). L'important, à cet endroit, est le bit n° 5 (RP0), le *Register Bank Select Bit*. Il est utilisé pour la sélection entre les deux banques de mémoire ; le bit doit être mis à 1 (levé) ou à 0 (effacé) avant que l'on ne puisse donner une adresse comme paramètre à une instruction subséquente. Nous avons accès aux emplacements de mémoire 80h à FFh lorsque le bit est levé ; si RP0 = 0, nous avons accès aux emplacements de mémoire 00h à 7Fh.
L'écriture du registre Config se fait lors du flashage du µC (cf. ci-dessous). Il définit par ex. l'utilisation (ou non) de l'oscillateur interne du µC.

Pour le moment, laissons de côté le registre des options. Maintenant nous allons nous intéresser brièvement à trois autres registres qui, à la différence de ceux qui ont précédé, s'occupent des périphériques.

TRISIO, GPIO et ANSEL

Les registres TRISIO, GPIO et ANSEL servent à commander les E/S. Pour le registre ANSEL, pour le moment il faut juste retenir que nous devons le mettre à 00h pour les premières expériences – ce qui désactive les fonctions analogiques.
Le registre TRISIO commande le sens de la communication. Les bits correspondants du registre TRISIO définissent l'utilisation d'une broche de port soit en entrée, soit en sortie. La valeur *0* en fait une sortie, un *1* une entrée. Si par ex. le premier bit est à zéro (on écrit `TRISIO<0> = 0`), la broche GP0 est utilisée en sortie.

La seule exception est, nous le disions plus haut, la broche de port GP3 (qui remplit également la fonction MCLR - *Master Clear*) ; elle ne peut être utilisée qu'en entrée. Le bit correspondant du registre TRISIO est donc toujours à 1.
Le registre GPIO nous permet de commander directement les entrées/sorties. Toute écriture dans le registre GPIO affecte directement les broches configurées en sortie, les entrées pouvant quant à elles être interrogées par une lecture du registre GPIO.

Premières instructions

Il est temps maintenant de découvrir quelques instructions d'assembleur (les *instructions de base* en quelque sorte). Le PIC12F675 fait partie des µC RISC (*Reduced Instruction Set* = à jeu d'instructions réduit) – il n'y a que 35 instructions à apprendre.

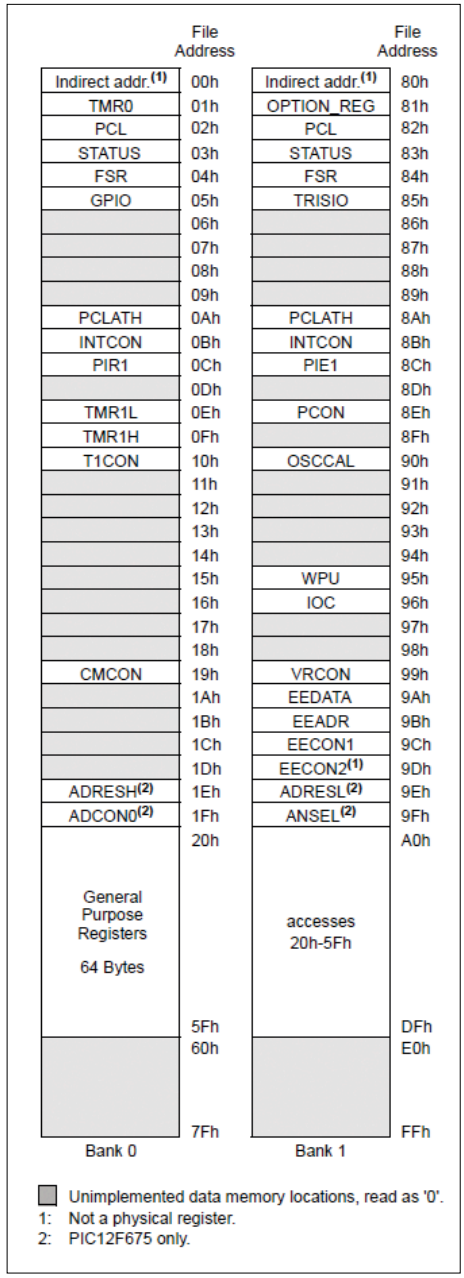

Figure 1. La cartographie de la mémoire donne les adresses des registres et des emplacements de mémoire utilisables librement.

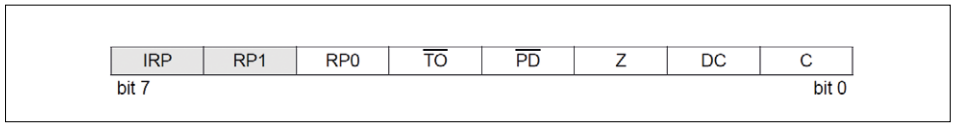

Figure 2. Les bits du registre d'état (*status*).

MOVLW et MOVWF

L'instruction `MOVLW` charge une valeur à 8 bits dans le registre W. La valeur elle-même peut être représentée en binaire, hexadécimal ou décimal.
La syntaxe de la commande est la suivante :

```
movlw   k
```

où *k* est une valeur se trouvant dans l'intervalle compris entre 00h et FFh (1 octet).

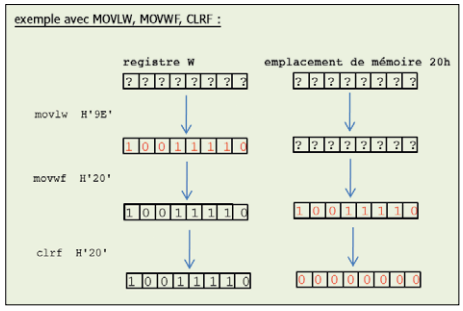

Figure 3. Mode opératoire des instructions *MOVLW*, *MOVWF* et *CLRF*.

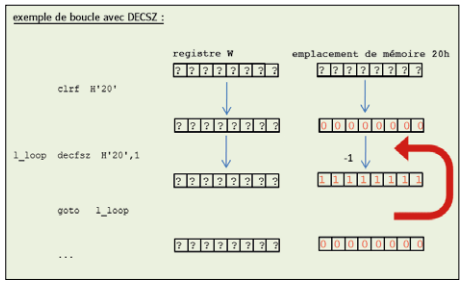

Figure 4. *DECFSZ* permet de réaliser des boucles.

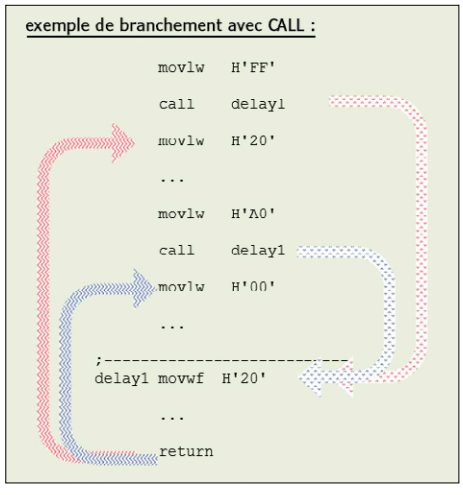

Figure 5. *CALL* sert à appeler un sous-programme.

D'ailleurs, la casse (écriture en majuscules ou minuscules) des instructions en assembleur est sans importance, nous pouvons donc également écrire :

```
MOVLW   k
```

Les exemples ci-après montrent comment écrire la valeur sous forme binaire (préfixe B), hexadécimale (H) et décimale (D) :

```
movlw   B'10011110'
movlw   H'9E'
movlw   D'158'
```

Les trois instructions sont identiques, elles chargent la valeur 9Eh dans le registre W.

L'instruction `MOVWF` copie le contenu du registre W vers un emplacement de mémoire :

```
movwf   f
```

où *f* est une valeur entre 00h et 7Fh qui représente l'adresse-cible (comme nous le disions plus haut, il faudra auparavant, selon le cas, lever ou effacer le 5e bit du registre d'état, pour commuter d'une banque de mémoire à l'autre).

Si nous voulons par ex. écrire la valeur 9Eh à l'adresse 20h, la séquence des instructions pourrait être :

```
movlw   H'9E'
movwf   H'20'
```

La première instruction charge la valeur 9Eh dans le registre W et la seconde écrit le contenu du registre W à l'emplacement de mémoire 20h.

CLRF

L'instruction met le contenu d'un emplacement de mémoire à 00h (CLRF = *Clear f*).
La syntaxe de l'instruction est :

```
clrf  f
```

où *f* est une valeur entre 00h et 7Fh qui représente l'adresse-cible.

Un exemple :

```
clrf  H'20'
```

se traduit par l'écriture de la valeur 00h à l'emplacement de mémoire 20h.

Pour résumer, l'exemple suivant reprend les instructions `MOVLW`, `MOVWF` et `CLRF` :

```
movlw   H'9E'
movwf   H'20'
clrf    H'20'
```

La **figure 3** montre l'effet sur les emplacements de mémoire (au départ, nous ne connaissons pas les valeurs qu'avaient le registre W et l'emplacement de mémoire 20h).

BSF et BCF

Contrairement aux instructions précédentes qui travaillent toujours avec des octets, les instructions `BSF` et `BCF` opèrent sur des bits. **`BSF`** est l'abréviation de *Bit Set f* et, par analogie, **`BCF`** l'abréviation de *Bit Clear f* où *f* représente l'adresse d'un emplacement mémoire. La syntaxe est la suivante :

```
bsf   f, d
bcf   f, d
```

où *f* est une valeur entre 00h et 7Fh et *d* un chiffre entre 0 et 7.

Il s'agit d'instructions qui mettent à un (`bsf`) ou effacent (`bcf`) un seul bit d'un octet se trouvant à un emplacement de mémoire indiqué. Un exemple :

```
bsf   H'03', H'05'
```

Cette instruction met à *1* le bit 5 (d = 05h) de l'emplacement de mémoire 03h (f = 03h).

GOTO

Tout comme avec bien d'autres langages de programmation, `GOTO` permet d'interrompre le déroulement linéaire du programme pour le faire se poursuivre à un autre endroit – spécifié dans l'instruction.
La syntaxe de l'instruction est :

```
goto  k
```

où *k* est une adresse de la mémoire de programme. En fait, *k* peut prendre une valeur entre 000h et 7FFh, mais comme notre µC ne possède que 1 024 mots de mémoire de programme, les valeurs au-delà de 3FFh sont hors des limites.
On pourra, au lieu de *k*, également spécifier une étiquette (*label*), c'est-à-dire un

nom attribué à un emplacement particulier de la mémoire de programme. Cette étiquette doit être définie ailleurs dans le code assembleur ; il suffit de l'intercaler juste avant l'instruction à partir de laquelle doit se poursuivre le programme. Ici, il faut tenir compte de la casse !

DECFSZ

Cette instruction est une abréviation de *DECrement F and Skip if Zero* (décrémenter F et sauter si zéro) ; nous pourrions également la nommer « Instruction de boucle » (*loop instruction*). Elle est chargée de l'exécution conditionnelle de l'instruction qui suit. On commence par décrémenter (diminuer de 1) la valeur de l'emplacement de mémoire *f* pour ensuite exécuter soit l'instruction suivante (si le résultat est différent de zéro), soit sauter ladite instruction et exécuter celle qui suit immédiatement après (si le résultat est égal à zéro).
La syntaxe est la suivante :

```
decfsz f,d
```

où *f* est l'adresse d'un emplacement mémoire et *d* définit où doit être enregistré le résultat de l'opération (décrémentation). Si *d* = 0, le résultat est stocké dans le registre W et l'emplacement de mémoire n'est pas modifié ; si *d* = 1, le résultat est écrit à l'emplacement de mémoire (le registre W reste inchangé).
La **figure 4** montre comment cette instruction permet de réaliser une boucle. Dans la première ligne, nous mettons à 00h le contenu de l'emplacement de mémoire 20h. Immédiatement après on a exécution de l'instruction `DECFSZ` suivie d'un 1 – le résultat écrase le contenu de l'emplacement 20h. Lors du premier passage, nous soustrayons un 1 de 00h ; le résultat est FFh, vu que nous calculons avec des octets non signés. Ensuite, le µC regarde si le résultat est égal à 00h. Ce n'est pas le cas ici, on a donc exécution de l'instruction suivante. Notons que la décision de savoir si le résultat est égal à zéro ou non est prise sur l'examen du bit dit zéro du registre d'état. Ce bit serait à 1 si le résultat de la soustraction avait été 00h.

Comme nous le montre la figure 4, la deuxième ligne est identifiée par une étiquette *l_loop*. À la ligne suivante, `goto l_loop` permet de revenir à l'instruction `decfsz`. Ici on a à nouveau une soustraction d'un 1 et réécriture en 20h. La valeur actuelle diminue ainsi et passe à FEh, et ainsi de suite.
Nous avons donc créé une boucle exécutée 256 fois dans notre exemple. Si la valeur en 20h atteint finalement 00h, on a saut de l'instruction `goto` ; nous sortons de la boucle. À noter que nous n'avons pas utilisé le registre W ici.
On peut ainsi utiliser une boucle lorsque, par ex., on souhaite intégrer une temporisation dans le déroulement du programme (pour faire clignoter une LED par ex.).

CALL / RETURN

Tout comme nous l'avons vu avec d'autres langages de programmation, `call` sert à appeler un sous-programme et `return` à terminer le sous-programme et reprendre le programme principal.
La syntaxe de l'instruction est très proche de celle de `goto` :

```
call k
```

où *k* est ici aussi une adresse de la mémoire de programme ; elle peut aller de 000h à 3FFh.
Ici à nouveau il va de soi qu'il est possible d'attribuer une étiquette. Nous préférons donner un nom explicite à notre sous-programme, *Delay* par exemple.

`return` ne comporte pas de paramètre additionnel :

```
return
```

L'instruction `call` permet d'appeler le même sous-programme depuis différents points dans le programme (cf. **fig. 5**). Une fois le traitement terminé et le `return` exécuté, le déroulement du programme continue toujours avec l'instruction qui suit le `call` en question.

NOP

Même ça, ça existe : cette instruction ne fait vraiment rien ; on pourra l'utiliser pour une temporisation. Nous nous en servirons dans notre exemple pratique.

Définitions de constantes

En assembleur, on peut utiliser le mot-clé `EQU` pour définir des constantes afin d'améliorer la lisibilité du code et sa portabilité. Si au début du code nous convenons par ex. que :

```
STATUS   EQU  H'0003'
```

Nous n'avons plus besoin, plus loin dans le code, d'indiquer explicitement l'emplacement auquel peut être trouvé le registre d'état et écrire par exemple :

```
bsf   STATUS,H'05'
```

au lieu de

```
bsf   H'0003',H'05'
```

pour mettre à un le 5e bit du registre d'état. Ne faisons pas les choses à moitié et convenons aussi que

```
RP0   EQU  H'05'
```

Nous pouvons dès lors, par

```
bsf   STATUS,RP0
```

et

```
bcf   STATUS,RP0
```

mettre à un et effacer le 5e bit du registre d'état et ainsi commuter entre les banques de mémoire.
Le code se laisse également plus facilement porter d'un type de µC à un autre (passage par ex. d'un petit PIC à un PIC plus grand). Le registre d'état peut, avec un autre type de µC, se trouver à un autre emplacement de mémoire ; il nous suffit dans ce cas de changer la définition des constantes, le reste du code est conservé.

__CONFIG

Cette commande (appelée directive) n'est pas, à l'inverse des instructions mentionnées précédemment, convertie en code machine exécuté lors du déroulement du programme. Elle est utilisée pour fixer le contenu du registre Config. Le registre de configuration est l'un des (très rares) registres, dans lequel on ne peut pas écrire à une adresse de la mémoire. La définition de la valeur du registre de configuration se fait dès le flashage du µC.

En général, nous retrouvons la directive `__CONFIG` au début du code source, par exemple :

```
__CONFIG        B'00000110000100'
```

Ces bits permettent de paramétrer l'utilisation de l'oscillateur interne sans quartz externe, la non-activation de la protec-

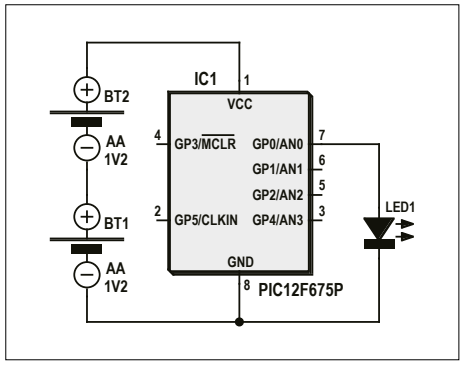

Figure 6. Difficile de faire plus simple pour obtenir le clignotement d'une LED.

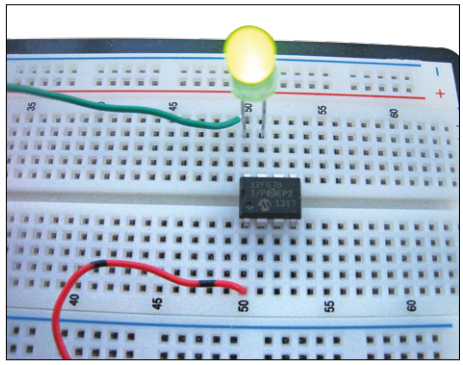

Figure 7. Le circuit peut être réalisé sur une plaque d'essais.

tion des données ou de la mémoire de programme, la désactivation de la fonction de chien de garde (*watchdog*), etc. Nous retrouvons, comme d'habitude, la description de chacun de ces bits dans la feuille de caractéristiques, souvent au chapitre « *Special Features of the CPU / Configuration Bits* » (les fiches en français sont rarissimes).

Commentaire

Un commentaire commence toujours par un point-virgule (;). Tout ce qui suit le point-virgule (jusqu'à la fin de ligne - ***End of Line***) est ignoré par le compilateur et considéré comme commentaire.
Si la ligne commence par un point-virgule, toute la ligne est traitée comme un commentaire.

Application « Hello World »

Nous savons maintenant tout ce qu'il faut pour créer une application simple. Allons-y.
Notre tâche est (relativement) facile. Nous allons, pour commencer, connecter une LED à la broche GP0 et la faire clignoter.
En termes de matériel, la solution est d'une mise en œuvre ultrasimple. Nous utiliserons l'oscillateur d'horloge interne (4 MHz) et n'aurons pas besoin de quartz externe. La LED sera connectée directement à GP0 (même sans résistance) – le µC limite lui-même le courant de sortie à environ 20 mA (alimentation de 5 V) par le biais de la résistance interne du port (**fig. 6**).
Le circuit peut être réalisé sur une plaque d'essais (**fig. 7**). La source d'alimentation pourra par ex. être constituée de deux piles de 1,2 V ; mais aussi n'importe quelle autre source qui peut fournir au moins 20 mA sous 2 à 5 V.

Code source

À quoi doit ressembler le programme en assembleur ? Que faut-il faire au minimum pour écrire un programme exécutable ?
Comme nous le disions plus haut, nous pouvons utiliser pour notre exemple un simple éditeur de texte tel que Notepad, y saisir le code et enregistrer le fichier. L'extension doit toujours être *.asm* (il faudra peut-être, après enregistrement, la modifier manuellement). Nous utiliserons comme nom de fichier : *01_LED_v_1p03.asm*. Vous pouvez télécharger un fichier tout fait sur la page de projet de cet article [3].

La **figure 8** reprend le début du code. Pour augmenter la lisibilité du programme et le rendre plus facile à comprendre par d'autres utilisateurs, nous l'avons doté de nombreux commentaires. Nous voulons de plus travailler avec des constantes. Des définitions telles que

```
STATUS EQU  H'03'
```

rendront le code plus portable si nous voulons plus tard nous essayer à un PIC un peu plus grand.
Heureusement il est inutile de réécrire à chaque fois ces définitions de constantes dans notre code. On retrouve, dans le fichier *P12F675.INC*, qui fait partie de la Suite MPASM, toutes les définitions essentielles (adresses de registres, positions de bits, etc.) pour notre type de µC.
Nous incluons ce fichier tout au début

```
;*****************************************************************************
;*                        Blinking LED                                       *
;*                      v 1.03 - 09.03.2015                                  *
;*****************************************************************************
;*         Hardw.: PIC12F675 used                                            *
;*         OSC.....: Internal osc. used     POWER.....:  5V                  *
;*****************************************************************************

;-------------------------------------------------------
;Pins connection:
;----------------------
;GP0 --> LED output
;GP1 --> N/C
;GP2 --> N/C
;GP3 --> N/C
;GP4 --> N/C
;GP5 --> N/C
;-------------------------------------------------------

 INCLUDE "P12F675.INC"

_U002                                   EQU     B'00000110000100'               ;MCLR = input
              __CONFIG        _U002

;=============================================================================
;       Variable Definition
;=============================================================================
TIMER1                  EQU             H'20'                   ;Used in delay routine
TIMER2                  EQU             H'21'                   ; "             "               "

;=============================================================================
```

Figure 8. Les commentaires sont importants, même en assembleur.

```
;-------------------------------------------------------
;
;       Register Definitions
;
;-------------------------------------------------------
;...
;-----Bank0-----------------
INDF            EQU  H'0000'
TMR0            EQU  H'0001'
PCL             EQU  H'0002'
STATUS          EQU  H'0003'
FSR             EQU  H'0004'
GPIO            EQU  H'0005'
PCLATH          EQU  H'000A'
INTCON          EQU  H'000B'
PIR1            EQU  H'000C'
TMR1            EQU  H'000E'
TMR1L           EQU  H'000E'
TMR1H           EQU  H'000F'
T1CON           EQU  H'0010'
CMCON           EQU  H'0019'
ADRESH          EQU  H'001E'
ADCON0          EQU  H'001F'
;-----Bank1-----------------
OPTION_REG      EQU  H'0081'
TRISIO          EQU  H'0085'
PIB1            EQU  H'008C'
PCON            EQU  H'008E'
OSCCAL          EQU  H'0090'
WPU             EQU  H'0095'
;...
;...
```

Figure 9. Définitions des registres dans le fichier *Include*.

Liens

[1] www.microchip.com/wwwproducts/Devices.aspx?product=PIC12F675

[2] www.microchip.com/pagehandler/en-us/family/mplabx

[3] www.elektor-magazine.fr/130483

de notre code par une directive INCLUDE. Cela fonctionne exactement comme avec tous les autres langages de programmation « classiques » ; le résultat est le même que si nous avions tout simplement copié, par un copier-coller, le code du fichier à inclure dans notre programme. La **figure 9** nous montre le contenu du fichier comportant les définitions de constantes typiques au µC.

La figure 8 nous apprend que nous avons en outre défini deux variables de type octet (c.-à-d. des emplacements de mémoire en SRAM), à savoir *TIMER1* et *TIMER2*. Nous les utiliserons dans notre boucle de temporisation.

La **figure 10** montre le reste du programme. Nous n'y utilisons que des instructions dont il a été question plus haut. L'illustration montre les différents registres avec tous leurs bits. Un point d'interrogation noir à la place du bit signifie que sa valeur est sans importance ici. Dans la boucle *Main* on voit un petit rond rouge ; il signale le moment d'allumage de la LED. Le rond gris identifie l'instruction qui éteint la LED. Les flèches représentent les sauts.

Le petit sous-programme *delay_r* introduit une temporisation. Vous voyez peut-être immédiatement comment cela fonctionne ? Note : on se trouve en présence de deux boucles imbriquées.
Tout à la fin du fichier on doit impérativement trouver la directive END.

Assembler

Une fois le programme saisi, nous pouvons assembler le code. Pour cela, nous appelons le programme MPASM.EXE (partie de l'IDE MPLAB).
Après le démarrage du programme (**fig. 11**), nous sélectionnons notre fichier texte (dans le champ *Source File Name*). Dans le champ *Processor*, il faut sélectionner le type de notre µC. Le reste peut être laissé avec les valeurs par défaut. Presser sur le bouton *Assemble* déclenche la compilation du fichier ASM.
Ce processus produit alors un fichier hex (et quelques autres fichiers que nous pouvons ignorer pour l'instant).

Le PICkit2 est accompagné du programme PICkit2V2.exe. Avant le lancement du programme, le kit doit être connecté à l'ordinateur via USB et le µC doit se trouver dans le support de programmation de la carte d'expérimentation. Lors du démarrage de PICkit2V2.exe, il y a détection automatique du µC (cf. **fig. 12**).
Ensuite, le point du menu *File – Import Hex* permet de charger le fichier hex et enfin le bouton *Write* de le flasher dans le µC.

C'est tout pour aujourd'hui – j'espère que vous avez apprécié vos premiers pas en assembleur. Dans le prochain article nous passerons en revue l'ensemble du jeu d'instructions du PIC12F675. Notre exercice pratique se matérialisera sous la forme d'un dé électronique.

(130483 – version française : Guy Raedersdorf)

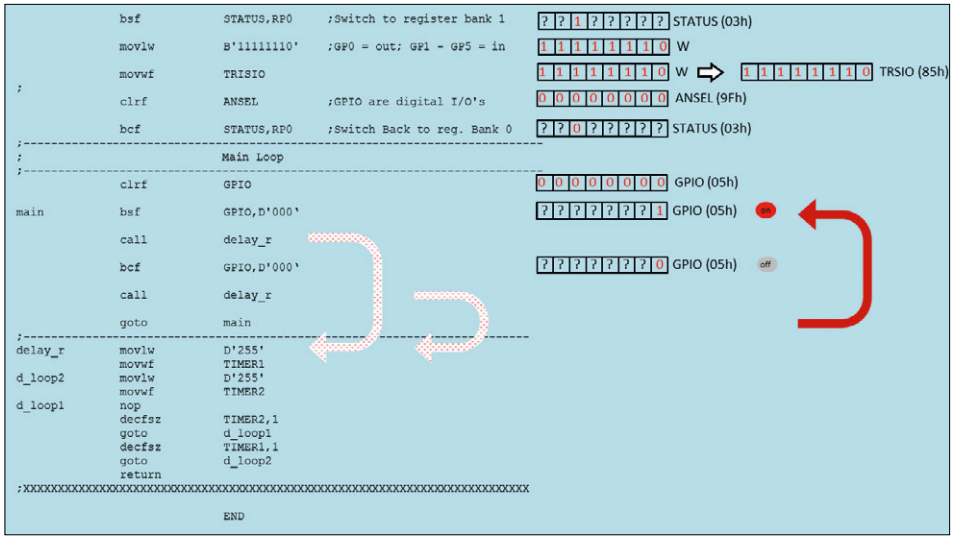

Figure 10. Pour la temporisation, nous utilisons un sous-programme.

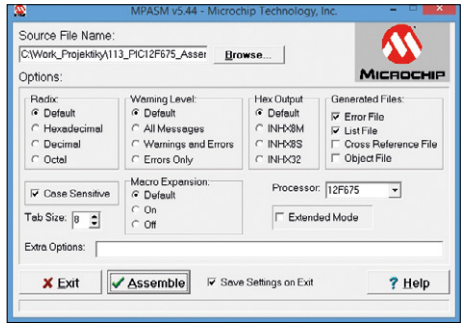

Figure 11. Capture d'écran de l'assembleur MPASM qui convertit le code en fichier Hex.

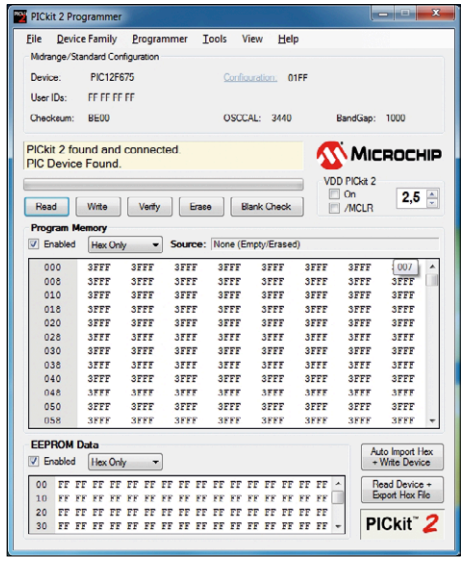

Figure 12. Le logiciel de programmation du PICkit nous permet de flasher le programme dans le µC.