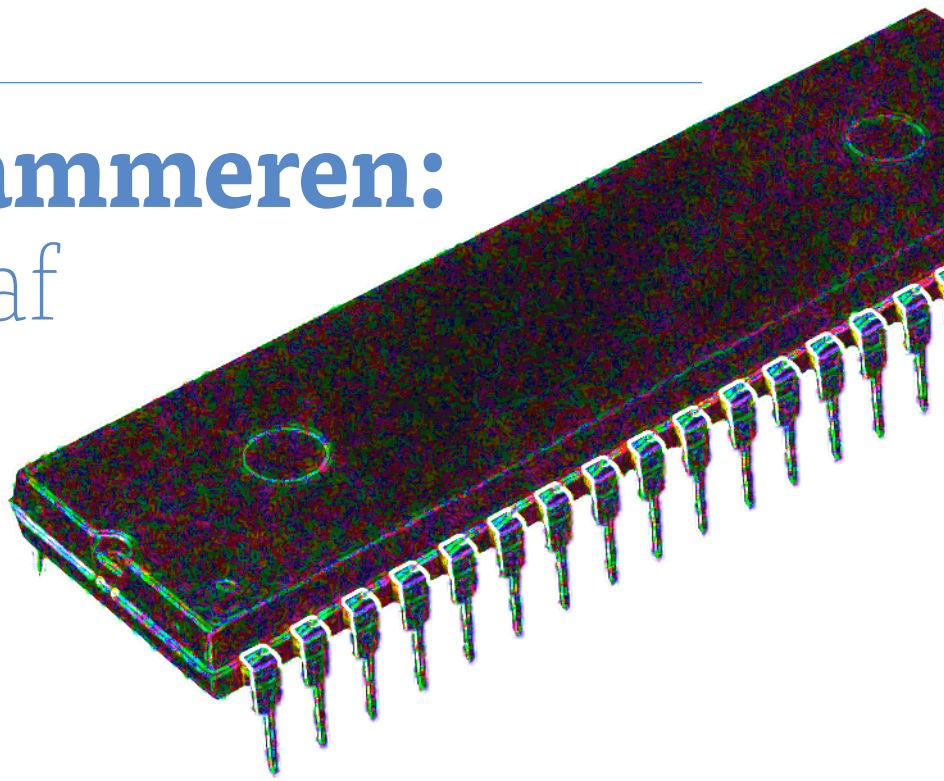
PIC's programmeren: van de grond af

een sinus in assembler



Tam Hanna (Slowakije)

Uw zegetocht begint bij de 8-bit microcontroller! Want wanneer u weet hoe u deze processoren van de grond af moet programmeren, kunt u ook de architectuur van complexere componenten doorgronden. Het is het beste om daarbij te programmeren in assembler. In dit artikel genereren we een sinus met behulp van de ADC.

We gebruiken MPLAB X als IDE en een PIC16F18877 als doelsysteem. Het programmeren verloopt via de ICSP-interface. U kunt zelf een ontwikkelboard kiezen (een breadboard volstaat eigenlijk al).
We willen een virtuele A/D-converter 'voeren' met 32 discrete waarden per periode, die we vooraf berekenen. Maar omdat de functie `sin()` in Excel niets kan beginnen met de waarden 0…31, moeten we ze eerst omzetten in een bruikbaar formaat. Het genereren van de waarden doen we in Excel, waarbij een kleine fout wordt geïntroduceerd. Helemaal links in de Excel-tabel in **figuur 1** zien we het volgnummer van elke waarde. De waarden in de tweede kolom worden berekend met de Excel-formule `=A2*((2*PI())/32)`. We krijgen dan waarden van 0 tot en met 2*π. In de derde kolom wordt `=SIN(B2)` gebruikt om de sinus van de waarden uit de tweede kolom te berekenen. Helaas heeft de sinusfunctie een waardenbereik –1…+1, terwijl de DAC alleen met positieve waarden werkt. Daarom transformeren we de waarden met `=1+C2`, zodat we waarden in het bereik 0…2 krijgen. Tenslotte transformeren we die waarden met `D3*(255/2)` naar de DAC-waarden 0…255 en ronden ze af met `ROUND(E10)`.

Tabellen uit het datageheugen

Aangezien de waarden niet veranderen, kunnen we ze opslaan in het datageheugen. Voor datatabellen gebruiken we de mnemonic RETLW, die een bepaalde waarde in W achterlaat na de RETURN.
Om technische redenen is het aan te bevelen dat tabellen op vaste adressen beginnen. We zullen later zien, waarom dat van belang is. Voorlopig gaan we er van uit, dat dit gedeelte van de code een bestemmingsadres krijgt (hier `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
```

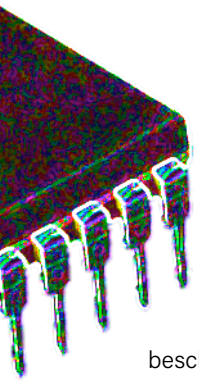
`dt` neemt de lijst met waarden in ontvangst. Elke waarde wordt een ingang in het programmageheugen. Het programma kan al gecompileerd worden, omdat de MPLAB-assembler veel kritische situaties niet als fouten beschouwt. In de output zien we alleen waarschuwingen:

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

We moeten zulke fouten serieus nemen. De tabel werkt niet, maar juist dat biedt mogelijkheden voor interessante experimenten.

Terzijde: disassembleren en constanten

Behalve bij macro's is er een direct verband tussen mnemonics en machinecode. De machinecode wordt gemaakt door de mnemonics te assembleren. We kunnen dat proces ook omkeren; dat wordt disassembleren genoemd.
Dubbelklik op *Usage Symbols Disabled* op het tabblad Dashboard dat rechtsonder in beeld verschijnt (**figuur 2**). De IDE opent dan een dialoogvenster. Vink het selectievakje *Load Symbols when programming or building for production* aan, om de analysetools in het compi-



latieproces op te nemen.
Klik dan op *Apply* en voer de compilatie opnieuw uit. De indicatoren voor het geheugengebruik worden dan bijgewerkt. Daarnaast is nu de optie *Window* > *Debugging* > *Output* > *Disassembly Listing File Project* beschikbaar. Die geeft de gedisassembleerde code weer (**figuur 3**).
De uitvoer bestaat uit zes kolommen: de getallen uiterst links beschrijven het 'logische' adres van het woord in het programmageheugen van de PIC. De volgende kolom toont de hexadecimale waarde van het commando. De derde kolom bevat de disassemblage die voortvloeit uit het hex-bestand (namen van constanten worden bijvoorbeeld niet meegenomen). De vierde kolom geeft het regelnummer in het .asm-bestand, na de dubbele punt staat de regel die verantwoordelijk is voor het hexadecimale woord verder naar links.
De code die bij onze datatabel hoort ziet er zo uit. Als u bekend bent met hexadecimale waarden, zult u meteen zien dat het merendeel van de gebruikte getallen veel te klein is:

```
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
. . .
```

De oorzaak van dit vreemde gedrag is een eigenaardigheid van de assembler. Deze kent vijf manieren om getallen te noteren. In plaats van de getallen 'zomaar' te schrijven, moeten er een punt voor elke decimale constante worden gezet om de assembler te dwingen zich correct te gedragen.

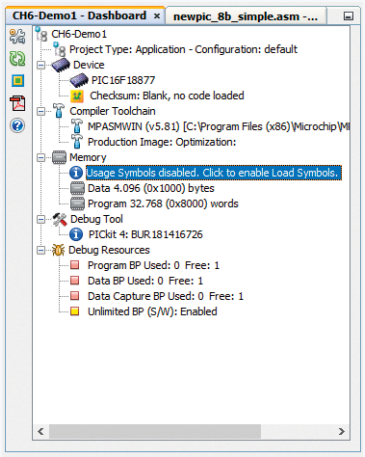
```
TABLE_VECT  CODE  0x0200 dt  .127, .152, .176, .198,
   .217, .233, .245, .252, .255, .252, .245, .233,
   .217, .198, .176, .152, .127, .102, .78, .56, .37,
   .21, .9, .2, .0, .2, .9, .21, .37, .56, .78, .102
END
```

Toegang tot de tabel

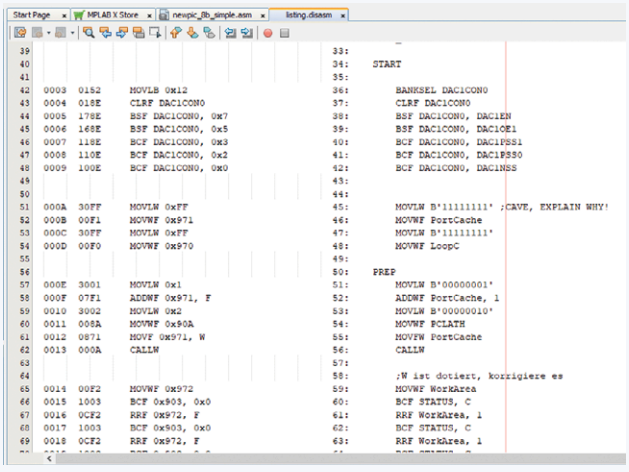
Om de in het programmageheugen geschreven informatie in te lezen, moet de controller naar het RETLW-blok springen. De return-instructies zorgen ervoor dat het W-register (omega) gevuld wordt met het betreffende getal. Om de werking te begrijpen moet het hexadecimale startadres `0x0200` worden omgezet in een binaire waarde. Het resultaat is `0000 0000 0000 0010 0000 0000`.
Sprongen naar een verplaatsbare locatie in het programmageheugen worden uitgevoerd met de CALLW-mnemonic, waarvan de declaratie is weergegeven in **figuur 4**.
Het programmageheugen van de PIC is 32768 woorden lang, dus de adrespointer moet 15 bit lang zijn. Omega is slechts 8 bit lang, zodat we extra bits uit het PCLATH-register nodig hebben om een adres aan te vullen.
Om timing-conflicten te voorkomen, heeft de PIC alleen toegang tot de PCLATH-waarde wanneer dit door mnemonics wordt vereist. We kunnen de waarde samenstellen voordat we het sprongcommando geven. Schrijven naar PCLATH beïnvloedt de programmateller niet.
De initialisatie van het programma begint met het verhogen van de lopende waarde:

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

Figuur 1. De gegevenstabel is klaar voor gebruik.



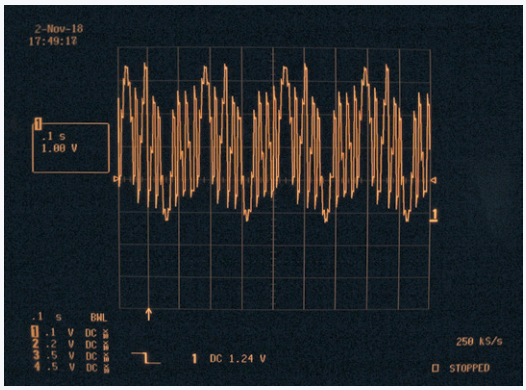
Figuur 2. Dubbelklik op deze regel om het venster met opties te openen.



Figuur 3. Disassembleren geeft informatie over de inhoud van de machinecode.

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.

Figuur 4. De CallW-mnemonic berekent het adres op een vrij ingewikkelde manier.



Figuur 5. Dit lijkt niet bepaald op een sinus…

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

We kennen het adres van de tabel en weten dus, hoe het hoogste deel van het binaire woord moet luiden. Dit moeten we via het omega-register naar PCLATH overbrengen:

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

Nu zijn we klaar voor de eigenlijke sprong. CALLW verwacht de doelwaarde in omega. Na de aanroep vinden we de geretourneerde waarde uit de tabel in omega. Die kunnen we nu naar de DAC sturen:

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

Als u het programma in zijn huidige vorm uitvoert, staat u een verrassing te wachten: de PIC 'gaat uit zijn bol' tot het programma door MPLAB wordt gestopt.
De oorzaak van dit gedrag is dat onze tabel maar 32 waarden bevat, terwijl we 255 waarden proberen te verwerken. De geheugenlocaties na de tabel zijn gevuld met willekeurige instructies die zijn overgebleven van een eerder programma. Op een gegeven moment gaan we dus spronginstructies uitvoeren naar een 'onbeschreven' geheugenbereik. De volgende code begrenst de waarde van PortCache met CLRF:

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

Tenslotte springen we terug naar boven om de volgende doorloop te starten:

```
   GOTO WORK        ; loop forever
```

Nu kunnen we ons programma uitvoeren. Maar de golfvorm van **figuur 5** lijkt amper op een sinus. De reden is dat de waarden die in het DAC-register worden geschreven van nul tot 255 lopen terwijl de DAC alleen overweg kan met waarden van nul tot 31. En dat leidt tot chaos.

Tabellen uit RAM

Natuurlijk kunnen we teruggaan naar Excel, de spreadsheet aanpassen en een bruikbare tabel maken, maar we demonstreren hier liever hoe we de parameters door de controller zelf kunnen laten aanpassen. Door een waarde één bit naar links te schuiven wordt die met een factor twee vermenigvuldigd; de waarde één bit naar rechts schuiven komt overeen met delen door twee. Wij willen van 256 naar 32, dus dat betekent delen door acht. Dat kan ook worden opgevat als drie keer na elkaar delen door twee.
Bij het 'kopiëren' van de waarden moeten we de doeladressen in het werkgeheugen berekenen. Hiervoor gaan we gebruik maken van de kernregisters. PCLATH kennen we al, nu zijn we geïnteresseerd in INDF en FSR.
Onze PIC heeft twee paar pointer- en adresregisters. Dat is handig voor ontwikkelaars die op twee plaatsen in het geheugen tegelijk willen werken. De INDF-registers (INDirect File) zijn niet fysiek geïmplementeerd. Ze wijzen naar het adres dat is ingesteld in de bijbehorende FSR's (File Select Register). Als u daar een adres in het werkgeheugen zet, kunt u op die locatie via INDF zowel lezen als schrijven. Als de FSR-registers naar het programmageheugen verwijzen, kunt u op die plek (normaal gesproken) alleen lezen.

Een bijzonderheid van de PIC is dat de keuze tussen programmageheugen en werkgeheugen wordt gemaakt met het zevende bit van het bovenste adresregister. Als dat bit 1 is, gaat het om een adres in het programmageheugen, anders om een adres in het datageheugen.
We beginnen opnieuw met het aanmaken van een variabele. Deze keer hebben we in totaal 32 byte geheugen nodig, dat is meer dan er in het 'gedeelde' geheugenbereik van de µC past.

We gebruiken geheugenruimte in een door de assembler te selecteren bank en werken met relatieve adressering. Bij MPASM kan de locatie van `DataBuffer` vrij worden gekozen dankzij het ontbreken van extra parameters:

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

Problematisch blijft het hoge en lage deel van het adres, maar gelukkig maken twee vrij onbekende operatoren van de linker het ons gemakkelijk:

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

Gewapend met deze kennis kunnen we informatie uit het programmageheugen naar het werkgeheugen kopiëren. Omdat schuifopdrachten niet rechtstreeks in W werken, is een extra variabele nodig:

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

Bij het werken met datatabellen moeten we altijd letten op de beginvoorwaarden. We initialiseren PortCache met 1111.1111 omdat de waarde in de lus wordt geïncrementeerd *voordat* hij wordt gebruikt. De eerste doorloop wordt dus gedaan met 0. Als we met 0 zouden initialiseren, zou de eerste doorloop in geheugenplaats 1 schrijven:

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

Ons programma heeft twee lussen. De `PREP`-lus is verantwoordelijk voor het invullen van de gegevenstabel, en `Work` kopieert de waarden naar de DAC. `PREP` begint met het aanroepen van de tabel:

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

Daarna moeten we de waarde verschuiven met drie RRF-instructies. Omdat schuifinstructies alleen in een F-register kunnen worden uitgevoerd, moeten we de waarde eerst tijdelijk opslaan:

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

Om te voorkomen dat er ten gevolge van het carry-bit fouten worden geïntroduceerd, roepen we voor elke RRF-instructie telkens `BCF STATUS, Z` aan. De code bevat nu nog twee kleine foutjes. Dat is opzettelijk, want de oplossing ervan is interessant!
Nu moeten we INDF naar de juiste locatie laten wijzen. Hiervoor worden de adresdata in de registers FSR0H en FSR0L geladen. Het H-register (high) krijgt het hogere deel, het L-register (low) het lagere deel:

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

INDF0 wijst nu naar het begin van het geheugengebied. We moeten daar de offset bij optellen die de individuele geheugenplaatsen identificeert. Het is niet zeker dat het begin van het veld aan het begin van een pagina staat. Als er bij het optellen van de offset een overflow optreedt in het L-deel, komt die niet automatisch in het H-deel terecht. Om dat probleem op te lossen wordt de waarde van het C-bit gecontroleerd en wordt de waarde van FSR0H verhoogd bij een overflow:

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

Het commando CLRC (Clear Carry) is een macro die het C-bit in het statusregister wist. INDF0 is daarmee correct geconfigureerd. We moeten nu de waarde die tijdelijk in het werkgebied is opgeslagen, laden en in INDF0 schrijven:

```
   MOVFW WorkArea
   MOVLW INDF0
```

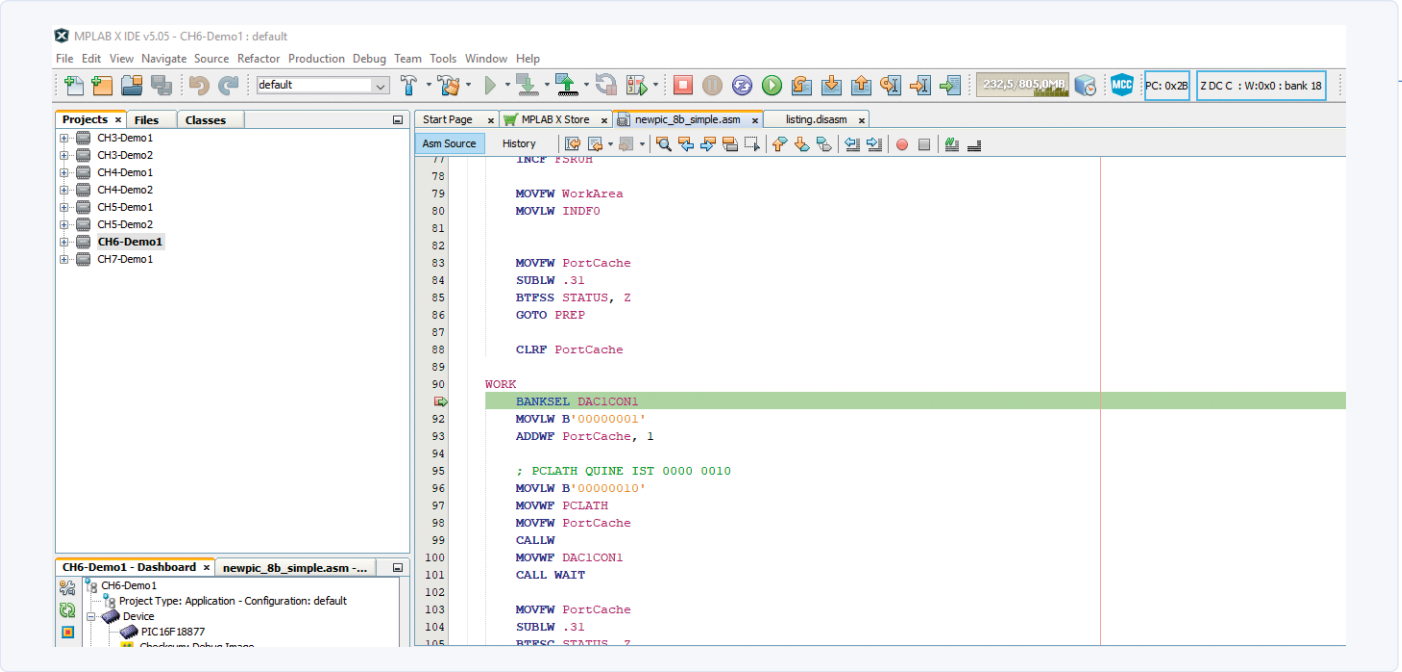
Tenslotte moeten we ervoor zorgen dat de lus wordt voortgezet.

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

Omdat de code tamelijk complex is, verdient het aanbeveling dat we ons van de juistheid van de uitvoer overtuigen.

Foutzoeken met assembler

Het plaatsen van breakpoints is in principe niet moeilijk. Klik in de IDE op een regelnummer om een rood stopsymbool in te voegen. Maar omdat de microcontroller maar één hardware-breakpoint kan beheren, krijgen we een foutmelding.
MPLAB gebruikt namelijk de debug-resources voor het stap voor stap uitvoeren van de code. Het emuleren van breakpoints in software is op dit moment niet gewenst, zodat we de foutmelding wegwerken door op *No* te klikken. Omdat de PIC software-breakpoints ondersteunt, kunt u dit bevestigen (*Yes*): zodra u meer dan één breekpunt in de assemblercode plaatst, zal Microchip de functie automatisch activeren. Omdat onze PIC slechts één hardware-breakpoint ondersteunt, klikt u in de werkbalk op de pijl naar beneden (naast het debugger-icoontje) en selecteert u de optie *Debug main project*. MPLAB opent dan een disassembler-venster, dat we meteen weer sluiten. Zodra het breakpoint is bereikt, geeft de IDE dat weer zoals in **figuur 6**.
De groen gemarkeerde regel met het pijltje aan de linkerkant is de momentele instructie. Met de stop- en sprong-pictogrammen in de



Figuur 6. De debugger heeft de uitvoering van het programma onderbroken.

werkbalk kunt u het verloop van het programma beïnvloeden. *Window Target Memory Views File Register* opent een venster dat de inhoud van het geheugen van de PIC weergeeft. Laat de muispointer boven de declaratie van DataBuffer om een tooltip-venster te openen met het adres van de eerste byte en de waarde ervan. Op het werkstation van de auteur was de positie `0xD20`.

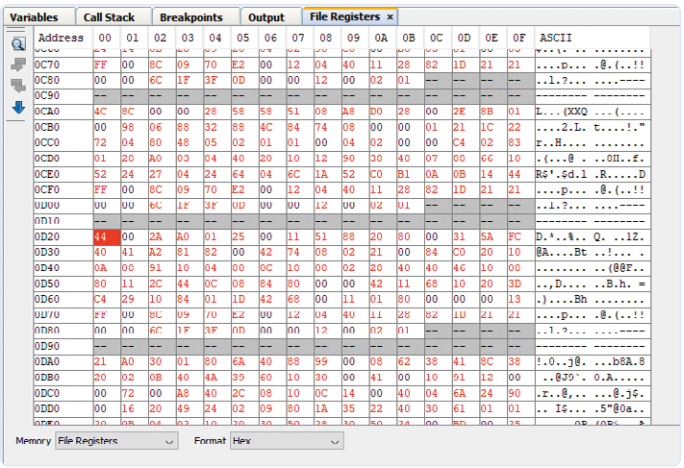
Het is handiger om op het blauwe pijltje-omlaag wijst te klikken (*GoTo*) in het venster *File Registers*. Selecteer *Symbol* in het veld *GoTo What* en kies *DataBuffer*. Sluit de popup na het aanklikken van de knop *Go To*. U krijgt dan iets te zien zoals in **figuur 7**. Het rood gemarkeerde vakje is de eerste byte van de toewijzing. Het is duidelijk dat het huidige programma niet werkt, omdat waarden zoals FC buiten het toegestane waardenbereik vallen.

Het opsporen van fouten gaat gemakkelijker, als u het geheugengebied vult met een gemakkelijk herkenbaar patroon. U kunt bijvoorbeeld de literal 1111.1111 in het indirecte register schrijven:

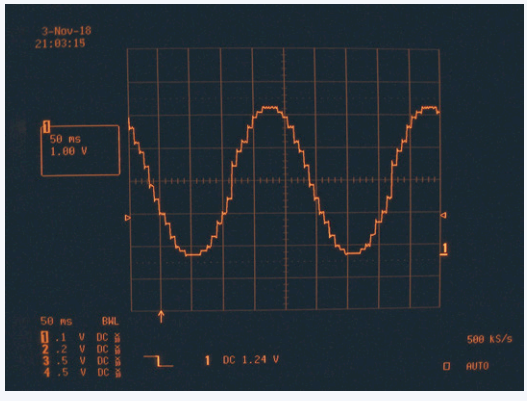
```
MOVFW PortCache
CLRZ
ADDWF FSR0L
BTFSC STATUS, C
INCF FSR0H
MOVFW B'11111111'
MOVLW INDF0
```

Als u het waardengebied opent in de debugger, ziet u een reeks met dezelfde waarden. In de meeste gevallen zou die waarde niet FF moeten zijn. In de code zit nog een foutje. We gebruiken het commando MOVLW, dat de waarde van een geheugenplaats in het omega-register laadt:

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



Figuur 7. Hier is iets vreemds aan de hand.



Figuur 8. Onze sinus (naar beste mogelijkheden van de DAC) op oscilloscoopscherm.

MPLAB beschouwt een literal zoals INDF0 als een getal: na het compileren zijn ook waarden zoals PORTA niets anders dan getallen. Ons programma kopieert dus het adres van het register naar alle 32 geheugenplaatsen. Toch zijn we nu een stap verder, want we hebben de adresgegevens al geverifieerd. Een gecorrigeerde versie van het programma ziet er zo uit:

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

Het berekenen van de geheugenadressen werkt, we kunnen nu de eigenlijke rekenfout elimineren. Het eerste probleem was het gebruik van MOVLW in plaats van MOVWF, waardoor het schrijven naar INDF werd uitgeschakeld:

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

Als we naar de uitvoer kijken, zien we dat er zeer kleine waarden verschijnen. Dat komt omdat de RRF-mnemonic met het carry-bit werkt. Onze code heeft tot nu toe echter het Z-bit gewist, wat leidt tot deze kleine verandering:

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

Het programma is nu klaar voor gebruik; de sinustabel verschijnt in het debugvenster. Om het programma af te ronden, moeten we er in de werklus voor zorgen dat de waarden uit het datageheugen worden gehaald. We moeten daarom de lopende variabele incrementeren om een doorlopende index te maken:

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

De indirecte adressering is geschikt voor lezen en schrijven. We laden de twee delen van het adres van de buffer in FSR0H en FSR0L. Daarna tellen we een offset op en controleren we of er sprake is van een overflow. Als er een overflow optreedt, verhogen we het hoge register:

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

Nieuw is dat we uit het register INDF0 lezen. De waarde gaat naar het uitgangsregister van de DAC:

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

De rest van het programma is een gewone lus die onder andere zorgt voor het incrementeren:

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

Hiermee is het programma afgerond: de uitvoer ziet er nu uit zoals in **figuur 8**.

Conclusie

Dit artikel laat zien dat we op achtbitters interessante experimenten met assembler-commando's kunnen uitvoeren. Meer hierover is te vinden in mijn nieuwe (Engelstalige) leerboek "Mikrocontroller Basics with PIC" (ook in het Duits verkrijgbaar). Ik verheug me op uw feedback!

200154-04