Datorteknik
Hyfsa kod

Michael Josefsson

Version 0.2 2018

Denna text beskriver hur man kan hyfsa kod till att bli både bättre, mer lättläst och mindre. Som exempel används en del av den kod som behövs för att sända ut morsetecken. Hela koden är inte angiven, men tillräckligt mycket för att kunna förenkla och diskutera runt den.

Lägg märke till att man ofta **inte behöver förstå vad koden faktiskt gör** för att genomföra hyfsningarna nedan. Det handlar om mekaniska transformationer för att:

- Formattera kod till läslighet
- Undvika onödiga och trassliga hopp
- Undvika repetition av kodstycken
- Införa parametrar/argument
- Generalisera rutiner
- Använda beskrivande namn på rutiner
- Rutinerna skall göra det de heter
- Sätta globala konstanter i kodens början

Notera att många av punkterna ovan automatiskt och från början blir uppfyllda med koden strukturerad enligt JSP.

Låt oss starta: Betrakta följande listning av ett första försök till en morsesändarkod.

```
  //Anvandning av subrutiner mojlig
    ldi     r16, HIGH(RAMEND)
    out     SPH, r16
    ldi     r16, LOW(RAMEND)
    out     SPL, r16

    ldi r17, $FF //ljud ut
    out DDRB, r17
  jmp START

;TEST:
; sbi portb,7
; call SOUND
; cbi portb,7
; jmp TEST

    .equ PITCH = 20
    .equ ONESOUND = 50
.equ SPEED = 50
//    .equ TWOSOUND = 2 * ONESOUND
//    .equ THREESOUND = 3 * ONESOUND

 //.equ String, "DATORTEKNIK"
 //.db String
MESSAGE:
    .db "DATOR TEKNIK",$00 //.db eller annan?

BTAB:
    .db    $60, $88,.... $C8 //alfabetet eller tabell? Hex eller vanligt?

START:
    clr        r16
    ldi        ZH, HIGH(MESSAGE * 2)
    ldi        ZL, LOW (MESSAGE * 2)
```

```
TRAVERSE:
        clr         r16
        lpm         r16, Z
        cpi         r16, $00
        breq    START
        call    MORSE
TRAV2:
        inc         ZL
        jmp         TRAVERSE


MELLANSLAG:
call NOSOUND
call NOSOUND
call NOSOUND
call NOSOUND

jmp TRAV2


MORSE:
subi    r16, $41
brmi MELLANSLAG
        call    LOOKUP ;far in en bokstav
        //lsl tills tom
        clc


        AGAIN:
            lsl         r16 ; shifta binarkoden (hex)

cpi        r16, 0//om det ar slut
breq KLARMEDTECKNET

            brcc   CHECKDONE //kollar om det ar en etta pa carry platsen
call   SOUND; om etta
call   SOUND; om etta
CHECKDONE:
            call    SOUND; om nolla
            call NOSOUND
jmp     AGAIN

KLARMEDTECKNET:
call NOSOUND
call NOSOUND
jmp TRAV2
            //ret
            //call    SEND

/*
CHECKDONE:
        cpi         r16, 0//om det ar slut
       brne    XXSOUND



XXSOUND:
call    SOUND; om etta
call    SOUND; om etta
call    SOUND; om etta
        jmp    AGAIN      */

LOOKUP:
        push    ZH
        push    ZL

        ldi         ZH, HIGH(BTAB * 2)
        ldi         ZL, LOW (BTAB * 2)

        //assci -> hex -> binar
        add         ZL, r16
```

```
lpm     r16, Z
        //brne     SOUND
        //inc        ZH
        pop         ZL
        pop         ZH
        ret
        //lagga over i register och se om det blir 44 t ex

NOSOUND:
        ldi r21,SPEED
NOSOUND1:
        cbi         PORTB,7
        call        DELAY
        cbi         PORTB,7
call DELAY
dec    r21
brne NOSOUND1
ret

SOUND:
        //lpm         r16, Z
ldi r21,SPEED
SOUND1:
        sbi         PORTB,7
        call        DELAY
        cbi         PORTB,7
call DELAY
dec    r21
brne SOUND1
ret

SEND:
        //if bit == 0?? hur kollar man det? behover vi kolla?
        sbrs    r16, 0 //om == 0 , gor det nedan, annars hoppa over
        ;call    BEEP(1)
        sbrc    r16, 0 //om == 1, gor det nedan, annars hoppa over
        ;call    BEEP(3)
```

Koden bär spår av olika provade försök och bortkommenterade rader och kommentarer som tydligt inte är relevanta längre.

En första hyfsning är att formattera koden till läslighet. Copy-paste från annat dokument eller pdf förlorar ofta sin formattering. Acceptera **inte** det, formattera om! Indentera **alltid!**

I denna kod har man använt versaler för labels, det fortsätter vi med. Man har indenterat koden i olika nivåer, något man aldrig gör med assemblerkod. I assembler skall labels börja i kolumn 0, instruktioner ett tabstopp in och argument ytterligare ett tab in. För bästa läslighet efter många timmar vid skärmen brukar man föreslå tabstopp om 8 mellanslag, så ska vi också göra.

Här kan också kommentarer rensas bort. Med rätt valda rutinnamn är ofta kommentarer överflödiga. För att enklare referera till enskilda rader har radnummer tillagts i vänstermarginalen.[1] Just nu ligger kommentarerna kvar som referenser men de tas bort i samtliga senare listningar.

[1] Observera att dessa har inget med faktiska programadresser (”`.org 200`”) att göra.

I texten refereras till radnummer i ***denna*** $\downarrow$ ***listning*** *genom nummer inom parentes, dvs rad 7 anges som (7).*

```
0           //Anvandning av subrutiner mojlig
1           ldi     r16,HIGH(RAMEND)
2           out     SPH,r16
3           ldi     r16,LOW(RAMEND)
4           out     SPL,r16
5
6           ldi     r17,$FF //ljud ut
7           out     DDRB,r17
8           jmp     START
9
10          .equ    PITCH = 20
11          .equ    ONESOUND = 50
12          .equ    SPEED = 50
13          .equ    TWOSOUND = 2 * ONESOUND
14          .equ    THREESOUND = 3 * ONESOUND
15
16  MESSAGE:
17          .db     "DATOR TEKNIK",$00 //.db eller annan?
18  BTAB:
19          .db     $60, $88,.... $C8 //alfabetet eller tabell? Hex eller vanligt?
20  START:
21          clr     r16
22          ldi     ZH,HIGH(MESSAGE * 2)
23          ldi     ZL,LOW (MESSAGE * 2)
24
25  TRAVERSE:
26          clr     r16
27          lpm     r16,Z
28          cpi     r16,$00
29          breq    START
30          call    MORSE
31  TRAV2:
32          inc     ZL
33          jmp     TRAVERSE
34
35  MELLANSLAG:
36          call    NOSOUND
37          call    NOSOUND
38          call    NOSOUND
39          call    NOSOUND
40          jmp     TRAV2
41
42  MORSE:
43          subi    r16,$41
44          brmi    MELLANSLAG
45          call    LOOKUP ;far in en bokstav
46          //lsl tills tom
47          clc
48
49  AGAIN:
50          lsl     r16 ; shifta binarkoden (hex)
51          cpi     r16, 0//om det ar slut
52          breq    KLARMEDTECKNET
53          brcc    CHECKDONE //kollar om det ar en etta pa carry platsen
54          brcs    SHORTSOUND
55  SHORTSOUND:
56          call    SOUND; om nolla
57          jmp     AGAIN
58
59
60  KLARMEDTECKNET:
61          call    NOSOUND
62          call    NOSOUND
63          jmp     TRAV2
64          //ret
65          //call    SEND
66
67  CHECKDONE:
68          cpi     r16, 0//om det ar slut
69          brne    XXSOUND
```

```
70          call    NOSOUND
71          call    NOSOUND
72          call    NOSOUND
73          jmp     AGAIN
74
75  XXSOUND:
76          call    SOUND; om etta
77          call    SOUND; om etta
78          call    SOUND; om etta
79          jmp     AGAIN
80
81  LOOKUP:
82          push    ZH
83          push    ZL
84          ldi     ZH,HIGH(BTAB * 2)
85          ldi     ZL,LOW (BTAB * 2)
86          //ascii -> hex -> binar
87          add     ZL,r16
88          lpm     r16,Z
89          //brne  SOUND
90          //inc   ZH
91          pop     ZL
92          pop     ZH
93          ret
94          //lagga over i register och se om det blir 44 t ex
95
96  NOSOUND:
97          ldi     r21,SPEED
98  NOSOUND1:
99          cbi     PORTB,7
100         call    DELAY
101         cbi     PORTB,7
102         call    DELAY
103         dec     r21
104         brne    NOSOUND1
105         ret
106
107 SOUND:
108         //lpm   r16, Z
109         ldi     r21,SPEED
110 SOUND1:
111         sbi     PORTB,7
112         call    DELAY
113         cbi     PORTB,7
114         call    DELAY
115         dec     r21
116         brne    SOUND1
117         ret
118
119 SEND:
120         //if bit == 0?? hur kollar man det? behover vi kolla?
121         sbrs    r16,0 //om == 0 , gor det nedan, annars hoppa over
122         ;call   BEEP(1)
123         sbrc    r16,0 //om == 1, gor det nedan, annars hoppa over
124         ;call   BEEP(3)
```

Med koden sålunda formatterad kan man börja studera den översiktligt: Från början sätts stackpekaren (1) för att kunna använda subrutiner. (6–7) konfigurerar sedan `PORTB` som utgång innan ett hopp till `START` tar exekveringen förbi programmets globala konstanter `.equ` och de båda strängarna `MESSAGE` och `BTAB`. Båda strängarna utförs som enstaka bytes, den första definieras direkt i ASCII och den andra som hexadecimala tal.

`START` inleds med att rensa `r16` ”för säkerhets skull” som det brukar kallas. Ofta är detta steg helt onödigt, här speciellt eftersom den rensas en gång till innan användning på (26). Båda raderna (21, 26) är ändå onödiga då `lpm r16,Z` strax läser in ett nytt värde i `r16`. Varför ska man då nollställa registret först?

Det är tydligt att `TRAVERSE` används för att traversera igenom strängen `MESSAGE`. Att då manuellt stega fram i strängen byte för byte (`inc ZL`)[2] är onödigt då det likaväl kan utföras med postinkrement i `lpm`-instruktionen.

```
20  START:
21          ldi     ZH,HIGH(MESSAGE * 2)
22          ldi     ZL,LOW (MESSAGE * 2)
23
24  TRAVERSE:
25          lpm     r16,Z+
26          cpi     r16,$00
27          breq    START
28          call    MORSE
29  TRAV2:
30          jmp     TRAVERSE
```

Det är svårt att placera rutinen `MELLANSLAG` då den å ena sidan gör ett explicit hopp in i `TRAVERSE` men den anropas från någon annanstans. Den får stå kvar men vi gör en mental not att den antagligen inte är på rätt ställe.

```
37  MELLANSLAG:
38          call    NOSOUND
39          call    NOSOUND
40          call    NOSOUND
41          call    NOSOUND
42          jmp     TRAV2
```

---

[2]Enbart `inc ZL` är dessutom inte tillräckligt i det allmänna fallet då hela pekaren är två bytes stor.

`MORSE` hoppas till (inte *anropas* då det inte är en subrutin!) med ett ASCII-kodat tecken i `r16`. Om tecknet är mindre än `'A'` sker hopp till `MELLANSLAG`, i annat fall anropas `LOOKUP` med `r16` nu innehållande ett positivt ordningstal motsvarande bokstavens position i alfabetet ('`A`' = 0, '`B`' = 1 osv).

`call LOOKUP` är kommenterad med ”får in en bokstav”. Vad betyder det? En bättre kommentar är ”översätt till binärkod” eller något ditåt. Om ens kommentaren behövs, *lookup* är ju exakt vad det handlar om. Kommentaren stryks.

Labeln `AGAIN` antyder att det som följer kommer ske många gånger och inleds med ett logiskt vänsterskift. Hmm... det eliminerar behovet av `clc` på raden innan också!

I databladet för processorn kan man läsa vilka flaggor `lsl` påverkar, den påverkar bland annat `C` och `Z` varför den efterföljande instruktionen som testar likhet med noll är onödig då `Z` redan är korrekt satt. `cpi` stryks alltså.

Om `r16` skulle vara lika med noll efter skiftet sker hopp till `KLARMEDTECKNET`. Om inte, studeras den utskiftade biten i carry och om `C=1` sker hopp till `CHECKDONE`. I annat fall testas om den utskiftade biten är `0` och genom ett hopp fortsätter programmet på nästa rad. Hopp till nästa rad? Två saker här: 1) Om carryn inte var 0 måste den vara 1, så det villkorliga hoppet (54) behövs inte. 2) Om hoppet ska ske till nästa rad behövs inget hopp, det är ju det programmet normalt gör. De raderna tas alltså bort!

De därefter kommande `ret` och `call SEND` kommer aldrig kunna nås med sin nuvarande placering, är antagligen gamla rester, så de tas också bort.

```
42  MORSE:
43          subi    r16,$41
44          brmi    MELLANSLAG
45          call    LOOKUP
46  AGAIN:
47          lsl     r16
48          breq    KLARMEDTECKNET
49          brcc    CHECKDONE
50          call    SOUND
51          jmp     AGAIN
52
53  KLARMEDTECKNET:
54          call    NOSOUND
55          call    NOSOUND
56          jmp     TRAV2
57
58  CHECKDONE:
59          cpi     r16,0
60          brne    XXSOUND
61          call    NOSOUND
62          call    NOSOUND
63          call    NOSOUND
64          jmp     AGAIN
65
66  XXSOUND:
67          call    SOUND
68          call    SOUND
69          call    SOUND
70          jmp     AGAIN
```

Med listningen förkortad och förenklad enligt ovan träder programmerarens tankemönster fram. I `AGAIN`, om C=0 sänd 1 `SOUND` om C=1 sänd 3 `SOUND` om inte `r16` samtidigt är noll för då skall 3 `NOSOUND` utföras. Allt detta under rubriken `CHECKDONE`. Det är inte bara svårt att beskriva i ord, rubriken är också gravt missvisande.

Man vill skilja mellan att

1. göra ett `SOUND` och
2. göra tre `SOUND` om inte `r16`=0 för att då göra tre `NOSOUND` istället?

då punkt 1 gör **en enda** sak medan punkt 2 är mer komplicerad med **två** möjliga utfall. I någon mening är de två punkterna inte ortogonala utan inslingrade i varann.

En mer strukturerad gång är, speciellt då `cpi r16,0` redan gjorts i och med `breq KLARMEDTECKNET`, följande:

1. om `r16 != 0`
   a) om C=1 gör tre `SOUND`
   b) om C=0 gör en `SOUND`
2. om `r16 == 0` gör tre `NOSOUND`

Här testas `r16` enbart en gång och en eller tre `SOUND` utförs på samma logiska nivå i programmet.

En omarbetad version av kodstycket ovan blir

```
42  MORSE:
43          subi    r16,$41
44          brmi    MELLANSLAG
45          call    LOOKUP
46  AGAIN:
47          lsl     r16
48          breq    KLARMEDTECKNET
49          brcc    SHORT
50          call    SOUND
51          call    SOUND
52  SHORT:
53          call    SOUND
54          jmp     AGAIN
55
56  KLARMEDTECKNET:
57          call    NOSOUND
58          call    NOSOUND
59          ret
```

Sådär, det blev ju mycket bättre och kortare. Labeln `SHORT` säger dessutom bättre vad det handlar om.

Återhoppet till `TRAV2` (40) är felaktigt med tanke på att programmet kom hit genom ett subrutinanrop (30). Vill vi tillbaka är `ret` den rätta instruktionen, så det har också korrigerats, sista raden i kodstycket till vänster.

Nu är `MORSE` en subrutin med **en** *entry point* och **en** *exit point*. Den fungerar på grund av deluppgifter som utförs i subrutiner någon annanstans och de hopp som finns kvar är på nödvändiga ställen och hoppen sker också till uppenbart bra labels/ställen.

När ska man sluta omorganisera koden på det här sättet? Den frågan går inte att ge ett exakt svar på. ”När det ser bra ut” kan man vara nöjd men det dyker upp möjligheter att snygga upp koden hela tiden. Här krävs ett visst mått av ”*känsla för feeling*”.

Sedan börjar en ny del av programmet. Ny del, då det är få kopplingar mellan efterföljande rutiner och de mer centrala sammanhållande delarna ovan. Först ut av dessa *hjälp*rutiner är `LOOKUP`:

```
81  LOOKUP:
82          push    ZH
83          push    ZL
84          ldi     ZH,HIGH(BTAB * 2)
85          ldi     ZL,LOW (BTAB * 2)
86          add     ZL,r16
87          lpm     r16,Z
88          pop     ZL
89          pop     ZH
90          ret
```

Kort, koncis och räddar pekarregistret `Z` genom stackhantering. Inget att anmärka på, den gör en uppslagning som namnet antyder.

Notera: Man kan luras att skriva rutinen som:

```
81  LOOKUP:
82          :
83          ldi     ZL,LOW (BTAB * 2)
84          subi    r16,$41
85          add     ZL,r16
86          lpm     r16,Z
87          :
88          ret
```

dvs de låter den utföra subtraktionen för att beräkna index också. Det är misstänkt! Plötsligt gör rutinen två saker: dels tabelluppslagningen som namnet lovar, dels omräkning av ASCII-värden. Det förut föreslagna upplägget är bättre, då `subi`-instruktionen mer naturligt passar på (43).

Rutinerna `SOUND` och `NOSOUND` inte bara låter lika till namnet de innehåller också i huvudsak samma kodrader. I en strävan att inte onödigtvis duplicera kod måste de betraktas tillsammans:

```
96  NOSOUND:
97          ldi     r21,SPEED
98  NOSOUND1:
99          cbi     PORTB,7
100         call    DELAY
101         cbi     PORTB,7
102         call    DELAY
103         dec     r21
104         brne    NOSOUND1
105         ret
106
107 SOUND:
108         ldi     r21,SPEED
109 SOUND1:
110         sbi     PORTB,7
111         call    DELAY
112         cbi     PORTB,7
113         call    DELAY
114         dec     r21
115         brne    SOUND1
116         ret
```

Åtminstone kan man klippa ur de gemensamma raderna och göra en subrutin, `SOUNDDELAY`, av dem:

```
96  SOUNDDELAY:
97          call    DELAY
98          cbi     PORTB,7
99          call    DELAY
100         ret
101
102 NOSOUND:
103         ldi     r21,SPEED
104 NOSOUND1:
105         cbi     PORTB,7
106         call    SOUNDDELAY
107         dec     r21
108         brne    NOSOUND1
109         ret
110
111 SOUND:
112         ldi     r21,SPEED
113 SOUND1:
114         sbi     PORTB,7
115         call    SOUNDDELAY
116         dec     r21
117         brne    SOUND1
118         ret
```

Det gör rutinerna mer kompakta och duplicering av kod undviks.

Mer elegant kan dock vara att tillverka en **generell** ljudrutin med ett argument som avgör om något skall låta eller inte. Här används registret `r22` som detta argument. Vinsten är dels att rutinen är generell, dels att `SOUNDDELAY` inte längre behövs då innehållet i denna inte behöver förekomma på mer än ett ställe:

```
 96  SOUND:
 97          ldi     r21,SPEED
 98  SOUND1:
 99          out     PORTB,r22
100          call    DELAY
101          cbi     PORTB,7
102          call    DELAY
103          dec     r21
104          brne    SOUND1
105          ret
```

`SEND` är en rest som bortkommenterades tidigare så den tas bort helt nu då den saknar relevans i den hyfsade koden.

```
119
120  SEND:
121          //if bit == 0?? hur kollar man det? behover vi kolla?
122          sbrs    r16,0 //om == 0 , gor det nedan, annars hoppa over
123          ;call   BEEP(1)
124          sbrc    r16,0 //om == 1, gor det nedan, annars hoppa over
125          ;call   BEEP(3)
```

Efter hyfsningen enligt ovan har koden blivit mindre, mer strukturerad, enklare att förstå och underhållsvänligare. Tomrader har använts som separator mellan kodstycken för att markera funktionella enheter för tydlighets skull. Stora delar av diskussionen ovan kunde undvikits om konstruktionen från början utgått från metoden med strukturerad programmering enligt JSP.

Resultatet av hyfsningen är nu ungefär som nedan. Argument för `SOUND` har lagts till. Antalet kodrader har minskat från cirka 125 till blott 70, **ungefär 40 procent har skalats av enbart genom omstrukturering till en tydligare kod!**

Observera också att denna omstrukturering har gjorts utan att veta vad koden handlar om! Ändå har den blivit bättre!

```
 0          ldi     r16,HIGH(RAMEND)
 1          out     SPH,r16
 2          ldi     r16,LOW(RAMEND)
 3          out     SPL,r16
 4
 5          ldi     r17,$FF //ljud ut
 6          out     DDRB,r17
 7
 8  START:
 9          ldi     ZH,HIGH(MESSAGE * 2)
10          ldi     ZL,LOW (MESSAGE * 2)
11
12  TRAVERSE:
13          lpm     r16,Z+
14          cpi     r16,$00
15          breq    START
16          call    MORSE
17  TRAV2:
18          jmp     TRAVERSEMELLANSLAG:
19          ldi     r22,$00
20          call    SOUND
21          call    SOUND
22          call    SOUND
23          call    SOUND
24          jmp     TRAV2
25
26  MORSE:
27          subi    r16,$41
28          brmi    MELLANSLAG
29          call    LOOKUP
30  AGAIN:
31          lsl     r16
32          breq    KLARMEDTECKNET
33          brcc    SHORT
34          ldi     r22,$FF
35          call    SOUND
36          call    SOUND
37  SHORT:
38          ldi     r22,$FF
39          call    SOUND
40          jmp     AGAIN
41  KLARMEDTECKNET:
42          ldi     r22,$00
43          call    SOUND
44          call    SOUND
45          ret
46
47  LOOKUP:
48          push    ZH
49          push    ZL
50          ldi     ZH,HIGH(BTAB * 2)
51          ldi     ZL,LOW (BTAB * 2)
52          add     ZL,r16
53          lpm     r16,Z
54          pop     ZL
55          pop     ZH
56          ret
57
58  SOUND:
59          ldi     r21,SPEED
60  SOUND1:
61          out     PORTB,r22
62          call    DELAY
63          cbi     PORTB,7
64          call    DELAY
65          dec     r21
66          brne    SOUND1
67          ret
```

I det här skicket kan det vara värt att kompilera och provköra koden. Än så länge är den varken korrekt eller komplett men avsevärt enklare att arbeta med och tänka om. Ett exempel: Om ett `NOSOUND` skall tillföras efter varje kort och långt ljud (teckendel), var ska koden i så fall ändras? Det handlar om att lägga till en enstaka kodrad. Med koden strukturerad enligt JSP är det enkelt (enklare?) att hitta exakt **var**!

I denna version av koden framstår nya optimeringsmöjligheter då det är åskilliga `call SOUND` efter varann. Inte sällan kombinerat med `ldi r22,$`*xx*. Koden blir trots allt plottrig med dessa. Ett försök till ytterligare hyfsning är då att återinföra `NOSOUND` som tystnad och `DOSOUND` för ljud.

```
NOSOUND:
        ldi     r22,$00
        call    SOUND
        ret
```

```
DOSOUND:
        ldi     r22,$FF
        call    SOUND
        ret
```

—o=Ö=o—