9. Preprocessor och kompilering

När koden är skriven måste den passera två steg innan den kan överföras till mikrokontrollern. Dessa två steg kallas tillsammans att *bygga* (eng *build*) koden. Det första är ett steg där symboler ersätts och i vissa fall enklare (statiska) uttryck kan beräknas. Det andra steget är en översättning av den resulterande preprocessade textmassan till hexadecimal kod för mikrokontrollerns FLASH-minne.

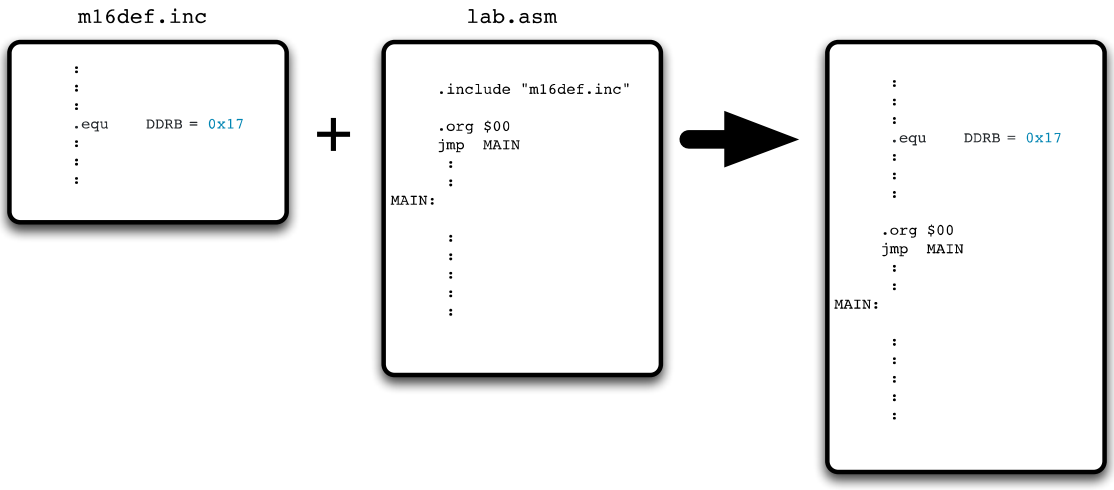
Preprocessor

Som ett inledande steg innan koden kompileras genomgår den en behandling i en *preprocessor*. Preprocessorns uppgift är att färdigställa koden så den går att kompilera, till exempel ersätts alla fördefinierade konstanter `DDRB` och så vidare med sina faktiska sifferadresser. Dessa fördefinierade konstanter är beskrivna i en *include*-fil som i Atmelstudio alltid automatiskt läggs till innan koden. I filen återfinns också storleken på tillgängligt minne med mera. Preprocessorn kan också utföra enklare beräkningar. Preprocessorns resultat går vidare till kompilering.

I den tidigare programmeringsmiljön AVRStudio behövde man själv lägga till `include`-filen i sin kod genom att inleda assemblerfilen med:

```
.include    "m16def.inc"
```

`.include` är här ett *direktiv* till preprocessorn att inkludera denna fil på plats som i figuren nedan:



Några vanliga och användbara direktiv anges nedan

Direktiv	Namn	Betydelse
`.org`	*origin*	Skriv här (adress)
`.byte`	*byte*	Namnge byte i SRAM
`.dseg`	*data segment*	Följande gäller SRAM
`.cseg`	*code segment*	Följande gäller programminnet
`.eseg`	*extra segment*	Följande gäller EEPROM
`.def`	*define*	Döp register till namn
`.equ`	*equate*	Döp konstant
`.db`	*define byte*	Skriv följande *byte* (8-bit) i minnet
`.dw`	*define word*	Skriv följande *word* (16-bit) i minnet
`.macro`	*macro*	”copy-paste” av följande
`.endmacro`	*endmacro*	...avsluta ett macro
`<<` n	*shift left*	vänsterskift n bitar
`&, |, ^`	*logical AND, OR, XOR*	bitvis OCH, ELLER, XOR
`+, -, *, /`	*arithmetic*	som förväntat
`HIGH, LOW`	*high low*	ger höga resp låga delen av följande uttryck

.org sätter kompilatorn på en specifik adress i SRAM- eller FLASH-minnet.

.cseg, .db `.cseg` anger explicit att följande kod hör till FLASH-minnet. Med `.db` kan sedan tabellvärden definieras:

```
        .cseg                       ; default
        .org    $0000
        jmp     START
        ;
        ; avbrottsvektorer
        ;
        .org    INT_VECTORS_SIZE    ; definierad i m16def.inc till 42
TAB:    .db     1, 2, 3, 4          ; Tabellen
START:
        ; Programstart
```

.dseg, .byte För att utföra definitioner i SRAM används först direktivet `.dseg`. Med `.byte` reserveras sedan ett antal bytes där. För att växla tillbaka till FLASH används `.cseg`:

```
        .dseg
        .org    $67 ; adress $67 i SRAM
ARR:                ; ARR=$67, handtag till struct nedan
VAR1:   .byte   7   ; VAR1, adressen till 0-te byten av dessa 7
VAR2:   .byte   2   ; VAR2, adressen till forsta lediga efter VAR1

        .cseg
        ; till programminnet igen
```

Observera att det inte finns någon inbyggd kontroll att de definierade adresserna för `.org` är rimliga. Det är till exempel fullt möjligt att definiera en tabell, `TAB`, med `.db` som skriver över befintlig kod:

```
        .org    $0000
        jmp     START
        ;
        ; avbrottsvektorer
        ;
        .org    $100
START:
        ; Programstart
        :
        kod
        :
        .org    $100
TAB:    .db     1, 2, 3, 4   ; <--- krasch!!
```

Exempel

Fallgrop. Varför blir även detta fel?

```
TAB:    .org    $200
        .db     1, 2, 3, 4
```

.macro Ett makro definierar ett kodstycke som skall kopieras in i koden. Skilj det alltså från ett subrutinanrop. Följande makro kan användas för att stuva undan `ZH:ZL`-registret:

```
        .macro   PUSHZ
                 push   ZH
                 push   ZL
        .endmacro

        :
        PUSHZ          <-- ersätts med de två raderna
        :
```

Använt som ovan har makrot enbart en kosmetisk effekt på koden. Ett makro kan dock ta argument vilket gör det mer användbart:

```
        .macro   SUBIW                ; macro med argumenten @0, @1 och @2
                 subi  @1,LOW(@0)
                 sbci  @2,HIGH(@0)
        .endmacro

        :
        SUBIW    $1234,r16,r17   ; r17:r16=r17:r16-$1234
                 ; @0  @1  @2
        :
```

Övrigt Preprocessorn kan också göra livet enklare för assemblerprogrammeraren med sitt stöd för aritmetiska och logiska operationer. Observera att dessa beräkningar görs innan kompileringen och långa uttryck resulterar alltså inte i längre kod.

```
ldi         r16,(1<<3)|(1<<ADSC) ; biten ADSC och bit 3 ett-satta i r16
        :
ldi         r16,HIGH(42*31)      ; högsta byten av 42*31=$0516, dvs $05
ldi   r17,LOW(42*31)      ; lägsta byten samma dvs $16
```

Kompilering

Kompileringssteget är det steg som känner igen assemblerinstruktioner och kan översätta de till hexadecimala tal som skall programmeras in i mikrokontrollerns FLASH-minne. Kompileringen sker i två steg, en så kallad *två-pass*-assembler (*two pass assembler*):

1. Först analyseras programkoden med avseende symboliska adresser och konstanter. *Labels*, exempelvis `START:`, ersätts med sina faktiska värden.
2. Med informationen ovan kan det andra steget köras. Detta är det egentliga kodgenererande steget. Här översätts assembler instruktioner till hexadecimala tal.

För den extra intresserade belyser vi hela processen i mer detalj med ett exempel.

Programraden

```
        START:  ldi         r16,HIGH(RAMEND)
```

omformas under byggsteget till hexadecimala tal:

```
START:  ldi         r16,HIGH($045F)  ; Preprocessorn ersätter RAMEND
START:  ldi         r16,$04          ; Preprocessorn använder HIGH på $045f
$0044:  ldi         r16,$04          ; Kompilatorn sätter adresser
$0044:  $E004                  ; Kompilatorn genererar hexadecimala tal
```

Kikar man närmare på det färdiga talet `E004` kan man se att det består av flera olika delar: Alla `ldi`-instruktioner börjar till exempel med `E` och innehåller sedan bitar som motsvarar argumenten, det vill säga

- vilket register som är inblandat (där `r16` är det nollte) och
- vilken konstant det handlar om (`$04` här).

För att vi inte ska behöva peta i detaljerna sköter kompilatorn om detta åt oss.

Nedan visas ett stycke kod med kompilerad form i en kolumn till höger:

```
        ldi     r16,$04          E004
        out     SPH,r16          BF0E
        out     SPL,r16          BF0D
        out     DDRB,r16         BB07
        clr     XL               27AA
        clr     XH               27BB
        ldi     ZH,HIGH(TEXT*2)  E0F0
        ldi     ZL,LOW(TEXT*2)   E6EE
BLANK:  rcall   TWOBEEP          D016
GETCH:  rcall   TWOBEEP          D015
          :
```

Högerkolumnen placeras sedan vid processorns programmering i FLASH-programminnet:

```
RAD  HEX
0000 E004
0001 BF0E
0002 BF0D
0003 BB07
0004 27AA
0005 27BB
0006 E0F0
0007 E6EE
0018 D016
0019 D015
```

För läslighets skull visas hexkoden skriven som ovan. I det fysiska minnet är hög och låg byte ombytta som vanligt: byte `0000` innehåller `04` och så vidare.

För den extra extra intresserade

Kompilatorns utfil är en så kallad *hexfil* (`.hex`) med ett standardiserat format:

```
:020000020000FC
:1000000004E00EBF0DBF07BBAA27BB27F0E0EEE65A
:1000100016D015D005910030...
```

Raderna som börjar med `:1` innehåller de hexadecimala talen som skall programmeras. Först kommer en adress och sedan 32 bytes med information. Det `5A` som är sist på raden återfinns inte i den hexadecimala koden utan är en checksumma för hela den raden.

—o-Ö-o—