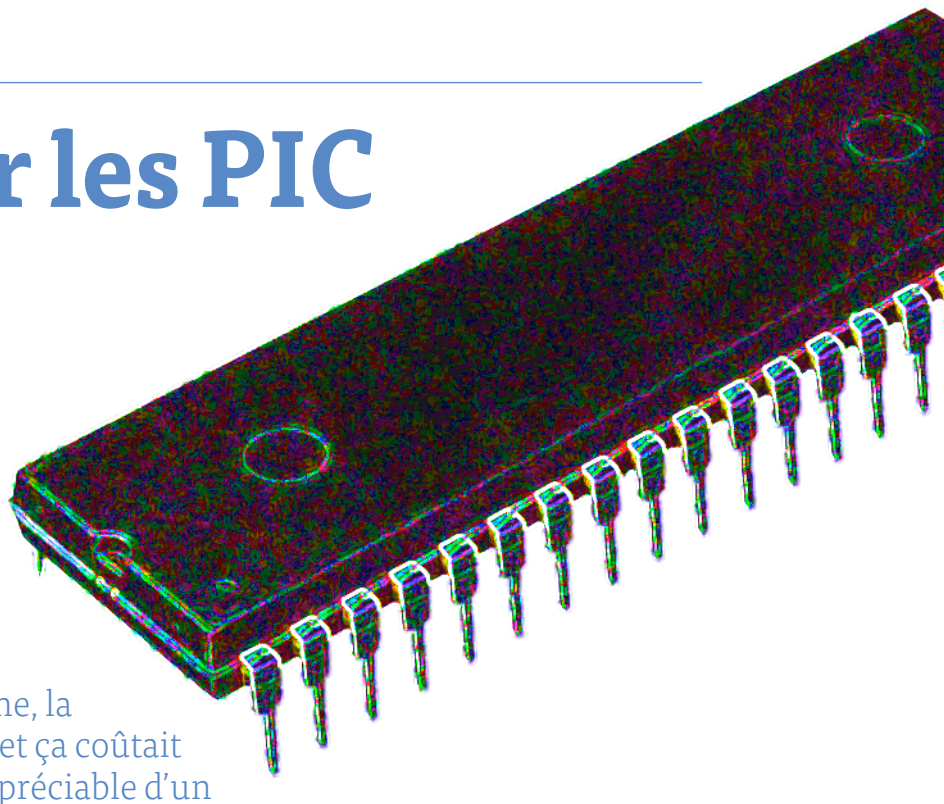
programmer les PIC à petits pas

Produire des sinusoïdes en assembleur



Tam Hanna (Slovénie)

À l'époque où les 8 bits étaient la norme, la capacité des mémoires était modeste et ça coûtait cher. Pour obtenir quelque chose d'appréciable d'un µP à moins de 5 MHz, il fallait un code costaud sans une once de gras. On n'avait donc pas d'autre choix que d'écrire en assembleur qu'il fallait transformer en un code hexécutable. Je propose ici d'écrire une routine en assembleur pour un PIC d'aujourd'hui, et produire une sinusoïde avec son CN/A intégré.

Pour le logiciel, nous utiliserons MPLAB X l'environnement intégré (IDE) de *Microchip,* et un PIC 16F18877. La programmation proprement dite passe par l'interface ICSP. Vous gardez donc la liberté de choisir la carte de développement ou de prototypage qui vous plaît.
Pour produire une onde sinusoïdale avec un CN/A, nous devons lui fournir à intervalles réguliers des valeurs numériques correspondant aux niveaux de tension successifs d'une sinusoïde. Par période, nous utiliserons ici 32 valeurs discrètes. La fonction `sin()` du tableur Excel permet de créer le tableau (**fig. 1**) qui donne dans la colonne de gauche les intervalles de temps de 0 à 31. C'est la formule `=A2*((2*PI())/32)` qui nous fournit les valeurs pour une période de 0 à 2π radians. La troisième colonne contient les valeurs sinusoïdales des radians de la colonne 2 `=SIN(B2)`. Ces valeurs sont comprises entre –1 et +1, or le CN/A n'accepte que des valeurs positives. Nous décalerons donc toutes les valeurs `=1+C2` de la plage de 1 à +1 dans la plage positive de 0 à 2. Ces valeurs ajustées sont ensuite mises à l'échelle à l'aide de `D3*(255/2)` pour les situer dans la plage de 0 à 255 qui correspond aux valeurs d'entrée codées sur 8 bits attendues par le CN/A. Enfin, les valeurs sont arrondies : `ARRONDI(E10)`.

Tableaux en stock

Les valeurs d'une fonction sinusoïdale sont des constantes qui peuvent être stockées en mémoire sous forme de table(au). Pour récupérer une valeur dans la table, nous pouvons utiliser l'instruction (mnémonique) `RETLW`. Celle-ci retourne avec la valeur de la table chargée dans W (l'accumulateur).
En assembleur, nous pouvons assigner une zone de mémoire à cette table. Dans les langages de niveau supérieur, on n'a pas à se soucier de tels détails, le compilateur se débrouille. Il est possible de commencer cette zone de stockage presque n'importe où, mais nous verrons plus tard qu'il y a des avantages à préférer certaines adresses. Ici, l'adresse de départ du tableau sera `0x200` :

```
TABLE_VECT  CODE  0x0200
  dt  127, 152, 176, 198, 217, 233, 245, 252, 255,
   252, 245, 233, 217, 198, 176, 152, 127, 102, 78,
   56, 37, 21, 9, 2, 0, 2, 9, 21, 37, 56, 78, 102
END
```

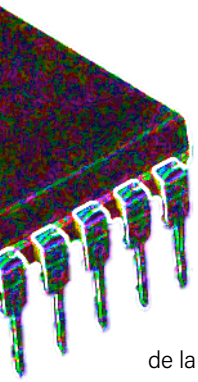
La directive `dt` (*define table*) stocke les valeurs fournies sous forme de tableau. Ce fragment de code peut déjà être passé en assembleur. L'assembleur MPLAB ignorera bien des situations critiques, mais lorsqu'il lance un avertissement tel que :

```
Warning[202] C:\USERS\TAMHA\MPLABXPROJECTS\CH6-
   DEMO1.X\NEWPIC_8B_SIMPLE.ASM 72 : Argument out of
   range.  Least significant bits used.
```

il faut le prendre au sérieux et résoudre le problème ; les erreurs sont une partie essentielle du processus d'apprentissage. C'est la façon de définir le tableau qui est incorrecte ici.

Détour : assemblage et désassemblage

Il existe un lien direct bien défini entre les mnémoniques (le nom des instructions) et le code machine sous-jacent. Le code machine est créé *par assemblage* des mnémoniques. Le processus inverse, du code machine aux mnémoniques, est le désassemblage.
Double-cliquez sur l'onglet *Usage Symbols disabled* **(fig. 2)** en bas à droite. Une fenêtre de réglage s'ouvre. Cochez *Load Symbols when programming or building for production (*c'est-à-dire *Charger les*



symboles lors de la programmation ou de la construction pour la production) afin d'activer les outils d'analyse durant la compilation.

Après avoir cliqué sur *Apply, relancez* le compilateur afin d'afficher l'utilisation de la mémoire. L'option *Window > Debugging > Output > Disassembly Listing File Project indique* au compilateur d'afficher une version désassemblée de la section de code originale. Cela peut être utile car cela nous montre comment le compilateur traite le code **(fig. 3)**.

La sortie se compose de six colonnes, avec à gauche l'adresse *logique* du mot dans la mémoire de programme du PIC. La colonne suivante est l'équivalent décimal de la commande. La troisième colonne contient la version désassemblée obtenue à partir du code machine hexadécimal (sans toutefois les noms de constantes ou de variables et les commentaires du fichier assembleur original). La quatrième colonne indique le numéro de ligne dans le fichier .asm et, après les deux-points, se trouve la ligne responsable du mot binaire à gauche.

Nous voyons maintenant que, par défaut, l'assembleur suppose que les valeurs stockées dans le tableau sont hexadécimales, et n'a donc utilisé que les deux derniers chiffres de chaque nombre (codée sur 8 bits, une valeur hexadécimale est constituée de deux caractères, son équivalent décimal en compte trois) :

```
0200  3427  RETLW 0x27     73:     dt  127, 152, 176,
   198, 217, 233, 245, 252, 255, 252, . . .
0201  3452  RETLW 0x52
0202  3476  RETLW 0x76
0203  3498  RETLW 0x98
0204  3417  RETLW 0x17
0205  3433  RETLW 0x33
. . .
```

Ces tableaux admettent toutes sortes des valeurs (hexadécimales, binaires, octales, décimales, etc.). Il faut donc, avant chaque valeur, en préciser sa «base». Ainsi, un point décimal indiquera à l'assembleur que la valeur est décimale. Pendant l'assemblage, le compilateur prendra la valeur décimale maximale (8 bits) de 255 et la convertira en valeur hexadécimale maximale FF pour l'utiliser dans le code. Maintenant le tableau se présente comme ceci :

```
TABLE_VECT  CODE  0x0200 dt  .127, .152, .176, .198,
   .217, .233, .245, .252, .255, .252, .245, .233,
   .217, .198, .176, .152, .127, .102, .78, .56, .37,
   .21, .9, .2, .0, .2, .9, .21, .37, .56, .78, .102
END
```

Accès aux tableaux

Pour récupérer les informations stockées dans la table, nous y accédons avec l'instruction `RETLW`. Au retour, le registre W (l'accumulateur) contient la valeur de la table. Pour comprendre, convertissons en binaire l'adresse de départ `0x0200` : `0000 0000 0000 0000 0010 0000 0000 0000 0000 0000`.

Les appels vers des emplacements dans la mémoire du programme peuvent être effectués avec l'instruction `CALLW`, (**fig. 4**).

La mémoire du PIC compte 32.768 mots, pour laquelle il faut donc un pointeur d'adresse de 15 bits. Les instructions comme `CALLW` utilisent le registre W pour transmettre la valeur du compteur ordinal, lequel pointe vers l'emplacement de la mémoire où commence le sous-programme. Comme le format de W n'est que de 8 bits, on utilisera le

Schritt	Laufwert	Sinuswert	Bereinigt	DAC-Wert	Abgerundet
0	0,0000	0,0000	1,0000	127,5000	127
1	0,1963	0,1951	1,1951	152,3740	152
2	0,3927	0,3827	1,3827	176,2921	176
3	0,5890	0,5556	1,5556	198,3352	198
4	0,7854	0,7071	1,7071	217,6561	217
5	0,9817	0,8315	1,8315	233,5124	233
6	1,1781	0,9239	1,9239	245,2946	245
7	1,3744	0,9808	1,9808	252,5501	252
8	1,5708	1,0000	2,0000	255,0000	255
9	1,7671	0,9808	1,9808	252,5501	252
10	1,9635	0,9239	1,9239	245,2946	245
11	2,1598	0,8315	1,8315	233,5124	233
12	2,3562	0,7071	1,7071	217,6561	217
13	2,5525	0,5556	1,5556	198,3352	198
14	2,7489	0,3827	1,3827	176,2921	176
15	2,9452	0,1951	1,1951	152,3740	152
16	3,1416	0,0000	1,0000	127,5000	127
17	3,3379	-0,1951	0,8049	102,6260	102
18	3,5343	-0,3827	0,6173	78,7079	78
19	3,7306	-0,5556	0,4444	56,6648	56
20	3,9270	-0,7071	0,2929	37,3439	37
21	4,1233	-0,8315	0,1685	21,4876	21
22	4,3197	-0,9239	0,0761	9,7054	9
23	4,5160	-0,9808	0,0192	2,4499	2
24	4,7124	-1,0000	0,0000	0,0000	0
25	4,9087	-0,9808	0,0192	2,4499	2
26	5,1051	-0,9239	0,0761	9,7054	9
27	5,3014	-0,8315	0,1685	21,4876	21
28	5,4978	-0,7071	0,2929	37,3439	37
29	5,6941	-0,5556	0,4444	56,6648	56
30	5,8905	-0,3827	0,6173	78,7079	78
31	6,0868	-0,1951	0,8049	102,6260	102

Figure 1. Table des données prête à l'emploi.

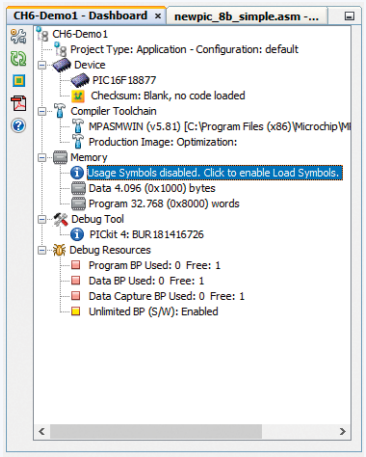


Figure 2. Double-cliquez sur cette ligne pour ouvrir la fenêtre d'options.

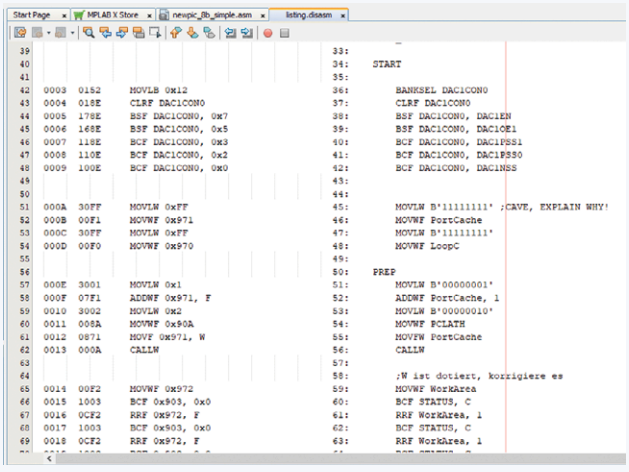


Figure 3. Le code machine brut.

CALLW	Subroutine Call With W
Syntax:	[*label*] CALLW
Operands:	None
Operation:	(PC) +1 → TOS, (W) → PC<7:0>, (PCLATH<6:0>) → PC<14:8>
Status Affected:	None
Description:	Subroutine call with W. First, the return address (PC + 1) is pushed onto the return stack. Then, the contents of W is loaded into PC<7:0>, and the contents of PCLATH into PC<14:8>. `CALLW` is a 2-cycle instruction.

Figure 4. La commande mnémonique CALLW calcule l'adresse par un processus relativement complexe.

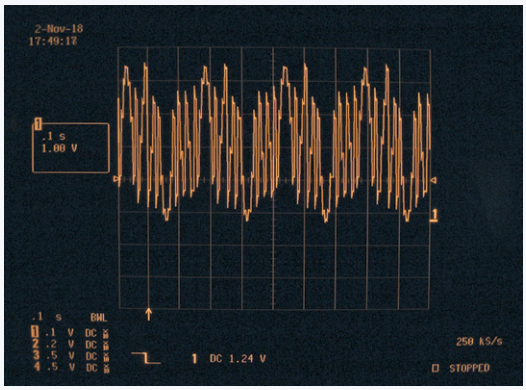


Figure 5. Voilà qui ne ressemble pas du tout à une sinusoïde.

registre PCLATH pour y stocker avant l'appel la partie supérieure du compteur ordinal. Pour réduire la charge et les éventuels conflits de chronologie, le PIC n'utilisera la valeur de PCLATH que si elle est nécessaire. Nous pouvons assembler les valeurs des pointeurs dans les registres avant d'effectuer l'appel. Les écritures sur PCLATH n'affectent pas la valeur actuelle du pointeur ordinal. L'initialisation du programme commence par l'incrémentation du compteur :

```
WORK
    BANKSEL DAC1CON1
    MOVLW B'00000001'
    ADDWF PortCache, 1
```

Dans l'adresse binaire complète ci-dessus, nous pouvons voir quelle valeur doit être chargée dans la partie supérieure du compteur ordinal. Cette valeur est transférée à PCLATH via le registre W :

```
    MOVLW B'00000010'
    MOVWF PCLATH
```

Nous sommes prêts. En exécutant `CALLW`, la valeur de la table est renvoyée dans le registre W. Cette valeur est ensuite transmise au CN/A pour produire le niveau de tension analogique correspondant :

```
    MOVFW PortCache
    CALLW
    MOVWF DAC1CON1
    CALL WAIT
```

Si nous exécutons le programme, ça fonctionne jusqu'à ce que MPLAB interrompe l'exécution à un moment donné parce que le tableau se compose de 32 valeurs alors que le programme en attend 255. Ces emplacements non prévus de la mémoire peuvent contenir des valeurs erratiques provenant de code installé précédemment. À mesure que le pointeur s'incrémente, le programme se retrouve en territoire inconnu. La section suivante du code limite la portée en soustrayant 31, en détectant si le drapeau zéro est levé par l'opération, puis en utilisant `CLRF` avec `PortCache` :

```
    MOVFW PortCache
    SUBLW .31
    BTFSC STATUS, Z
    CLRF PortCache
```

Enfin, nous revenons au début de la boucle :

```
    GOTO WORK        ; loop forever
```

Nous pouvons maintenant exécuter le programme. Le résultat de la **fig. 5** semble chaotique et ne ressemble en rien au sinus attendu. Le CN/A du processeur n'accepte en effet que des valeurs comprises entre 0 et 31, alors que nous lui fournissons des valeurs entre 0 et 255.

Tableaux en mémoire

Nous pourrions recourir à Excel pour réduire les valeurs requises pour la sinusoïde et les utiliser dans le tableau. Nous pourrions aussi utiliser le processeur pour effectuer la division nécessaire des valeurs dans le logiciel. Si nous utilisons l'assembleur, nous avons un accès direct aux valeurs binaires dans les registres. La méthode la plus simple pour effectuer la division par 2 d'une valeur binaire consiste à décaler son motif de uns et de zéros dans le registre d'une position vers la droite. Cette opération ne tient pas compte des bits de retenue et met à zéro le bit de poids le plus fort dans le registre. Pour convertir des valeurs comprises entre 0 et 255 en valeurs comprises entre 0 et 31, nous devons diviser chaque valeur par huit. Cela revient à la décaler trois fois vers la droite.

Lorsque les valeurs sont copiées dans la mémoire, nous devons calculer les adresses cibles. Pour ce faire, nous examinons les registres de base. PCLATH est déjà connu, mais nous nous intéressons également à INDF et FSR. Notre PIC possède deux registres de sélection de fichiers (FSR) de 16 bits. Ceux-ci sont capables d'accéder à tous les registres de fichiers et à la mémoire du programme, ce qui permet d'avoir un seul pointeur de données pour toute la mémoire. Les registres indirects de fichiers (INDF) ne sont pas des registres physiques. Une instruction qui accède à un registre INDFn accède au registre à l'adresse spécifiée par le registre de sélection de fichier (FSR). Les opérations d'écriture dans la mémoire de programme ne peuvent pas être effectuées via les registres INDF.

Une particularité du PIC est le fait que le choix entre mémoire de programme et mémoire de travail se fait par le bit 7 du registre d'adresse supérieur FSRxH. S'il est à 1, l'adresse est celle d'un emplacement dans la mémoire de programme ; sinon c'est dans la mémoire de données. On recommence en stockant les variables. Cette fois, nous avons besoin de 32 octets de mémoire en tout – c'est trop pour une utilisation comme mémoire partagée.

Nous accédons à la mémoire dans une banque que sélectionne l'assembleur et utilisons l'adressage relatif. Comme aucun paramètre supplémentaire n'est spécifié, MPASM laisse donc le libre choix dans

la position du `DataBuffer` :

```
udata_shr
   LoopC res 1
   PortCache res 1
udata
   DataBuffer res 32
```

Les parties haute et basse de l'adresse restent incertaines. Heureusement, le *linker* nous facilite le travail avec deux opérateurs :

```
START
  MOVLW high DataBuffer
  MOVLW low DataBuffer
```

Sachant cela, copions les informations de la mémoire du programme dans la mémoire principale. Les opérations de décalage ne fonctionnent pas directement dans W, il faut donc une variable supplémentaire :

```
udata_shr
   LoopC res 1
   PortCache res 1
   WorkArea res 1
```

Lorsqu'on travaille avec des tableaux de données, la condition initiale est cruciale. Nous chargeons 1111.1111 dans PortCache car la boucle s'incrémente *avant* utilisation. Avec la première incrémentation, elle passera donc à 0. Si nous avions chargé 0, la première incrémentation nous ferait écrire à l'emplacement 1 :

```
START
. . .
   MOVLW B'11111111'
   MOVWF PortCache
   MOVLW B'11111111'
   MOVWF LoopC
```

Il y a deux boucles : la boucle PREP prépare et écrit le tableau des valeurs des sinusoïdes dans un tableau ; la boucle `Work` envoie les valeurs au CN/A afin que la forme d'onde du signal puisse être sortie. PREP commence par un accès au tableau ::

```
PREP
   MOVLW B'00000001'
   ADDWF PortCache, 1
   MOVLW B'00000010'
   MOVWF PCLATH
   MOVFW PortCache
   CALLW
```

Maintenant, avec la valeur dans W, nous devons la déplacer vers le registre F et exécuter trois instructions de rotation à droite pour diviser sa valeur par 8 :

```
   MOVWF WorkArea
   BCF STATUS, Z
   RRF WorkArea, 1
   BCF STATUS, Z
   RRF WorkArea, 1
```

Pour éviter les erreurs causées par le bit de retenue dans le registre d'état, `BCF STATUS, Z` est utilisé pour effacer le drapeau de retenue dans le registre avant chaque opération de décalage. Le registre contient deux petites erreurs intéressantes à corriger.
Nous devons maintenant nous assurer que l'INDF indique le bon emplacement de mémoire : Pour ce faire, les registres FSR0H et FSR0L sont chargés avec des données d'adresse. Le registre H (haut) contient la partie supérieure et le registre L (bas) la partie inférieure :

```
   MOVLW high DataBuffer
   MOVWF FSR0H
   MOVLW low DataBuffer
   MOVWF FSR0L
```

INDF0 indique maintenant le début de la zone de mémoire. Nous devons ajouter le décalage qui identifie chaque emplacement. Il n'est pas certain que le début de la zone se trouve au début d'une page. S'il y avait une retenue lors de l'ajout du décalage dans la partie L, la partie H ne le remarquerait pas. Comme solution à ce problème, le statut du bit C est vérifié et la valeur de FSR0H est incrémentée s'il y a une retenue :

```
   MOVFW PortCache
   CLRC
   ADDWF FSR0L
   BTFSC STATUS, C
   INCF FSR0H
```

La commande `CLRC` (*Clear Carry*) est une macro qui efface le bit C (retenue) dans le registre d'état. Cela permet de s'assurer que INDF0 est correctement configuré. Nous devons charger et écrire la valeur stockée temporairement dans la zone de travail :

```
   MOVFW WorkArea
   MOVLW INDF0
```

Enfin, nous devons nous assurer que la boucle continue de tourner :

```
   MOVFW PortCache
   SUBLW .31
   BTFSS STATUS, Z
   GOTO PREP
   CLRF PortCache
```

Comme ça se complique, il serait utile d'utiliser certains des outils de l'EDI pour examiner de plus près l'exécution du programme et en vérifier le bon fonctionnement.

Débogage en assembleur

Théoriquement, il est facile de placer des points d'arrêt ; cliquez sur les numéros de ligne dans l'IDE pour placer un point d'arrêt rouge. Comme le µC ne reconnait qu'un seul point d'arrêt matériel, il y aura un message d'erreur.
MPLAB utilise des ressources de débogage pour pouvoir fonctionner par étapes. Nous n'avons pas besoin dans le logiciel d'émulation de point d'arrêt pour le moment, nous pouvons donc nier le message en cliquant sur *No*. Le PIC supporte lui les points d'arrêt logiciels, vous

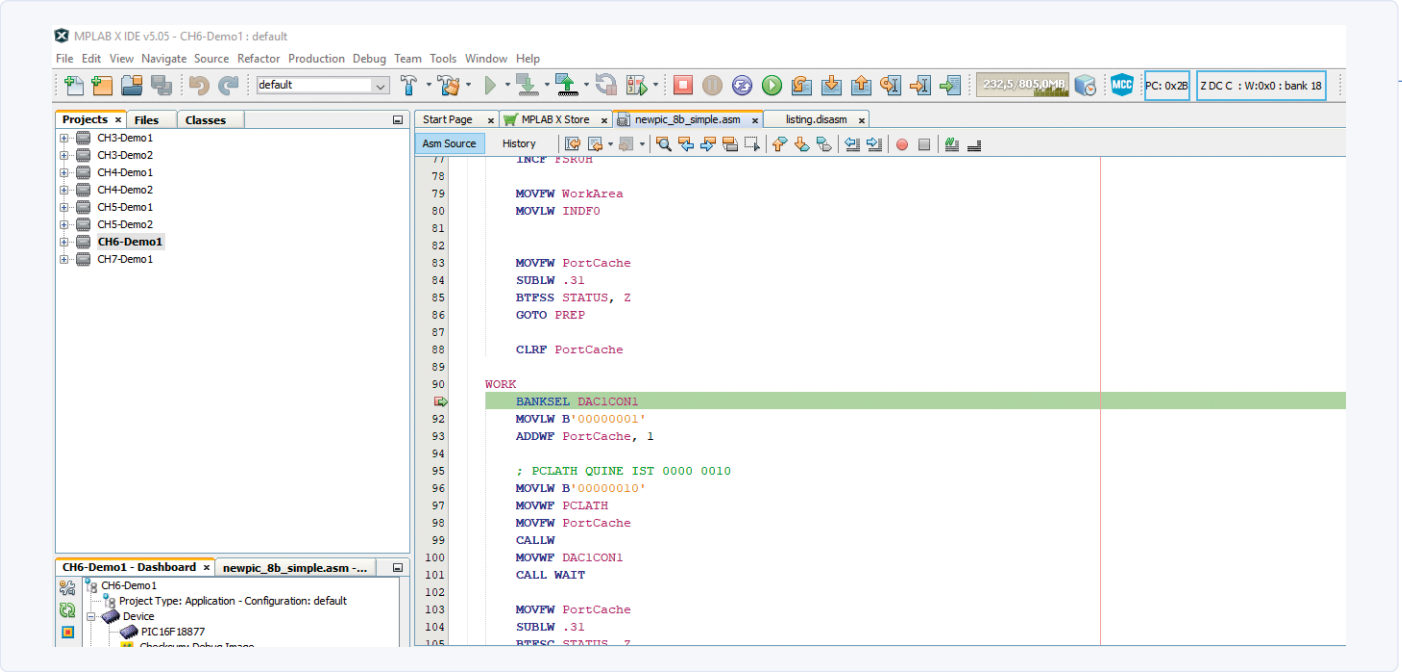


Bild 6. Le débogueur a interrompu l'exécution du programme.

pouvez confirmer avec *Yes.* Dès que vous placerez plus d'un point d'arrêt dans le fichier assembleur, *Microchip* activera cette fonction automatiquement.

Puisque notre PIC ne supporte qu'un seul point d'arrêt matériel, cliquez sur la flèche vers le bas à côté du symbole du débogueur dans la barre d'outils et sélectionnez l'option *Debug main project.* MPLAB ouvre une fenêtre de désassemblage que nous pouvons ensuite fermer. Après avoir atteint le point d'arrêt, l'EDI affiche l'état (**fig. 6**).

La ligne avec le fond vert et la flèche vers la droite dans la colonne des numéros de ligne est l'instruction en cours. Les symboles de la barre d'outils, tels que *stop* et *saut*, permettent à l'utilisateur d'interagir avec le programme. *Window Target Memory Views File Registers* ouvre une fenêtre pour visualiser l'espace mémoire du PIC. Placez le curseur comme le pointeur de la souris sur la déclaration du *DataBuffer*. Cela ouvre une info-bulle avec l'adresse du premier octet et sa valeur. Sur mon poste de travail , cette position est `0xD20`.

Il est plus pratique de cliquer sur la flèche bleue (*GoTo*) dans la fenêtre *File Registers*. Dans la case *GoTo What*, sélectionnez *Symbol* et *DataBuffer*.

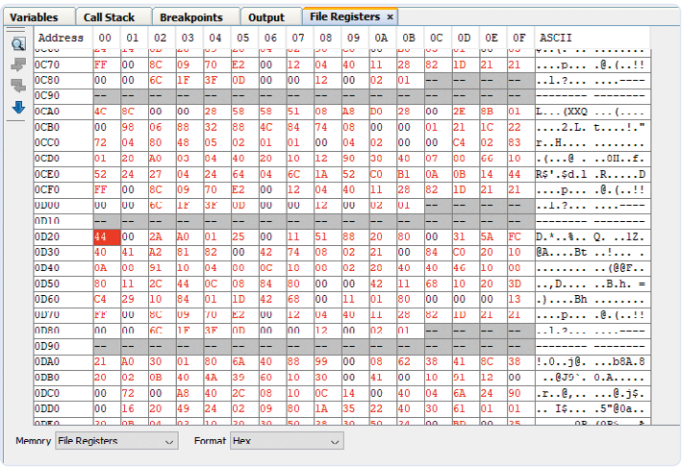


Figure 7. Ces valeurs ont une drôle d'allure.

Fermez le popup après avoir cliqué sur le bouton GoTo pour voir le résultat (**fig. 7**). La case surlignée en rouge représente le premier octet du tableau. Il est évident qu'il y a une erreur car d'autres valeurs figurant dans le tableau, comme FC, sont en dehors de la plage autorisée.

Pour enquêter, nous pouvons remplir la zone de mémoire avec un modèle de bits facilement reconnaissable. Un bon exemple serait d'écrire la valeur 11111111 dans le registre indirect :

```
MOVFW PortCache
CLRZ
ADDWF FSR0L
BTFSC STATUS, C
INCF FSR0H
MOVFW B'11111111'
MOVLW INDF0
```

En visualisant le code dans le débogueur, vous verrez une séquence des mêmes valeurs. Dans la plupart des cas, elles ne doivent pas avoir la valeur FF. Le code comporte une petite erreur. Nous pouvons utiliser l'instruction `MOVLW` pour charger la valeur d'un emplacement mémoire dans le registre W :

```
MOVFW B'11111111'
MOVLW INDF0
```

Du point de vue de MPLAB, le littéral INDF0 est un nombre : après compilation, les valeurs comme PORTA ne sont que des nombres. Notre programme copie donc l'adresse du registre dans les 32 emplacements de mémoire. Néanmoins, nous avons fait un pas de plus puisque nous avons déjà vérifié les données d'adresse. Une version corrigée du programme ressemble maintenant à ceci :

```
MOVLW B'11111111'
MOVWF INDF0
```

Le calcul de l'adresse mémoire fonctionne. Nous pouvons éradiquer l'erreur de calcul réelle. Le premier problème était d'utiliser l'instruc-

tion `MOVLW` au lieu de l'instruction `MOVWF`, ce qui rendait impossible l'écriture dans l'INDF :

```
MOVFW WorkArea
MOVWF INDF0
MOVFW PortCache
SUBLW .31
BTFSS STATUS, Z
GOTO PREP
```

L'examen des résultats révèle que les valeurs ne sont pas correctes. La cause de ce problème est que l'instruction `RRF` peut positionner le bit de retenue. Notre code n'a cependant pris en charge que le drapeau Z. Corrigeons :

```
MOVWF WorkArea
BCF STATUS, C
RRF WorkArea, 1
BCF STATUS, C
RRF WorkArea, 1
BCF STATUS, C
RRF WorkArea, 1
```

Le programme est prêt à fonctionner et le tableau des valeurs des sinusoïdes apparaît dans la fenêtre du débogueur. Pour terminer, nous devons nous assurer dans la boucle de travail que les valeurs sont extraites de la mémoire de données. Cela nécessite un incrément de la variable d'exécution afin d'obtenir un index continu :

```
WORK
   BANKSEL DAC1CON1
   MOVLW B'00000001'
   ADDWF PortCache, 1
```

L'adressage indirect convient pour la lecture et l'écriture. Nous chargeons les deux parties de l'adresse du tampon dans FSR0H et FSR0L. Ensuite, nous ajoutons le décalage et vérifions s'il y a un dépassement de capacité. Le cas échéant, nous incrémentons le registre supérieur :

```
MOVLW high DataBuffer
MOVWF FSR0H
MOVLW low DataBuffer
MOVWF FSR0L
MOVFW PortCache
CLRZ
ADDWF FSR0L
BTFSC STATUS, C
INCF FSR0H
```

Ce qui est nouveau, c'est que nous lisons le registre INDF0. La valeur est chargée dans le registre DAC1CON1 du CN/A :

```
MOVFW INDF0
MOVWF DAC1CON1
CALL WAIT
```

Der Rest des Programms ist eine gewöhnliche Schleife, die unter anderem die Inkrementierung vornimmt:

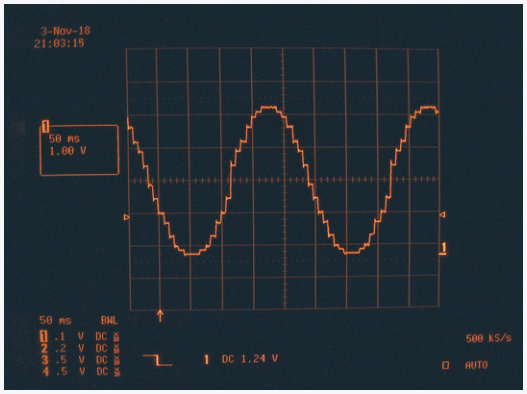


Figure 8. Le signal de sortie du CN/A correspond approximativement à un sinus de 4 Hz.

```
MOVFW PortCache
SUBLW .31
BTFSC STATUS, Z
CLRF PortCache
GOTO WORK
```

Le programme est prêt, il produit le signal de sortie sinusoïdal de la **fig. 8**.

Conclusion

Ce projet montre l'intérêt des expérimentations avec des µC à 8 bits. Vous trouverez plus d'informations sur ce sujet dans mon nouveau livre «*Microcontroller Basics with PIC*». Si vous avez apprécié cet article, faites-le moi savoir. Les commentaires constructifs sont toujours les bienvenus !

200154-03