Strukturerad Assembler

Petter Källström

Version 1.1, 7 maj 2021

Detta dokument är tänkt som en enklare guide inom några aspekter för den som vill programmera assembler på AVR. Det mesta är egna reflektioner, ibland mixat med inspiration från andra människor eller webbsidor.

Även om den riktar sig till AVR-processorn, så kan tankesättet även inspirera programmering för andra plattformar.

1 Syfte

I datorteknikkurserna har vi några syften:

1. Du ska lära dig hur en dator fungerar, och till det är assembler helt centralt.
2. Så som i resten av utbildningen, så ska du lära dig skriva snygg kod som är lätt att förstå, felsöka/verifiera och underhålla (underhålla = vidareutveckla och/eller anpassa till ändrade förutsättningar).

Dessa två går nästan parallellt. Där de skiljer sig tenderar vi att prioritera punkt 2, och då delen som handlar om att förstå. Som laborationsassistent kan jag dock tycka att delen "felsöka/verifiera" är ännu relevantare. Exempel på när det skiljer: Funktionen `DELAY_T` har en loop som snurrar 4000 varv. Siffran 4000 kan definieras som konstanten `"T_NUM"` överst, eller så skrivs den in som den är i `DELAY_T`. Om det är en konstant överst, så blir den lätt att hitta och ändra, om man vill justera fördröjningen. Vid felsökning blir det jobbigare att verifiera att `DELAY_T` fördröjer just 16000 klockcykler, eftersom dess beteende beror på något som finns någon annanstans i koden. I avsnitt 3.3, "Ökad läsbar kod", ser vi fler exempel på där det skiljer.

I ett studentprojekt över en period eller termin, så är det rimligt att flera personer ska utveckla samma kod, och dessutom hinner man glömma vad som står var, och vad syftet med olika rader är. I industrin är det viktigt att koden går att underhålla/utveckla, även av någon som kommer långt efter att du slutat. I båda fallen är felsökning genom kodgranskning centralt.

När du tänker ut hur programmet ska fungera så får du naturligt full koll på upplägg och vad som händer var. Läsaren saknar dock den kollen. Det måste du ha i bakhuvudet, och så gott det går guida läsaren, genom att vara pedagogisk när du väljer struktur, namn, kommentarer etc.

Tumregel: Skriv inte kod för att få datorn att göra det du vill. Skriv kod för att en människa ska kunna läsa den (bivillkor: datorn ska göra det du vill).

2 Modularisering

Ett litet program på säg 20 rader får plats på en skärm, och blir överblickbart även om det är ganska "trassligt" (trassel = programpekaren hoppar omkring huller om buller). Men när programmet växer, så är det ju bra att klumpa ihop instruktioner till en funktion eller modul. En sådan funktion kan då ha ett enkelt beroende mot övrig kod, och gömma undan ett litet internt trassel, som blir överblickbart just för att det är så litet. Jag ser det som att man stoppar ner koden i en låda, och skriver en lapp på lådan som summerar vad innehållet gör. En utvecklare behöver då bara läsa på lappen på lådan, för att få nödvändig överblick.

2.1 Gränssnitt

Vad som står på den lilla lappen kan beskrivas som funktionens gränssnitt. Detta bestämmer vad funktionen ska göra (men inte hur), och vilka bieffekter det får ha. Det är alltså en slags överenskommelse mellan funktionen och dess omgivning. Ett exempel på gränssnitt för funktionen `LCD_WRITE_NUM` kan vara att ta r16 som argument, samt att funktionen har rätt att förstöra innehållet i r16, r24 och r25, och förstås att skriva till LCD'n.

Det är ofta bra att låta alla funktioner ha så liknande gränssnitt som möjligt. På så sätt kan det gemensamma gränssnittet skrivas ner någon annanstans, så slipper man klottra ner källkoden med onödiga kommentarer. Exempel på gemensamt gränssnitt:

> *"Alla funktioner har rätt att förstöra innehållet i register r16-r19. För eventuella argument eller returvärden används r16."*

Huvuduppgift för varje enskild funktionen framgår ju av funktionsnamnet, och ytterligare information kan dokumenteras t.ex. som kommentaren `ADC_READ(r16=channel) -> r16=value.`

2.2 Funktioner

Låt oss påminna oss om vad en funktion egentligen är, i assembler/maskinkod. Vi har jump/branch för att hoppa omkring. När vi gör ett hopp, så kan vi spara en återhoppsadress på stacken genom instruktionen `call`, och sedan hoppa tillbaka med `ret`. Koden som körs kallar vi ju för funktion.

I programminnets flöde av instruktioner finns det dock inget formellt som säger att "här börjar en funktion". Det finns inget som hindrar oss från jmp till delar av en annan funktion, eller call till delar av sig själv. Det finns smarta lösningar som utnyttjar sådant, men det förstör den eleganta "paketeringen" som funktioner hjälper oss med. Definitionen är alltså egentligen ganska vag, och det är lätt att göra en ungefär-funktion. Priset för slarv är högt: Oläsbarhet och kanske en trasig stack, vilket är ett elände att felsöka. **Därför behövs disciplin**. Vi måste själva bestämma var en funktion börjar och var den slutar, och vi måste förlita oss på teknik som kommentarer, etiketter etc. för att det ska bli tydligt.

Många tycker att det här med funktioner är krångligt, och det kan de ha rätt i. Att dela in programmet i funktioner leder onekligen till att antalet rader ökar. Framför allt om funktionerna ska få enkla gränssnitt, så kan vissa jobb behöva göras dubbelt, t.ex. push:a/pop:a. Fördelarna är dock enorma, och väger upp nackdelarna många gånger om. Att dela upp koden i funktioner är ett fenomenalt bra sätt att packa ner instruktioner i lådor, enligt ovan, låta dem ha en tydligt avgränsad roll med enkelt överblickbara bieffekter. Får man till små och tydliga lappar på lådorna, så blir det enkelt att verifiera att innehållet gör vad lappen säger.

Kom ihåg att vi måste koda för att någon som inte kan läsa tankar ska förstå koden. Det blir snabbt ett hav av alla dessa små lappar. Utmaningen i planeringsstadiet är att komma på en lämplig uppdelning av projektet i funktioner, som minskar den totala mängden effekter och bieffekter att ha koll på, utan att varje funktion i sig blir oöverskådligt stor.

2.3 Moduler

Om man får oöverskådligt många funktioner, så kan man "packa ihop" dessa till moduler. Det gäller att packa ihop funktioner som har mycket med varandra att göra. En modul kan då ha ett antal *publika* funktioner och variabler, som kan användas från andra funktioner/moduler.

En funktion kan ju ha lokala etiketter för hopp inom funktionen, som *egentligen* är åtkomliga från andra funktioner, men vi låtsas in om dem. Vi kallar dem *privata*, och de står inte med på lilla lappen på lådan. På samma sätt kan en modul ha privata funktioner, som egentligen är publika, men vi skriver inte upp dem på modulens lapp.

Exempel på modul: Ta alla funktioner som används till att prata med en LCD-skärm och packa ihop i en modul, och kalla den för drivrutin. Den kan ha publika funktioner som `LCD_WRITE_NUM`, `LCD_CLEAR`, eller `LCD_SET_CURSOR`. Privata funktioner kan vara t.ex. `LCD_WRITE_BYTE`.

2.4 Filuppdelning

För att inte få en oöverskådligt stor fil, så kan man dela upp koden i flera filer. Jättemånga småfiler kan också vara svårhanterbart. Typiskt en modul per fil, och en "samlingsfil" för flera små moduler.

I `main.asm` så kan man skriva t.ex. `.include "i2c.inc"`. Vid kompilering ersätts då den raden av aktuell fil.

Det kan vara bra att överst i varje fil skriva ner "lapparna" på den/de moduler som finns i filen. Alltså en lista på alla publika funktioner och variabler i filen, med korta beskrivningar.

Så här skulle toppen på filen för PS2 kunna se ut:

```
; Reads PS/2 keyboard. Uses interrupt INT0.
; void PS2_INIT();
; void PS2_KEY_READY(); -> Z=1 if true
; r16  PS2_GET_KEY(); -> r16 = key code, not ascii
; r16  PS2_WAIT_FOR_KEY();
.include PS2_keycodes.inc ; Defines e.g. PS2_KEY_LShift

PS2__START:
.org INT0addr
   jmp  PS2__ISR
.org PS2__START

.dseg
.equ PS2_BUFF_SIZE = 16
PS2__buffer:   .byte PS2_BUFF_SIZE
PS2__shiftreg: .byte 1
.cseg
```

De översta raderna beskriver modulens gränssnitt, i form av kommentarer, följt av en inkludering av en fil innehållande definitioner av knappkoder.

Därefter kommer några spännande rader. `PS2__START` får den adress som modulens kod startar på. `.org INT0addr` säger att kommande rader ska hamna på avbrottsvektorn INT0addr. Där skrivs en jmp-instruktion till vad som verkar vara en privat avbrottsrutin. `.org PS2__START` så att säga "hoppar tillbaka" till där vi var i programminnet.

Slutligen kommer ett antal privata variabler i RAM-minnet.

3 Programupplägg och stilguide

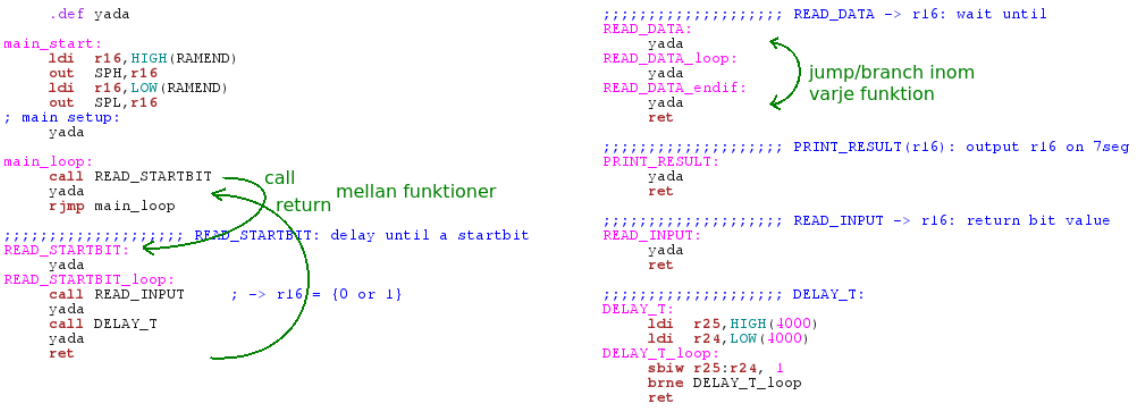
Den som inte är insatt i programmet, eller har glömt bort strukturen, har hjälp av olika tekniker som gör koden mer lättförståelig och läsbar. Även för felsökning/verifiering är läsbarheten viktig.

3.1 Visuell Hjälp

För att underlätta läsningen av program, bestäm dig för ett visst format på hur koden ser ut. Ögat ska ha lätt att hitta i koden, och det ska vara lätt att förstå vad saker gör eller betyder. Jag har funnit följande format hjälpsamt:

- Tydliga funktionsnamn är jätteviktigt.
- Ha tydliga visuella separatorer mellan funktioner, t.ex. en lång rad med semikolon. Se figur 1.
- Varje funktion har:
  - EN entry point överst (vars etikett är funktionsnamnet)
  - Kod i mitten
  - EN exit point längst ner.
- Mellan funktioner, kör med call/return.
- Inom varje funktion, kör med jmp/branch (breq, brne, rjmp etc). Organisera koden så att alla hopp är framåt (nedåt). Undantaget är loopar, som förstås måste hoppa tillbaka.

- Privata etiketter inom en funktion, heter samma sak som funktionen, följt av något talande, t.ex. `DECODE_KEY_elseif3`.



Figur 1: Exempel på utseende i assemblerkod

3.2 Registerstrategier

En stor del av gränssnittet mellan funktioner, handlar om register. I ett väldigt litet projekt, så kan man ha ett register till varje variabel, men det fungerar inte i större. Man måste lägga variabler i RAM-minnet, och använda register lokalt.

En vettig strategi:

- Låt r2 vara konstant = 0. Det är ibland bra att ha ett register som man vet är noll. Använd `.def zeroreg = r2`.
- Låt register r16 – r19, samt Z-pekaren, vara "temporära register". Alla funktioner har rätt att förstöra innehållet i dem vid anrop. Det är anroparens ansvar att push:a/pop:a dessa, om anroparen vill behålla värdena.
- Låt register r20 – r29 (inkl X- och Y-pekarna) "ägas uppifrån". En funktion som vill använda dessa, måste pusha/pop:a dem, som vi lärt oss, för att inte förstöra för anroparen. I toppen äger main-loopen dessa.
- Använd i första hand r18 som argument (både in i funktioner, och för att returnera värden). Behövs ett argument till, så används r19. Z-pekaren används när argumentet är en pekare (d.v.s. en adress).
- Avbrottsrutiner har ansvar för att återställa *alla* använda register. Detta då den avbrutna funktionen rimligen inte kan ta ansvaret att push:a/pop:a något.

En sammanfattning av de register som AVR:en har:

r0 – r1 Resultatet av multiplikation.

r0 – r15 Stöds ej av vissa instruktioner, t.ex. `ldi`.

r24 – r31 Utökat stöd för vissa 16-bitars-operationer, t.ex. `adiw`.

x = r27:r26 Kan adressera SRAM.

y = r29:r28 Kan adressera SRAM, inkl. disposition.

z = r31:r30 Kan adressera SRAM, PROM och användas som funktionspekare.

3.3 Ökad läsbar kod

Det finns massa sätt att formulera och skriva sin assembler-kod, som har för- och nackdelar.

3.3.1 .def – registernamngivning (rekommenderas INTE)

Man kan definiera nya namn på register, genom t.ex. `.def key=r16`. Det är nästan aldrig bra. Om man använder `key` genom hela programmet, så kan en liten patch använda sig av r16, eftersom det registret ser oanvänt ut. Vid stegning så kan man se vad r16 etc har för värde, och om man då har massa namngivna register, så måste man "avkoda" namngivningen manuellt för att se vad t.ex. `key` har för värde. Exemplet med `.def zeroreg=r2` kan vara okej.

3.3.2 .equ/.set – namngivna konstanter (rekommenderas ofta)

Att sätta namn på konstanter, t.ex. `.set SDA_pin=PC4`, gör koden betydligt mer läsbar. Man förstår i mycket högre grad vad koden gör.

Dock ger det ett beroende av definitionen (platsen i koden där .set står). Om det är långt bort, så kan det bli svårt att verifiera att koden gör rätt.

Notera att `.equ` sätter en konstant "för alltid", medan `.set` sätter ett värde som kan ändras lite senare i koden.

Ett exempel på där .set förtydligar koden:

```
.set loop_counter=14326
ldi  r25,HIGH(loop_counter)
ldi  r24,LOW(loop_counter)
```

Namnet `loop_counter` hjälper läsaren att förstå att 14326 är ett antal varv. Om vi vill ändra värdet, så behöver vi bara göra det på ett ställe. Vi kan återanvända namnet loop_counter på fler ställen. När det står precis ovanför, så kan kodläsaren se värdet direkt, vilket är en fördel.

3.3.3 Makron (rekommenderas oftast INTE)

Assemblermakron kan vara ett helt kapitel för sig, men jag ska hålla mig kort.

En väldigt vanlig kombination av instruktioner är

```
ldi         r16,VALUE
out         ADDR,r16
```

Dessa två *är* effektivaste sätter att lägga konstanten `VALUE` på adress `ADDR`. Likväl gör det koden lite klottrig. Vi kan dock slå ihop dessa i ett *makro*, och kallar det för t.ex. `outi`:

```
.macro outi       ; outi ADDR,VALUE
  ldi  r16,@1
  out  @0,r16
.endmacro
```

Så här kan det användas:

```
outi SPH,HIGH(RAMEND)
outi SPL,LOW(RAMEND)
outi DDRD,(1<<LED_pin)
```

Vid kompilering så kommer makrona att ersättas ("*expanderas*") till vad de egentligen är:

```
ldi  r16,HIGH(RAMEND)
out  SPH,r16
ldi  r16,LOW(RAMEND)
out  SPL,r16
ldi  r16,(1<<LED_pin)
out  DDRD,r16
```

Visst blev det lite "renare" kod när vi använde `outi`?

Dock finns det väldigt mycket faror med makron, som gör det svårmotiverat:

1. De ser ut som vanliga instruktioner, men kan ha bieffekter som att det påverkar r16.
2. De försvårar felsökning etc. När man stegar, så går det inte att stega in i makrot.

3. Den som läser koden måste vara väl införstådd med vilka makron som finns och i princip kunna dem utantill (i.a.f. vid felsökning). Varje makro blir alltså en slags belastning, och kräver ganska många anrop för att det ska vägas upp. Om du använder makron, förbered dig på strul när du ska förklara koden för en handledare eller programmeringspartner.

4. Felmeddelanden kan bli ännu knepigare att förstå.

5. Makron är *riktigt* luriga tillsammans med skip-instruktionerna – Betänk att en aktiv skip bara hoppar över första instruktionen i makrot.

I enstaka fall kan dock ett makron vara motiverat, men då bör det ha ett övertydligt namn. Exempel på sådana makron (utöver outi, ovan):

```
.macro ldi_pointer ; e.g. ldi_pointer z, MESSAGE<<1
  ldi  @0h, HIGH(@1)
  ldi  @0l, LOW(@1)
.endmacro
```

(denna sätter `zh` och `zl`, eller dito för x eller y).

```
.macro loop_until_IO_bit_set ; Usage: loop_until_IO_bit_set PIND,STROBE_pin
xx: sbis @0,@1
    rjmp xx
.endmacro
```

(Etiketten `xx` kommer att ersättas med något unikt varje gång `loop_until_IO_bit_set` expanderas).

3.3.4 Literaler

Att skriva literaler (konstanter) kan göras på flera sätt. Decimalt, hexadecimalt, ASCII (`'a'`), som t.ex. `(1<<ADSC)` eller något annat. Använd det sätt som bäst berättar vad det är för värde och vad det används till. Ska du skriva ut ett "X" på en LCD, så använd `'X'` istället för `$58`. Ska du räkna exakt 100 varv, så säger troligen 100 mer än $64. Att r16='0' säger, utöver $30, även att r16 representerar ett tecken.