Datorteknik
Morselabben i C
En kodstudie

—

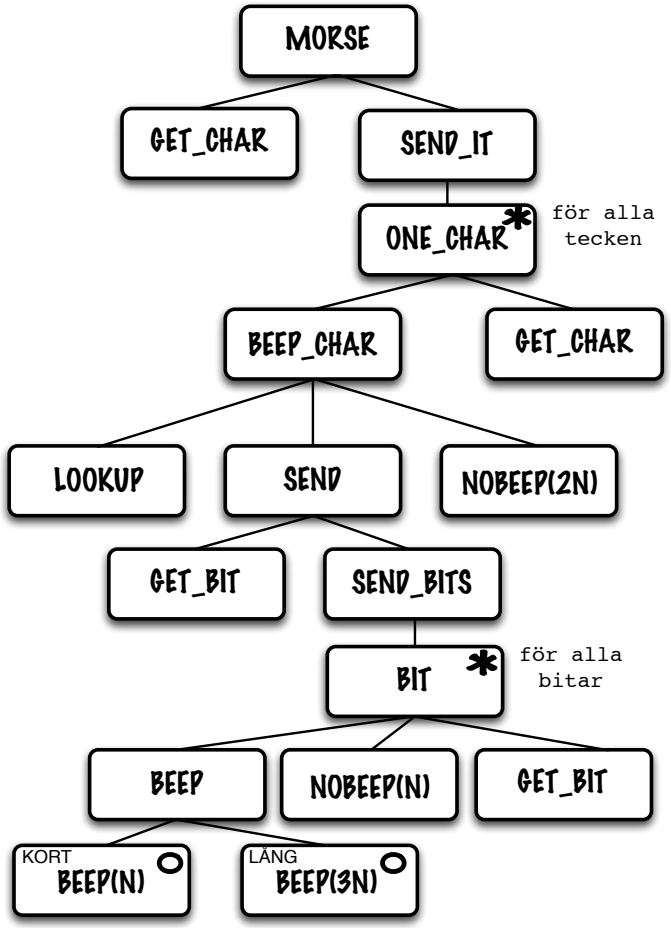
Michael Josefsson

Version 0.1 2018

Inledning Utan att avslöja för mycket om hur morselaborationen (Lab2) utförs i assembler kan vi betrakta dess tämligen strukturerade implementation i C. I detta dokument kommer vi titta närmare på hur C-kod transformeras till AVR-assembler av en kompilator.

Du bör vara förtrogen med laborationens lösning i assembler för att få fullt utbyte av denna text.

Labhäftet anger JSP-strukturdiagram och pseudokod med dessa bilder:




```
MORSE{
      CALL GET_CHAR;
      while( char != 0){
         CALL LOOKUP;
         CALL SEND;
         CALL NOBEEP(2N);
         CALL GET_CHAR;
      }
}
```

```
SEND{
      CALL GET_BIT;
      while ( Z != 0){
           if bit==0
                CALL BEEP(N);
           else
                CALL BEEP(3N);
           CALL NOBEEP(N);
           CALL GET_BIT;
      }
}
```

Utifrån denna initiala programbeskrivning kan kod skrivas för vilket programspråk som helst. I Lab2 översätts underlaget exempelvis till AVR-assemblerkod men i och med att problemet är nedbrutet till pseudokod är även andra programspråk möjliga.

I strukturdiagrammet är uppgiften uppdelad i åtskilliga små rutiner. Dessa rutiner utför uppgifter som måste finnas för ett komplett program. Typiskt motsvarar en ruta i strukturdiagrammet en subrutin i assembler.

Att slaviskt konstruera programmet enligt strukturdiagrammet ger dock en hel massa subrutiner. Rutiner som kan vara svåra att hålla reda och, inte minst, hitta bra namn till. I pseudokoden ovan till höger har JSP-strukturdiagrammet översatts till större enheter och därmed fått en kanske mer hanterbar uppdelning centrerad kring var sin uppslagning i en tabell: `MORSE` letar upp nästa tecken som skall sändas och `SEND` letar upp nästa teckendel och sänder dessa.

Andra uppdelningar av diagrammet finns naturligtvis. En första implementering i C är denna:

```
#include <avr/io.h>
#define SPEED 40
#define PITCH 50
#define DAH   3*SPEED
#define DIT   1*SPEED
#define DIT2  2*SPEED
#define DIT4  4*SPEED

const char *msg="DATORTEKNIK";
const char btab[]={ // a, b, c, ...
  0x60, 0x88, 0xa8, 0x90, 0x40,
  0x38, 0xd0, 0x08, 0x20, 0x78,
  0xb0, 0x48, 0xe0, 0xa0, 0xf0,
  0x68, 0xd8, 0x50, 0x10, 0xc0,
  0x30, 0x18, 0x70, 0x98, 0xb8,
  0xc8};

void  beep(int len,  char mask){
volatile
int i;

  while(len--){
    PORTA = mask;
    i = PITCH;  while(i--);
    PORTA = 0x00;
    i = PITCH;  while(i--);
  }
}
```

```
void  send(char binCode){
unsigned
int i = (unsigned int)binCode;

  i = i << 1;
  while(i & 0x00FF){
    i & 0x0100 ? beep(DAH,1) : beep(DIT,1);
    beep(DIT,0);
    i = i << 1;
  }
}


int main(void){
char  key;
int p;

  DDRA = 0xff;
  while(1){
    p=0;
    while(key=msg[p++]){
      send(btab[key -'A']);
      beep(DIT2,0);
    }
    beep(DIT4,0);
  }
}
```

Ett avsteg från, den i labhäftet rekommenderade, binärkodningen av morsetecknet har här måst göras då statusregistrets `C`-flagga inte är tillgänglig i C. Förändringen är sådan att binärkodningen temporärt är ett 16-bitars tal:

```
unsigned
int i = (unsigned int)binCode;
```

Teckendelslängden bestäms dock fortfarande av den utskiftade biten:

```
while(i & 0x00FF){
    i & 0x0100 ? beep(DAH,1) : beep(DIT,1);
  :
  :
```

I övrigt stämmer implementation väl med den i assembler.

Varför C? Om man nu måste våldföra sig på den eleganta (?) assemblerkoden i laborationen genom att konstruera om programmet, varför då ens skriva den i C?

Det är en berättigad fråga som kanske främst har två svar:

- Det blir mer läsbar och lättbegriplig kod i C. Koden blir samtidigt lättare att underhålla och felsöka.
- I ett högninvåspråk är ändringar och prov relativt snabba att göra. I assembler skulle man dra sig för att göra en så enkel sak som att byta *byte* till *word* då det sannolikt för med sig ändringar nedströms i koden. Ändringar som man själv är ansvarig för. Kompilatorn genomför flera av dessa automatiskt.

I detta dokument finns det ytterligare en anledning: Att titta på C-koden som kompilatorn ger ifrån sig! Det är inte en meningslös övning utan ger tvärtemot en inblick i vad kompilatorn har för sig och kan också ge tips om hur vi skall respektive *inte* skall skriva kod i C.

Förutsättningen för den presenterade koden är här hela tiden att vara så liten som rimligt är möjligt. Programminnet i mikrokontrollern är en dyr och begränsad resurs varför så lite som möjligt av den alltid skall användas.

Den kompilerade koden ovan blir följande ganska omfattande assemblerfil:[1]

```
000  JMP 0x02A      Jump
002  JMP 0x03F      Jump
004  JMP 0x03F      Jump
006  JMP 0x03F      Jump
008  JMP 0x03F      Jump
00A  JMP 0x03F      Jump
00C  JMP 0x03F      Jump
00E  JMP 0x03F      Jump
010  JMP 0x03F      Jump
012  JMP 0x03F      Jump
014  JMP 0x03F      Jump
016  JMP 0x03F      Jump
018  JMP 0x03F      Jump
01A  JMP 0x03F      Jump
01C  JMP 0x03F      Jump
01E  JMP 0x03F      Jump
020  JMP 0x03F      Jump
022  JMP 0x03F      Jump
024  JMP 0x03F      Jump
026  JMP 0x03F      Jump
028  JMP 0x03F      Jump
02A  CLR R1         Clear Register
02B  OUT 0x3F,R1    Out to I/O location
02C  LDI R28,0x5F   Load immediate
02D  LDI R29,0x04   Load immediate
02E  OUT 0x3E,R29   Out to I/O location
02F  OUT 0x3D,R28   Out to I/O location
030  LDI R17,0x00   Load immediate
031  LDI R26,0x60   Load immediate
032  LDI R27,0x00   Load immediate
033  LDI R30,0xEA   Load immediate
034  LDI R31,0x01   Load immediate
035  RJMP PC+0x03   Relative jump
036  LPM R0,Z+
037  ST X+,R0       Store indirect
038  CPI R26,0x88   Compare
039  CPC R27,R17    Compare
03A  BRNE PC-0x04   Branch if not equal
03B  CALL 0x0BC     Call subroutine
03D  JMP 0x0F3      Jump
03F  JMP 0x000      Jump
```

```
    22: void  beep(int len,
char mask){
041  PUSH R28       Push
042  PUSH R29       Push
043  RCALL PC+0x01  Relative call
044  RCALL PC+0x01  Relative call
045  PUSH R1        Push
046  IN R28,0x3D    In from I/O location
047  IN R29,0x3E    In from I/O location
048  STD Y+4,R25    Store indirect
049  STD Y+3,R24    Store indirect
04A  STD Y+5,R22    Store indirect
    26:   while(len--){
04B  RJMP PC+0x26 Relative jump
    27:     PORTA = mask;
04C  LDI R24,0x3B   Load immediate
04D  LDI R25,0x00   Load immediate
04E  LDD R18,Y+5    Load indirect
04F  MOVW R30,R24   Copy register pair
050  STD Z+0,R18    Store indirect
    28:     i = PITCH;  while(i--);
051  LDI R24,0x32   Load immediate
052  LDI R25,0x00   Load immediate
053  STD Y+2,R25    Store indirect
054  STD Y+1,R24    Store indirect
055  NOP    No operation
--- No source file
056  LDD R24,Y+1    Load indirect
057  LDD R25,Y+2    Load indirect
058  MOVW R18,R24   Copy register pair
059  SUBI R18,0x01  Subtract immediate
05A  SBC R19,R1     Subtract
05B  STD Y+2,R19    Store indirect
05C  STD Y+1,R18    Store indirect
05D  SBIW R24,0x00  Subtract immediate word
05E  BRNE PC-0x08   Branch if not equal
    29:     PORTA = 0x00;
05F  LDI R24,0x3B   Load immediate
060  LDI R25,0x00   Load immediate
061  MOVW R30,R24   Copy register pair
062  STD Z+0,R1     Store indirect
```

[1] Med `gcc -S` kan den erhållas även i andra miljöer.

```
    30:     i = PITCH;  while(i--);
063  LDI R24,0x32    Load immediate
064  LDI R25,0x00    Load immediate
065  STD Y+2,R25     Store indirect
066  STD Y+1,R24     Store indirect
067  NOP     No operation
--- No source file
068  LDD R24,Y+1     Load indirect
069  LDD R25,Y+2     Load indirect
06A  MOVW R18,R24    Copy register pair
06B  SUBI R18,0x01   Subtract immediate
06C  SBC R19,R1      Subtract
06D  STD Y+2,R19     Store indirect
06E  STD Y+1,R18     Store indirect
06F  SBIW R24,0x00   Subtract immediate word
070  BRNE PC-0x08    Branch if not equal
071  LDD R24,Y+3     Load indirect
072  LDD R25,Y+4     Load indirect
073  MOVW R18,R24    Copy register pair
074  SUBI R18,0x01   Subtract immediate
075  SBC R19,R1      Subtract
076  STD Y+4,R19     Store indirect
077  STD Y+3,R18     Store indirect
078  SBIW R24,0x00   Subtract immediate word
079  BRNE PC-0x2D    Branch if not equal
07A  POP R0          Pop register
07B  POP R0          Pop register
07C  POP R0          Pop register
07D  POP R0          Pop register
07E  POP R0          Pop register
07F  POP R29         Pop register
080  POP R28         Pop register
081  RET             Subroutine return
    34: void  send(char binCode){
082  PUSH R28        Push
083  PUSH R29        Push
084  RCALL PC+0x01   Relative call
085  PUSH R1         Push
086  IN R28,0x3D     In from I/O location
087  IN R29,0x3E     In from I/O location
088  STD Y+3,R24     Store indirect
    36: int i = (unsigned int)binCode;
089  LDD R24,Y+3     Load indirect
08A  MOV R24,R24     Copy register
08B  LDI R25,0x00    Load immediate
08C  STD Y+2,R25     Store indirect
08D  STD Y+1,R24     Store indirect
    38:   i = i << 1;
08E  LDD R24,Y+1     Load indirect
08F  LDD R25,Y+2     Load indirect
090  LSL R24         Logical Shift Left
091  ROL R25         Rotate Left Through Carry
092  STD Y+2,R25     Store indirect
093  STD Y+1,R24     Store indirect
    39:   while(i & 0x00FF){
094  RJMP PC+0x1D    Relative jump
    40:     i & 0x0100 ? beep(DAH,1) : beep(DIT,1);
095  LDD R24,Y+1     Load indirect
096  LDD R25,Y+2     Load indirect
097  CLR R24         Clear Register
098  ANDI R25,0x01   Logical AND
099  SBIW R24,0x00   Subtract immediate
09A  BREQ PC+0x07    Branch if equal
--- No source file
09B  LDI R22,0x01    Load immediate
09C  LDI R24,0x78    Load immediate
09D  LDI R25,0x00    Load immediate
09E  CALL 0x041      Call subroutine
0A0  RJMP PC+0x06    Relative jump
0A1  LDI R22,0x01    Load immediate
0A2  LDI R24,0x28    Load immediate
0A3  LDI R25,0x00    Load immediate
0A4  CALL 0x041      Call subroutine
0A6  LDI R22,0x00    Load immediate
0A7  LDI R24,0x28    Load immediate
0A8  LDI R25,0x00    Load immediate
0A9  CALL 0x041      Call subroutine
0AB  LDD R24,Y+1     Load indirect
0AC  LDD R25,Y+2     Load indirect
0AD  LSL R24         Logical Shift Left
0AE  ROL R25         Rotate Left Through Carry
0AF  STD Y+2,R25     Store indirect
0B0  STD Y+1,R24     Store indirect
0B1  LDD R24,Y+1     Load indirect
0B2  LDD R25,Y+2     Load indirect
0B3  CLR R25         Clear Register
0B4  SBIW R24,0x00   Subtract immediate
0B5  BRNE PC-0x20    Branch if not equal
0B6  POP R0          Pop register
0B7  POP R0          Pop register
0B8  POP R0          Pop register
0B9  POP R29         Pop register
0BA  POP R28         Pop register
0BB  RET             Subroutine return
    47: {
0BC  PUSH R28        Push
0BD  PUSH R29        Push
0BE  RCALL PC+0x01   Relative call
0BF  PUSH R1         Push
0C0  IN R28,0x3D     In from I/O location
0C1  IN R29,0x3E     In from I/O location
    51:   DDRA = 0xff;
0C2  LDI R24,0x3A    Load immediate
0C3  LDI R25,0x00    Load immediate
0C4  SER R18         Set Register
0C5  MOVW R30,R24    Copy register pair
0C6  STD Z+0,R18     Store indirect
    53:     p=0;
0C7  STD Y+2,R1      Store indirect
0C8  STD Y+1,R1      Store indirect
    54:     while(key=msg[p++]){
0C9  RJMP PC+0x11 Relative jump
    55:       send(btab[key -'A']);
0CA  LDD R24,Y+3     Load indirect
0CB  MOV R24,R24     Copy register
0CC  LDI R25,0x00    Load immediate
0CD  SUBI R24,0x41   Subtract immediate
0CE  SBC R25,R1      Subtract
0CF  SUBI R24,0x92   Subtract immediate
0D0  SBCI R25,0xFF   Subtract immediate
0D1  MOVW R30,R24    Copy register pair
0D2  LDD R24,Z+0     Load indirect
0D3  CALL 0x082      Call subroutine
    56:       beep(DIT2,0);
0D5  LDI R22,0x00    Load immediate
0D6  LDI R24,0x50    Load immediate
0D7  LDI R25,0x00    Load immediate
0D8  CALL 0x041      Call subroutine
    54:     while(key=msg[p++]){
0DA  LDS R18,0x60 Load direct from
0DC  LDS R19,0x61 Load direct from
0DE  LDD R24,Y+1     Load indirect
0DF  LDD R25,Y+2     Load indirect
0E0  MOVW R20,R24    Copy register pair
0E1  SUBI R20,0xFF   Subtract immediate
0E2  SBCI R21,0xFF   Subtract immediate
0E3  STD Y+2,R21     Store indirect
0E4  STD Y+1,R20     Store indirect
0E5  ADD R24,R18     Add
0E6  ADC R25,R19     Add
0E7  MOVW R30,R24    Copy register pair
0E8  LDD R24,Z+0     Load indirect
0E9  STD Y+3,R24     Store indirect
0EA  LDD R24,Y+3     Load indirect
0EB  TST R24         Test
0EC  BRNE PC-0x22    Branch if not equal
```

```
    58:     beep(DIT4,0);
0ED  LDI R22,0x00   Load immediate
0EE  LDI R24,0xA0   Load immediate
0EF  LDI R25,0x00   Load immediate
0F0  CALL 0x041     Call subroutine
    59:   }
0F3  CLI            Global Interrupt Disable
0F4  RJMP PC-0x00   Relative jump
```

Vi ser C-kodsrader insprängda i assemblerkoden vilket underlättar. Men vi ser också en hel del rader som saknar motsvarighet i C-koden: *No source file* anger assemblerkod som kan vara utvärdering av deluttryck eller intern *house keeping*. Man kan notera att kompilatorn inte tar hänsyn till om avbrott använts eller inte utan förutsätter att den måste lämna plats för avbrottsvektorerna:

```
000  JMP 0x02A      <<<<---Reset Vector
002  JMP 0x03F      Jump
004  JMP 0x03F      Jump
006  JMP 0x03F      Jump
008  JMP 0x03F      Jump
00A  JMP 0x03F      Jump
00C  JMP 0x03F      Jump
00E  JMP 0x03F      Jump
010  JMP 0x03F      Jump
012  JMP 0x03F      Jump
014  JMP 0x03F      Jump
016  JMP 0x03F      Jump
018  JMP 0x03F      Jump
01A  JMP 0x03F      Jump
01C  JMP 0x03F      Jump
01E  JMP 0x03F      Jump
020  JMP 0x03F      Jump
022  JMP 0x03F      Jump
024  JMP 0x03F      Jump
026  JMP 0x03F      Jump
028  JMP 0x03F      Jump
02A  CLR R1         Clear Register
```

Här ser vi att programstart på rad `$000` genomför ett hopp över samtliga avbrottsvektorer till C-programmets initiering, vanligen kallad `__init`. Det kan vara intressant att se vart hoppet sker, vad finns på `0x3F`?

C-miljön initieras genom att stacken sätts och sedan hanteras konstantsträngarna ”`DATORTEKNIK`” och ”`btab[]`” genom att de läggs över till SRAM[2]

```
02A  CLR R1         Clear R1 and SREG
02B  OUT 0x3F,R1    *
02C  LDI R28,0x5F   SP = RAMEND
02D  LDI R29,0x04   *
02E  OUT 0x3E,R29   *
02F  OUT 0x3D,R28   *
030  LDI R17,0x00   Load immediate
031  LDI R26,0x60   X = SRAMstart
032  LDI R27,0x00   *
033  LDI R30,0xEA   Z = datasegment
034  LDI R31,0x01   *
035  RJMP PC+0x03   Copy data to SRAM
036  LPM R0,Z+      *
037  ST X+,R0       *
038  CPI R26,0x88   *
039  CPC R27,R17    *
03A  BRNE PC-0x04   Branch if not equal
03B  CALL 0x0BC     Call subroutine
03D  JMP 0x0F3      Jump
03F  JMP 0x000      Jump
```

C-miljön kräver tydligen ett programslut på rad `$0F3`, normalt kallad `__exit:`.

```
0F3  CLI            Global Interrupt Disable
0F4  RJMP PC-0x00   Relative jump
```

Innehållet efter `$0F4` är datasegmentet som innehåller `*msg` och `btab[]`.

[2] Varför i Herrans namn gör den så? Läsning från FLASH är lika snabb som läsning från SRAM och `const`-deklarationen betyder att innehållet inte kan eller får röras. Dessa strängar borde lagts i FLASH, nu tar de upp värdefull plats i SRAM...

C-programets `int main()`-rutin återfinns på adress `$0BC`:

```
0BC  PUSH R28        Save Y
0BD  PUSH R29        *
0BE  RCALL PC+0x01   Relative call
0BF  PUSH R1         Push
0C0  IN R28,0x3D     Y points to SP
0C1  IN R29,0x3E     *
```

`main()` inleds med att sätta en egen *stack frame* genom att spara `Y` på stacken och sedan göra ett subrutinanrop till resten av `main()` som inleds med att spara `R1` på stacken och sedan sätta `Y` till nuvarande stackpekare `SPH:SPL`. För C-miljöns del pekar alltså `Y` *just nu* på samma plats i minnet som stackpekaren.

Anledningen till att bevara `R1` är att kompilatorn använder detta register till att *alltid* innehålla värdet noll.

Med detta avklarat börjar vi känna igen vår C-kod. Rad 51: `DDRA = 0xff;` översätts till exempel med följande assemblerrader:

```
 51:  DDRA = 0xff;
0C2  LDI R24,0x3A    R25:R24 = adr DDRA
0C3  LDI R25,0x00    *
0C4  SER R18         R18 = 0xff
0C5  MOVW R30,R24    Z = adr DDRA
0C6  STD Z+0,R18     DDRA = R18
```

Åtkomst av minne sker ofta med pekare och denna rutin använder `Z` för att slutligen sätta alla bitar i `DDRA` till ett.

Jämfört med hur man skulle gjort i assembler:

```
ser   r18
out   DDRA,r18
```

är kompilatorn omständlig och verkar inte känna till `in`/`out`-instruktionen!

Det lämnas som en intressant övning åt läsaren att identifiera övriga C-kodsrader och deras assemblermotsvarighet. Man noterar efter en stund att kompilatorn har ett antal standardförfaranden vid översättning till assembler.

Optimering Hittills har vi inte sett hela sanningen! Kompilatorn kan göra bättre ifrån sig med olika optimeringsalternativ. Koden hittills är helt utan optimering. I `gcc`-kompilatorn finns ytterligare optimeringar att välja:

- `-O0` ingen optimering
- `-O1` standardoptimering
- `-O2` mer aggressiv optimering
- `-O3` optimera för hastighet
- `-Os` optimera för kodstorlek

Det brukar vara störst skillnad, i både kodstorlek och hastighet, mellan `-O0` och `-O1` medan ytterligare optimering (`-O2`) sällan ger väsentlig skillnad. Vid optimering för hastighet `-O3` blir koden i allmänhet större eller mycket större eftersom loopar rullas ut till större kodstycken bland annat.

Optimeringsnivån `-Os` är den enda rimliga med vårt mål att få så mycket funktionalitet i så lite programminne som möjligt. C-miljöns initiering kommer vi inte undan (fortfarande tas hopptabellen för avbrott i anspråk till exempel). Resulterande assemblerkod vid `-Os` blir:

```
02A  CLR R1          Clear Register
02B  OUT 0x3F,R1     Out to I/O location
02C  LDI R28,0x5F    Load immediate
02D  LDI R29,0x04    Load immediate
02E  OUT 0x3E,R29    Out to I/O location
02F  OUT 0x3D,R28    Out to I/O location
030  LDI R17,0x00    Load immediate
031  LDI R26,0x60    Load immediate
032  LDI R27,0x00    Load immediate
033  LDI R30,0x4A    Load immediate
034  LDI R31,0x01    Load immediate
035  RJMP PC+0x03    Relative jump
036  LPM R0,Z+       Load program memory
037  ST X+,R0        Store indirect
038  CPI R26,0x88    Compare
039  CPC R27,R17     Compare
03A  BRNE PC-0x04    Branch if not equal
03B  CALL 0x083      Call subroutine
03D  JMP 0x0A3       Jump
03F  JMP 0x000       Jump
    22: void  beep(int len,  char mask){
041  PUSH R28        Push register
042  PUSH R29        Push register
043  RCALL PC+0x01   Relative call
044  IN R28,0x3D     In from I/O location
045  IN R29,0x3E     In from I/O location
    28:     i = PITCH;  while(i--);
046  LDI R18,0x32    Load immediate
047  LDI R19,0x00    Load immediate
    26:   while(len--){
048  SBIW R24,0x01   Subtract immediate
049  BRCS PC+0x18    Branch if carry set
    27:     PORTA = mask;
04A  OUT 0x1B,R22    Out to I/O location
    28:     i = PITCH;  while(i--);
04B  STD Y+2,R19     Store indirect
04C  STD Y+1,R18     Store indirect
04D  LDD R20,Y+1     Load indirect
04E  LDD R21,Y+2     Load indirect
04F  MOVW R30,R20    Copy register pair
050  SBIW R30,0x01   Subtract immediate
051  STD Y+2,R31     Store indirect
052  STD Y+1,R30     Store indirect
053  OR R20,R21      Logical OR
054  BRNE PC-0x07    Branch if not equal
055  OUT 0x1B,R1     Out to I/O location
056  STD Y+2,R19     Store indirect
057  STD Y+1,R18     Store indirect
058  LDD R20,Y+1     Load indirect
059  LDD R21,Y+2     Load indirect
05A  MOVW R30,R20    Copy register pair
05B  SBIW R30,0x01   Subtract immediate
05C  STD Y+2,R31     Store indirect
05D  STD Y+1,R30     Store indirect
05E  OR R20,R21      Logical OR
05F  BRNE PC-0x07    Branch if not equal
060  RJMP PC-0x18    Relative jump
061  POP R0          Pop register
062  POP R0          Pop register
063  POP R29         Pop register
064  POP R28         Pop register
065  RET             Subroutine return
    34: void  send(char binCode){
    i = i << 1;
066  PUSH R28        Push register
067  PUSH R29        Push register
068  MOV R28,R24     R28 = i
069  LDI R29,0x00    i = i << 1;
06A  LSL R28         * 16 bits rotate
06B  ROL R29         *
06C  MOVW R24,R28    i = R28
    40:     i & 0x0100 ? beep(DAH,1) : beep(DIT,1);
06D  CLR R25         Clear Register
06E  OR R24,R25      i == 0?
06F  BREQ PC+0x11    Branch if equal
070  LDI R22,0x01    arg to beep = 1
071  SBRS R29,0      '?'
072  RJMP PC+0x04    goto 075
073  LDI R24,0x78    DAH
074  LDI R25,0x00    *
075  RJMP PC+0x03    goto 078
076  LDI R24,0x28    DIT
077  LDI R25,0x00    *
078  CALL 0x041      Call beep()
07A  LDI R22,0x00    arg to beep = 0
07B  LDI R24,0x28    DIT
07C  LDI R25,0x00    *
07D  CALL 0x041      Call beep()
07F  RJMP PC-0x15    Relative jump
080  POP R29         Pop register
081  POP R28         Pop register
082  RET             Subroutine return
}
    51:   DDRA = 0xff;
083  SER R24         Set Register
084  OUT 0x1A,R24    Out to I/O location
    53:     p=0;
085  LDI R28,0x00    Load immediate
086  LDI R29,0x00    Load immediate
    54:     while(key=msg[p++]){
087  LDS R30,0x60    Load direct
089  LDS R31,0x61    Load direct
08B  ADD R30,R28     Add
08C  ADC R31,R29     Add with carry
08D  LDD R30,Z+0     Load indirect
08E  TST R30         Test
08F  BREQ PC+0x0E    Branch if equal
090  ADIW R28,0x01   Add immediate to word
091  LDI R31,0x00    Load immediate
092  SUBI R30,0xDF   Subtract immediate
093  SBCI R31,0xFF   Subtract immediate
094  LDD R24,Z+0     Load indirect
095  CALL 0x066      send()
097  LDI R22,0x00    Load immediate
098  LDI R24,0x50    Load immediate
099  LDI R25,0x00    Load immediate
09A  CALL 0x041      Call subroutine
09C  RJMP PC-0x15    Relative jump
09D  LDI R22,0x00    Load immediate
09E  LDI R24,0xA0    Load immediate
09F  LDI R25,0x00    Load immediate
0A0  CALL 0x041      Call subroutine
0A2  RJMP PC-0x1D    Relative jump
0A3  CLI             Global Interrupt Disable
0A4  RJMP PC-0x00    Relative jump
0A5  ???     Memory out of bounds
0A6  LDD R6,Z+16     Load indirect
0A7  ???     Memory out of bounds
0A8  CPI R20,0x80    Compare
0A9  SBC R13,R0      Subtract
```

```
0AA  ANDI R18,0x80  Logical AND           0B2  RJMP PC-0x0747 Relative jump
0AB  SBCI R27,0x80  Subtract immediate    0B3  SBCI R20,0x14  Subtract immediate
0AC  LDD R14,Z+32   Load indirect         0B4  SBCI R21,0xF4  Subtract immediate
0AD  ORI R31,0x80   Logical OR            0B5  SUBI R21,0x42  Subtract immediate
0AE  SUBI R29,0x08  Subtract immediate    0B6  SBCI R20,0xB5  Subtract immediate
0AF  RJMP PC+0x11   Relative jump         0B7  SBCI R20,0x9E  Subtract immediate
0B0  SUB R3,R0      Subtract              0B8  ???    Memory out of bounds
0B1  CBI 0x0E,0     Clear bit             0B9  NOP            Undefined (0x0000)
```

Den här koden är värd att studera. Inledningsraderna är som tidigare bortsett från att konstantsträngarna har ny adress, så de har utelämnats, men från programstart på rad $083 är koden lite annorlunda:

```
    51:   DDRA = 0xff;                      091  LDI R31,0x00   Load immediate
083  SER R24         Set Register          092  SUBI R30,0xDF  Subtract immediate
084  OUT 0x1A,R24    Out to I/O location   093  SBCI R31,0xFF  Subtract immediate
    53:     p=0;                            094  LDD R24,Z+0    Load indirect
085  LDI R28,0x00    Load immediate        095  CALL 0x066     Call subroutine
086  LDI R29,0x00    Load immediate        097  LDI R22,0x00   Load immediate
    54:     while(key=msg[p++]){            098  LDI R24,0x50   Load immediate
087  LDS R30,0x60    Load direct           099  LDI R25,0x00   Load immediate
089  LDS R31,0x61    Load direct           09A  CALL 0x041     Call subroutine
08B  ADD R30,R28     Add                   09C  RJMP PC-0x15   Relative jump
08C  ADC R31,R29     Add                   09D  LDI R22,0x00   Load immediate
08D  LDD R30,Z+0     Load indirect         09E  LDI R24,0xA0   Load immediate
08E  TST R30         Test                  09F  LDI R25,0x00   Load immediate
08F  BREQ PC+0x0E    Branch if equal       0A0  CALL 0x041     Call subroutine
090  ADIW R28,0x01   Add immediate to word 0A2  RJMP PC-0x1D   Relative jump
```

Nu har kompilatorn plötsligt kommit ihåg `out`-instruktionen! Pekaradressen `p` sätts noll på ett mindre resurskrävande sätt än att använda pekare: även om det tar *lika lång tid* på denna processor behöver inte pekarregistren låsas upp för detta.

För C-raden `while(key=msg[p++])` noterar vi att kompilatorn varit trofast C-filen och givit oss en pekare (på grund av "`char *msg`") till strängen `DATORTEKNIK` (istället för adressen till strängens första byte 44='D'), så pekaren hämtas först med `lds`:

```
087  LDS R30,0x60    Z = pekare
089  LDS R31,0x61    *
08B  ADD R30,R28     Z = Z + p
08C  ADC R31,R29     *
08D  LDD R30,Z+0     ASCII in R30
08E  TST R30         while(...)
08F  BREQ PC+0x0E    *
```

Pekarvärdet, två bytes "`7c 00`", finns i datasegmentet i SRAM på adress $60:

```
0060  7c 00
0062  60 88 a8 90 40 38 d0 08 20 78 b0 48 e0 a0   $60 $88 $a8...
0070  f0 68 d8 50 10 c0 30 18 70 98 b8 c8

007C  44 41 54 4f 52 54 45 4b 4e 49 4b            D A T O...
```

Tabelluppslagningen och beräkningen `btab[key-'A']` görs som:

```
091  LDI R31,0x00    ZH = 0
092  SUBI R30,0xDF   key = key -  -$21
093  SBCI R31,0xFF
094  LDD R24,Z+0     btab-value
```

Subtraktionen med –$21 (=$-33_{10}$) oväntad då ASCII-`A`=$41 men får sin förklaring då `btab` börjar på adress $62 och $62-$41=$21. Så det är egentligen additionen `key = key + $21` som utförs.

När väl binärkoden är beräknad läggs den i `R24` och C-funktionen `send()`, här subrutinen på rad $66, anropas. I listningen finns några förklarande kommentarer i denna rutin och man ser att kompilatorn gjort ett omfattande jobb med den ingående koden.

Den genomskådar att det bara behövs ett enda vänsterskift och gör en 16-bits rotate då inargumentet av typen `char` (8 bitar) är castat till `insigned int` (16 bitar).

Kompilatorn är däremot tvungen att använda `Y=R29:R28` och därmed pusha dessa på stacken antingen de används i övrigt eller inte. Som assemblerprogrammerare skulle man kunna välja andra register då rutinen inte kräver att `Y` används som pekare. Kompilatorn genomskådar heller inte att `rcall` skulle kunnat användas överallt.

Det är emellertid en lätt extra kostnad i kod att bära för möjligheten att beskriva programmet i C. I alla fall försåvitt inte extrema krav finns på programmet i fråga om exekveringstid och storlek.

Minnesåtgång Ett omsorgsfullt byggt assemblerprogram med samma funktion drar cirka 120 bytes kod, tabellerna (om 40 bytes) undantagna då de inte är så mycket att göra någonting åt. Det *går* att komma ner till under 100 bytes kod men då på bekostnad av läsbarhet och framtida underhåll.

Den icke-optimerade versionen av C-koden förbrukar 490 bytes programminne för koden enbart och den storleksoptimerade (`-Os`) 370 bytes, en reducering med 25 procent!

Utan att förfalla till fulkod gjordes flera försök att minska C-programmets omfattning. Det handlade om omstruktureringar, ändra skrivsätt, ändra variabelstorlekar och antal osv. Allt i syfte att hitta en *sweet spot* där kompilatorn, ställd på `-Os`, kunde generera lite mindre kod. Det handlade alltså om att skriva kod på ett sådant sätt att kompilatorn fick ett lättare jobb att optimera koden. Slutresultatet blev denna kod:

```
#include <avr/io.h>
#define SPEED 40
#define PITCH 50
#define DIT   1*SPEED
#define DIT2  2*SPEED
#define DAH   3*SPEED
#define DIT4  4*SPEED

const char *msg="DATORTEKNIK";
const char btab[]={
  0x60, 0x88, 0xa8, 0x90, 0x40,
  0x38, 0xd0, 0x08, 0x20, 0x78,
  0xb0, 0x48, 0xe0, 0xa0, 0xf0,
  0x68, 0xd8, 0x50, 0x10, 0xc0,
  0x30, 0x18, 0x70, 0x98, 0xb8,
  0xc8};

void sbeep(int len,  char mask){
volatile
int i;
  while(len--){
    PORTA = mask;
    i = PITCH;  while(i--);
    PORTA = 0x00;
    i = PITCH;  while(i--);
  }
}

void  beep(int len,  char mask){
  sbeep(len, mask);
  len = DIT;
  mask = 0;
  sbeep(len, mask);
}

int main(void){
char  key, i, *s;

  DDRA = 0xff;
  while(1){
    s = msg;
    while(key=*s++){
      i =  btab[key -'A'];
      while(i != (i & 0x80)){
        i & 0x80 ? beep(DAH,1) : beep(DIT,1);
        i <<= 1;
      }
      beep(DIT,0);
    }
    beep(DIT2,0);
  }
}
.
```

Man kan se att övergång till byte-variabler har skett där så varit möjligt. Då mikrokontrollern redan är byte-orienterad internt borde det leda till kortare kod.

Den största förändringen är att `beep()` nu sönderfaller två rutiner: `beep()` och `sbeep()`. Denna uppdelning krympte koden så mycket att det ansågs värt att behålla den (även om `beep()` onekligen blev fulare!)

Slutresultatet, efter ganska många tröstlösa försök, bestod av 344 bytes kod. En förbättring med $1 - 344/490 = 30$ % jämfört med den ooptimerade originalkoden och blott $1 - 344/370 = 7$ % mindre än originalkoden optimerad för storlek. Man kan verkligen ifrågasätta om den nedlagda tiden motsvarar en förbättring i det senare fallet?

Hela assemblerkoden återfinns i appendix på nästa sida.

Not: Genom att deklarera textsträngen som

```
const char msg[]="DATORTEKNIK";
```

kunde ytterligare två bytes (pekarvariabeln) skalas bort från programmet och 342 blev den minsta, slutliga, programstorleken.

Appendix. Sista versionen kompilerad med `-Os`

```
02A  CLR R1          Clear Register
02B  OUT 0x3F,R1     Out to I/O location
02C  LDI R28,0x5F    Load immediate
02D  LDI R29,0x04    Load immediate
02E  OUT 0x3E,R29    Out to I/O location
02F  OUT 0x3D,R28    Out to I/O location
030  LDI R17,0x00    Load immediate
031  LDI R26,0x60    Load immediate
032  LDI R27,0x00    Load immediate
033  LDI R30,0x30    Load immediate
034  LDI R31,0x01    Load immediate
035  RJMP PC+0x03    Relative jump
036  LPM R0,Z+       Load program memory
037  ST X+,R0        Store indirect
038  CPI R26,0x88    Compare
039  CPC R27,R17     Compare
03A  BRNE PC-0x04    Branch if not equal
03B  CALL 0x06D      Call subroutine
03D  JMP 0x096       Jump
03F  JMP 0x000       Jump
    43: void sbeep(int len,  char mask){
041  PUSH R28        Push register
042  PUSH R29        Push register
043  RCALL PC+0x01   Relative call
044  IN R28,0x3D     In from I/O location
045  IN R29,0x3E     In from I/O location
    48:     i = PITCH;  while(i--);
046  LDI R18,0x32    Load immediate
047  LDI R19,0x00    Load immediate
    46:   while(len--){
048  SBIW R24,0x01   Subtract immediate
049  BRCS PC+0x18    Branch if carry set
    47:     PORTA = mask;
04A  OUT 0x1B,R22    Out to I/O location
    48:     i = PITCH;  while(i--);
04B  STD Y+2,R19     Store indirect
04C  STD Y+1,R18     Store indirect
04D  LDD R20,Y+1     Load indirect
04E  LDD R21,Y+2     Load indirect
04F  MOVW R30,R20    Copy register pair
050  SBIW R30,0x01   Subtract immediate
051  STD Y+2,R31     Store indirect
052  STD Y+1,R30     Store indirect
053  OR R20,R21      Logical OR
054  BRNE PC-0x07    Branch if not equal
055  OUT 0x1B,R1     Out to I/O location
056  STD Y+2,R19     Store indirect
057  STD Y+1,R18     Store indirect
058  LDD R20,Y+1     Load indirect
059  LDD R21,Y+2     Load indirect
05A  MOVW R30,R20    Copy register pair
05B  SBIW R30,0x01   Subtract immediate
05C  STD Y+2,R31     Store indirect
05D  STD Y+1,R30     Store indirect
05E  OR R20,R21      Logical OR
05F  BRNE PC-0x07    Branch if not equal
060  RJMP PC-0x18    Relative jump
061  POP R0          Pop register
062  POP R0          Pop register
063  POP R29         Pop register
064  POP R28         Pop register
065  RET     Subroutine return
066  CALL 0x041      Call subroutine
068  LDI R22,0x00    Load immediate
069  LDI R24,0x28    Load immediate
06A  LDI R25,0x00    Load immediate
06B  JMP 0x041       Jump
    27:   DDRA = 0xff;
06D  SER R24         Set Register
06E  OUT 0x1A,R24    Out to I/O location
06F  LDS R28,0x60    Load direct
071  LDS R29,0x61    Load direct
    30:     while(key=*s++){
073  LD R30,Y+       Load indirect
074  TST R30         Test
075  BREQ PC+0x1B    Branch if equal
    31:       i =  btab[key -'A'];
076  LDI R31,0x00    Load immediate
077  SUBI R30,0xDF   Subtract immediate
078  SBCI R31,0xFF   Subtract immediate
079  LDD R17,Z+0     Load indirect
    32:       while(i != (i & 0x80)){
07A  MOV R24,R17     Copy register
07B  ANDI R24,0x80   Logical AND
07C  CP R17,R24      Compare
07D  BREQ PC+0x0D    Branch if equal
    33:         (i & 0x80) ? beep(DAH,1) : beep(DIT,1);
07E  LDI R22,0x01    Load immediate
07F  SBRS R17,7      Skip if bit set
080  RJMP PC+0x04    Relative jump
081  LDI R24,0x78    Load immediate
082  LDI R25,0x00    Load immediate
083  RJMP PC+0x03    Relative jump
084  LDI R24,0x28    Load immediate
085  LDI R25,0x00    Load immediate
086  CALL 0x066      Call subroutine
088  LSL R17         Logical Shift Left
089  RJMP PC-0x0F    Relative jump
08A  LDI R22,0x00    Load immediate
08B  LDI R24,0x28    Load immediate
08C  LDI R25,0x00    Load immediate
08D  CALL 0x066      Call subroutine
08F  RJMP PC-0x1C    Relative jump
    38:     beep(DIT2,0);
090  LDI R22,0x00    Load immediate
091  LDI R24,0x50    Load immediate
092  LDI R25,0x00    Load immediate
093  CALL 0x066      Call subroutine
    39:   }
095  RJMP PC-0x26    Relative jump
096  CLI             Global Interrupt Disable
097  RJMP PC-0x00    Relative jump
098  ???     Memory out of bounds
099  LDD R6,Z+16     Load indirect
09A  ???     Memory out of bounds
09B  CPI R20,0x80    Compare
09C  SBC R13,R0      Subtract
09D  ANDI R18,0x80   Logical AND
09E  SBCI R27,0x80   Subtract immediate
09F  LDD R14,Z+32    Load indirect
0A0  ORI R31,0x80    Logical OR
0A1  SUBI R29,0x08   Subtract immediate
0A2  RJMP PC+0x11    Relative jump
0A3  SUB R3,R0       Subtract
0A4  CBI 0x0E,0      Clear bit
0A5  RJMP PC-0x0747  Relative jump
0A6  SBCI R20,0x14   Subtract immediate
0A7  SBCI R21,0xF4   Subtract immediate
0A8  SUBI R21,0x42   Subtract immediate
0A9  SBCI R20,0xB5   Subtract immediate
0AA  SBCI R20,0x9E   Subtract immediate
0AB  ???     Memory out of bounds
```

—o=Ö=o—