Laboration 3 i Digital- och Mikrodatorteknik

Utvecklingssystemet MPLAB IDE
Grundläggande assemblerprogrammering för PIC

Målet med laborationen är att få begrepp om

- Assemblerprogrammering med MPLAB IDE
- Grundläggande assembler
- Enkel inmatning

Uppgift1: MPLAB IDE

Syfte:

Lära sig använda grunderna i utvecklingsverktyget MPLAB IDE.

Beskrivning:

Vi skall i laborationerna använda oss av den utvecklingsmiljö som heter MPLAB IDE för utveckling av PIC-processorer (finns att ladda ned från www.microchip.com).
I MPLAB förekommer begreppet ”Projekt”. Det är en samling av textfiler och lite annat, som hör samman med ens arbetsuppgift. Enbart **en** .asm-fil utgör stommen i projektet, övriga filer räknas som ”include”-filer, dvs. sådana som kan kallas på av vårt huvudprogram.
Tänk på att i början av programmet ange **vem** som skrivit programmet, **när** programmet är skrivet, **filnamn**et och en **kort beskrivning** av vad programmet utför.

Skriv också in **kommentarer i programmet** så att det underlättar förståelsen.
Kommentarer skrivs efter semikolon ; .

Följande steg behövs för att förbereda program för nedladdning till microcontrollern:

1. Skapa ett projekt
2. Skriv ett program (.asm-fil)
3. Kompilera
4. Simulera

Steg1: Skapa projekt

- Click on *Project- Project Wizard* [Next]
- Step1: Select a device PIC16F877A
- Step2: Select a toolsuite Microchip MPASM Toolsuite
- Step3: Name your project lab3_1
  Project Directory C:\Program Files\MPLAB IDE\labU2
  [Next], [Finish]

Steg2: Skriv ett assemblerprogram

Teori: Assemblykod

För att kunna förstå fortsättningen, behövs vissa kunskaper om *assemblerspråket*, som är det mest maskinnära språket där varje rad motsvarar en datorinstruktion. Programexemplet illustrerar några vanliga *instruktioner* och *assemblerdirektiv* (direktiv till *assemblatorn*, programmet som översätter till maskinkod). Den text som utgör programmet kallas källkod. Källkoden är ”källan” eller ursprunget till det som ska bli ett program som mikrodatorn kan förstå. En rad består av ett antal fält:

- Symbolfält (labels)
- Instruktionsfält
- Kommentarsfält

1. Symboler

Nästan alla rader kan inledas med en symbol. Dessa skrivs alltid *längst till vänster*. I programmet är **MAIN, LOOP, ADD0, ADD1 och STOP** symboler. De definierar rader i programkoden, och dessa brukar kallas *lägen* eller *etiketter* (eng. *labels*). I programkoden använder man ofta symboler före en instruktion för att kunna hoppa till instruktionen. Symboler bör ha ett förklarande namn. I programmet skulle symbolerna ha kunnat heta SYM1 och SYM2. Detta är inte speciellt klokt. Man bör välja sina symbolnamn med omsorg, så att de beskriver vad de symboliserar.

2. Instruktioner och direktiv

Kärnan av programmet utgörs av instruktioner och assemblerdirektiv. Om man inte använder lägen framför dessa, ska de alltid inledas med *blanktecken* eller *TAB*. Direktiven motsvarar ingen speciell programkod, utan är som namnet anger, direktiv till assemblern. I vår källkod förekommer följande direktiv:

EQU ger ett värde ett namn, det som står framför EQU (Equate Symbol Value)
ORG säger till kompilatorn att placera efterföljande kod på den adress som anges (Origin)
END markerar slutet på programmet

3. Kommentarer

De rader, som *inleds med* semikolon, är kommentarer till programmet. När man skriver program i assemblerspråk, är det mycket viktigt att man då och då gör en kommentar som förklarar det som inte är uppenbart i programmet. Vad som är uppenbart, är naturligtvis olika från fall till fall. En erfaren programmerare kanske bara har några få inledande kommentarer, som beskriver vad ett programavsnitt gör. Den oerfarne däremot, skriver gärna alltför många kommentarer så att programmet ändå blir svårt att förstå. I programexemplet ovan är de flesta rader kommenterade, men det beror på att det är det första exemplet i denna kurs. En regel är, att så fort man funderat över en eller flera instruktioner, bör man skriva en liten förklaring i form av en kommentar.

Kodbeskrivning

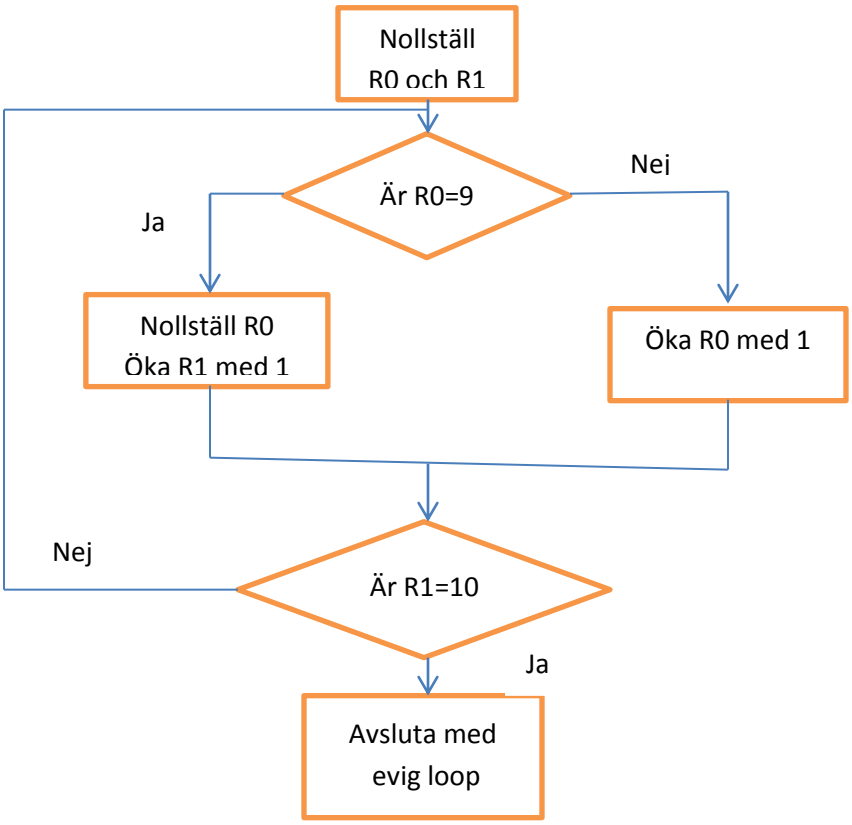
```
                LIST        P=16F877A
                INCLUDE     <P16F877A.INC>

R0              EQU         0X20
R1              EQU         0X21

                ORG         0X00
MAIN            CLRF        R0          ; nollställ filregister R0
                CLRF        R1          ; nollställ filregister R1
LOOP            MOVLW       0X9         ; ladda w-reg med 0x9
                XORWF       R0,W        ; jämför R0 och w-reg
                BTFSS       STATUS, Z   ; skippa nästa instr. om R0=w
                GOTO        ADD0        ; annars hoppa till ADD0
                CLRF        R0          ; nollställ R0
                INCF        R1,F        ; öka R1 med 1
                GOTO        ADD1        ; hoppa till ADD1

ADD0            INCF        R0,F        ; öka R0 med 1

ADD1            MOVLW       0X0A        ; ladda w-reg med 0x0A (10)
                XORWF       R1,W        ; jämför
                BTFSS       STATUS, Z   ; skippa nästa instr om R0=w
                GOTO        LOOP        ; annars hoppa till LOOP
STOP            GOTO        STOP

                END
```

Skriv in programmet enligt följande:

- Click on *File-New* Write your file and save
  New file should now be included into the project
- Right-click on *Source Files* in the window lab3_1.mcw. Choose *Add Files*.

Första delen av koden lägger upp konstanter (R0 och R1). Dessa tar ingen minnesplats utan är endast till för assemblatorn (översättningsprogrammet). Då översättningen görs ersätts de definierade namnen i koden med dess motsvarande värden.

Detta följs av den kod som utgör själva programmet. Den beskrivs enklast med ett flödesschema:

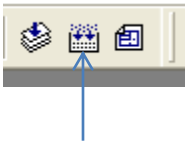


Koden räknar upp **R0** tills den blir 9. Så fort **R0** ska bli 10 slår i stället **R1** om.
R0 nollställs och allt fortsätter tills **R1** blir 10 (0x0A). Då stannar programmet. Man skulle kunna säga att **R0** räknar ental och **R1** tiotal.

Bekanta dig väl med koden innan du går vidare!

Steg3: Kompilera

- Click on *Project-Build All* (alt *Make*)
- Check that the compiling was successful (succeeded), otherwise double-click on *Error*, make a correction and compile again.



Steg4: Simulera

We can monitor current variable values, register values and status of port pins.

- Simulator is activated by clicking on *Debugger-Select Tool-MPLAB SIM*
- Reset — Click on *Debugger-Reset* [F6] alt. The yellow icon (up to the right) =>address 0 is marked (See below)

To see the contents of registers:

- Click *View-Watch* => you can mark the variables and registers which are of special interest (mark register/variable and then click *Add*).
  (Högerklicka på *Value* så kan du få värdena binärt också.)

Intressanta register/variabler att titta på här är: WREG och STATUS (till vänster)
R0 och R1 (till höger)

- Alternatives:
  1. Click *View-Special Functon Register* => the contents of all SFR
  2. Click *View-File Register* => the contents of the addressarea 0-0x7f

- (Changing inputs (instimulis, insignals)
  Click *Debugger-Stimulus-New Workbook*)

Step through the program and for each step check the contents of the registers/variables.


Starts program execution at full speed. In this example, the simulator executes the program at full (normal) speed until it is halted by clicking the icon below.


Pauses program execution. Program can continue executing step by step or at full speed again.


Starts program execution at optional speed. The speed of execution is set in dialog *Debugger/Settings/Animation/Realtime* Updates.


Starts step-by-step program execution. Instructions are executed one after another. Furthermore, clicking on this icon enables you to step into subroutines and macros.


This icon has the same function as the previous one except it has the ability to step into subroutines.


Resets microcontroller. By clicking this icon, the program counter is positioned at the beginning of the program and simulation can start.

Steg5: Köra koden

Välj ut och titta på följande register: WREG, STATUS, R0 och R1 (*View-Watch*)
Börja stega i koden (kör rad för rad) genom att klicka på **step-by-step**-knappen (alt. tryck på **F7**).

Notera hur registren ändrar sig. Man kan se vilken senaste registerskrivning är då det registret blir rödmarkerat.

Notera också hur programräknarpilen i kodrutan flyttas framåt.

Stega vidare i programmet och notera hur **R0** ändrar sig. Stega fram tills **R0** når värdet 9 och du står på första raden efter LOOP.

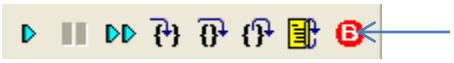
Stega förbi den raden och notera hur statusregistret **STATUS** ändras. Det datorn utför vid instruktionen XOR är att jämföra *operand1* med *operand2* (resultatet blir 0 om operanderna är lika). I vårt fall jämförs **R0** med 9.

Vad blir Z-flaggan (bit2) i STATUS-registret nu? ……..

Notera hur **R0** nollställs och **R1** ökas då man fortsätter stega.

Ett annat sätt att stega är att sätta en brytpunkt i koden och köra full fart mellan brytpunkterna. Lämpligt kan vara att sätta brytpunkten på den rad som uppdaterar **R1** (rad7, INCF R1,F) så behöver man inte stega genom uppräkningen av **R0**, nu när man vet att det fungerar.

- Sätt en brytpunkt på raden `INCF R1,F` genom att trycka på den röda markeringen.



- En gul markering kommer fram på aktuell rad i koden.

- Kör nu vidare i koden fast med **Run**-knappen. Den kommer att stanna på brytpunkten. Prova detta tills **R1** blivit 5.

- Ta bort brytpunkten och tryck på **go**. Programmet kör nu i full fart!

Notera att du inte kan se hur någonting på skärmen ändrar sig då du kör i fullfart. Detta är inte så konstigt, då registren i processorn uppdateras många gånger snabbare än vad man skulle hinna uppfatta. Vill man kunna se något får man stega eller sätta brytpunkter.

- Tryck på **paus**-knappen!

Programmet stannar nu på den raden den körde för tillfället. I vårt fall raden:

```
LOOP        GOTO  LOOP
```

Denna eviga loop är ett effektivt sätt att stanna en kod så att den inte skenar iväg i minnet. Hade man inte låst upp koden på detta sätt hade processorn fortsatt hur långt som helst i minnet. Detta är ju inte så konstigt då det enda en processor gör är att utföra instruktionscykeln.

Uppgift2: Modifiering av kod. BCD-räknare

Syfte:

Lära sig skapa egen kod med utgångspunkt från ett färdigt projekt.

Beskrivning:

Målet är att utgå från ett färdigt projekt och modifiera det till ett eget. Du ska modifiera en egen kod

Du ska nu ändra programmet enligt följande:

1. Hitta raden som ökar på **R1** med 1. Ändra registret från **R1** till **R0**, dvs. så att det blir **R0** som ökar med ett. Ta även bort raden ovanför som nollställer **R0**.
2. Kompilera och gå över i debugläge och prova programmet. Det ska nu räkna upp **R0** hela tiden.

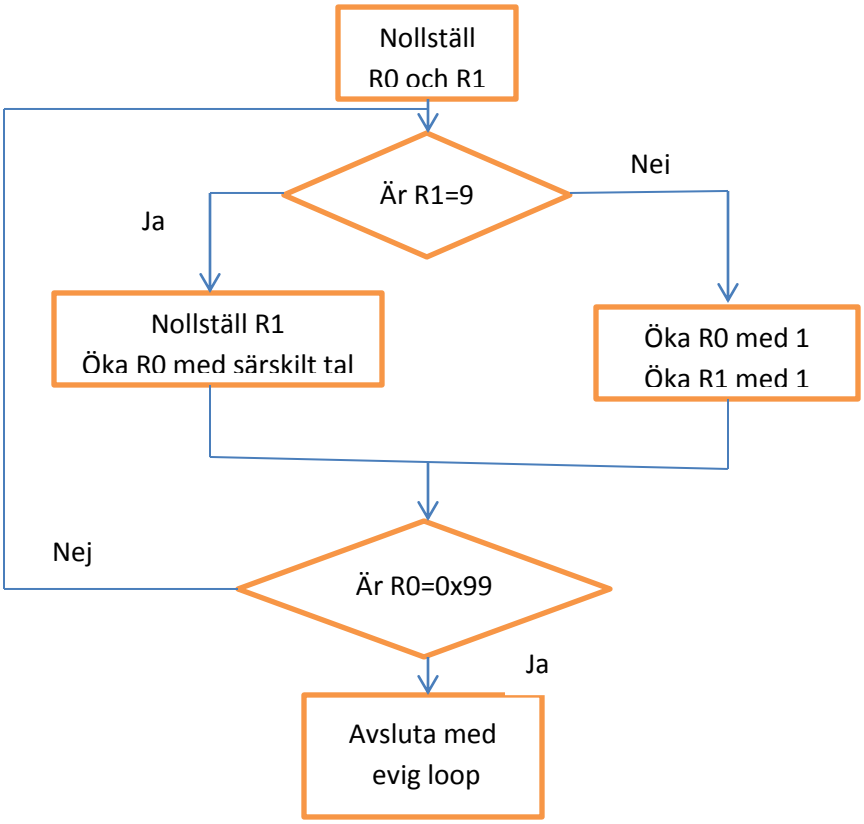
Testa och ändra tills ovanstående stämmer!
Notera att **R0** kan visas hexadecimalt/binärt/decimalt.

Steg4: Slutgiltigt program. BCD-räknare

Som slutkläm ska ett program som räknar ”decimalt” i R0 konstrueras. Man vill helt enkelt ha följande räknesekvens i R0 (i hexadecimal form)
0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15 ända till 0x99

En lösning (flera finns) är att låta ett extra arbetsregister, t ex R1, räkna upp 0-9. Varje gång det registret blir 9 ska man öka ett visst tal i R0, annars bara talet 1. På så vis kan man få R0 att räkna upp till synes decimalt. Man brukar säga att R0 då räknar BCD-kod.

Ett program som gör detta får följande flödesdiagram:



Konstruera ett program som gör detta!

Testa, simulera och visa upp!

Uppgift3: Skrivning till port

Syfte:

Lära sig konstruera enkelt program som kommunicerar med hårdvaran på PortC

Steg1: Konfigurering av PORTC

Teori:

Alla portar har ett eget datariktningsregister kopplat till sig. Det ställer portens elektriska attribut. Portarna går att konfigurera som både ut- resp. insignal. Alltså måste datariktningsregistret ställas beroende på den hårdvara som är kopplad till systemet. Initieringen av kontrollregistret sker oftast endast en gång, och då i början av koden.

I/O anslutningarna på en PIC kan konfigureras som utgångar respektive ingångar. Detta görs genom att modifiera TRIS-registret (datariktningsregistret) för respektive port. En minnesregel för att hålla reda på vad som definierar en utgång respektive en ingång är att tänka: **1=I***nput* samt **0=O***utput*. TRIS-registret modifieras genom att lagra det värde som finns i W-registret till TRIS-registret.

Alla portar på PIC (PORTA-PORTE) nås genom att läsa alt. skriva till en speciell adress. T ex har PORTC adressen 0x07 (Bank0) och tillhörande datariktningsregister TRISC adressen 0x87 (Bank1). Principen att kunna skriva till resp läsa från en port precis som om det vore en del av själva minnet kallas *minnesmappad I/O*. Huvudskälet till detta är för att spara instruktioner. Annars skulle man behöva specialinstruktioner för att nå varje port på chipet. Alltså nås den fysiska porten precis som om den vore vilken minnesplats som helst!

Adress till PORTC:s dataregister (där data hamnar och skrivs) nås genom namnet **PORTC.**

Adress till datariktningsregistret nås genom namnet **TRISC**. Dessa är fördefinierade i filen INCLUDE <P16F877.INC> som är inkluderad i projektet.

Utför följande förändringar i koden som finns:

- Lägg till följande kod mellan raderna ORG 0x00 och MAIN

```
;Initiera utgångar på PORTC
INIT    BSF     STATUS, RP0     ; övergång till BANK1
        CLRW                    ; 00000000 → W-reg
        MOVWF   TRISC           ;RB7-0 UT
        BCF     STATUS, RP0     ; återgång till BANK0
```

Steg3: Skrivning av data till PORTC

Data från/till PORTC ligger sen på en adress som kallas **PORTC**.
Vi ska nu visa R1:s värde på displayer. För att vi ska hinna med att se själva uppräkningen måste vi fördröja programmet och detta görs genom att vi här använder en fördröjningsrutin DELAY, som skrivs in efter den oändliga loopen i slutet av programmet.

```
DELAY   MOVLW       0XFF
        MOVWF       VAR2
LOOP2   MOVLW       0XFF
        MOVWF       VAR1
LOOP1   DECFSZ      VAR1,F
        GOTO        LOOP1
        DECF        VAR2,F
        GOTO        LOOP2
        RETURN
```

Komplettera också i början med (vi reserverar minnesceller för variabelvärdena):

```
VAR1    EQU         0X22
VAR2    EQU         0X23
```

Fördröjningsrutinen DELAY anropas genom att införa en rad: `CALL DELAY`

Vi ska också presentera värdet som finns i filregister R0 på 7-segmentdisplayer. Detta kan göras genom att värdet i filreg. R0 först kopieras till w-reg., vars värde därefter skrivs till PORTC.

Följande rader infogas direkt efter det att filreg. R0 och R1 har uppdaterats:

```
        MOVWF       R0,W        ;läs in R0:s värde till w-reg.
        MOVF        PORTC       ;skriv detta värde till PORTC
        CALL        DELAY       ;fördröjning så att vi hinner se vad som händer
        CALL        DELAY
        CALL        DELAY
```

Till PORTC ansluts två displayer, RC7-4 ansluts till D-A på den vänstra displayen och RC3-0 ansluts till den högra displayens D-A.

Kompilera och simulera! Glöm inte att se hur värdet i PORTC ändras.

Steg4: Programmering

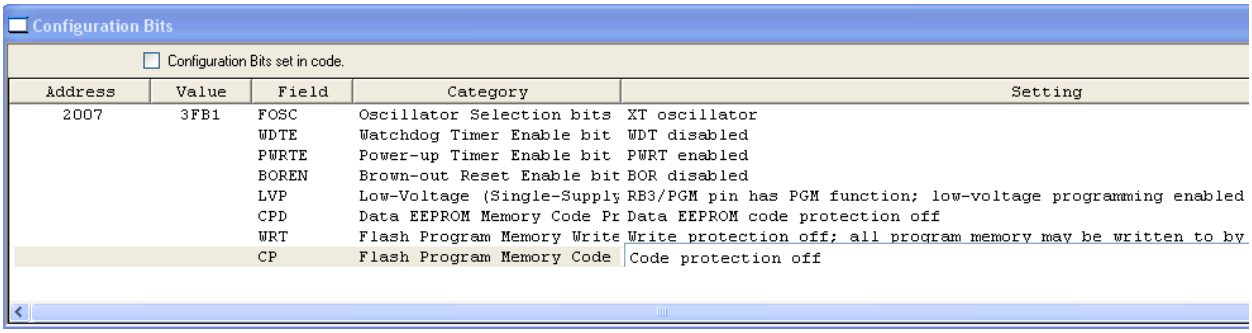
- Testa programmet i simulator
- Programmera PIC-kretsen med ditt program (se nedan)
- Koppla upp processor + 2 st displayer
- Kör programmet och verifiera funktionen

Programing the device

To program the PIC-processor we need a programmer, PICSTART PLUS. Connect PICSTART PLUS to the com-port. Place the PIC-processor in the socket. **OBS! Urfräsningen mot spaken!**

1. Click on *Programmer-Select Programmer* and choose PICSTART Plus.

2. Click on *Configure-Configuration Bits*. Choose like:



3. Click on *Programmer-Enable Programmer.* (If the com-port is wrong, change by clicking *Programmer-Settings-Communications).*

4. Click on *Programmer -Program* (programming the device).

Summering

Efter utförd laboration har studenten praktisk kunskap i följande:

Färdighet i MPLAB IDE

- Skapa och kompilera färdigt projekt
- Stega och debugga kod. Breakpoints, kontrollera register- och variabelvärden
- Skapa projekt och lägga till och modifiera filer
- Programmera microcontroller

PIC-assembler

- Förståelse för enkla instruktioner och direktiv
- Initierat PortC och gjort enkel skrivning