Örebro universitet
Institutionen för naturvetenskap och teknik
JPZ

DATORKOMMUNIKATION OCH NÄT

Laboration 5

Controller Area Network (CAN)

Laborationens förhållande till teorin

Laborationen **Controller Area Network** använder en protokollstack enligt fältbussmodellen och en typisk CAN-nod (CAN Controller, signalomvandlare och mikroprocessor).

Redovisning

Laborationen **Controller Area Network** redovisas med en skriftlig rapport. I denna rapport ska du beskriva laborationen, redovisa dina svar och programmen som har skrivits. Använd kursens rapportmall. Rapporten ska bestå av följande delar:

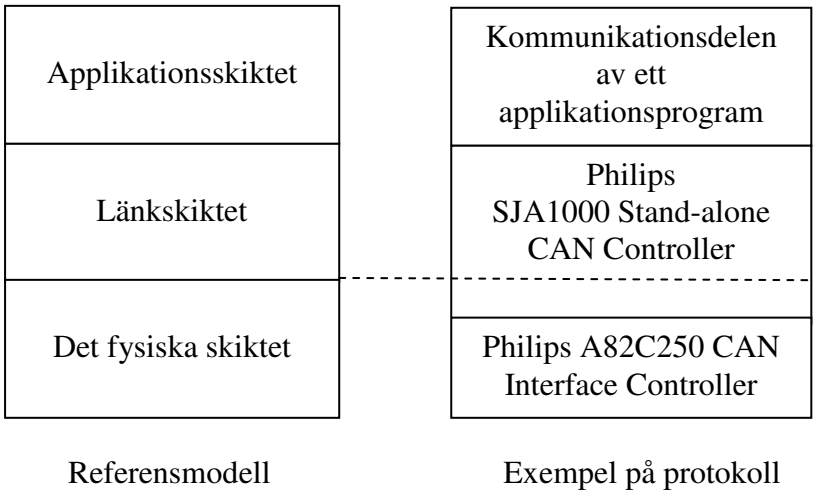
1. Titelsida
   Fyll i rubrik (laborationens titel) och personuppgifter. Gör en kort sammanfattning av laborationen.
2. Innehållsförteckning
   (Höger-klicka på innehållsförteckningen för uppdatera.)
3. Bakgrund
   Laborationens koppling till datorkommunikation
4. Resultat
   Uppgift A: beskrivning av uppgiften och skiss över systemet
   Uppgift B: beskrivning av uppgiften och kort om resultat
   Uppgift C: beskrivning av uppgiften och resultat
   Uppgift D: beskrivning av uppgiften och hänvisning till bilagorna
5. Bilagor
   1. Programkod för nod A
   2. Programkod för nod B
   3. Om flera noder har kopplats ihop, redovisa programkod för respektive.

Introduktion

Denna laboration är tänkt som en introduktion till **Controller Area Network** (CAN). Vi använder endast två noder per CAN-buss. (Endast som frivillig uppgift ska fler noder än två kopplas ihop.) Varje CAN-nod hanteras av en grupp på maximalt två studenter. För att koppla ihop två CAN-noder ska alltså två grupper samarbete. Varje grupp behöver också en dator för programutveckling.

De program som ska skrivas är delar av både applikationsprogram och applikationssprotokoll, enligt figur 1. Med applikationsprotokoll avser vi en beskrivning som realiseras i form av programvara. Applikationsprotokollet ska tillhandahålla tjänster för överföring av meddelanden på CAN-bussen. När data sänds över CAN-bussen eller data tas emot från CAN-bussen, är det allstå fråga om applikationsprotokollet. Allt som har med inmatning av data från tangentbord, utmatning av data på terminalfönster och bearbetning hör istället till vårt applikationsprogram.

Under denna laboration kommer vi att använda **Basic CAN**. Detta innebär att applikationsprotokollet också kommer att agera som acceptansfilter. (Det är acceptansfiltret som avgör vilka inkommande ramar som ska behandlas av CAN-noden.)



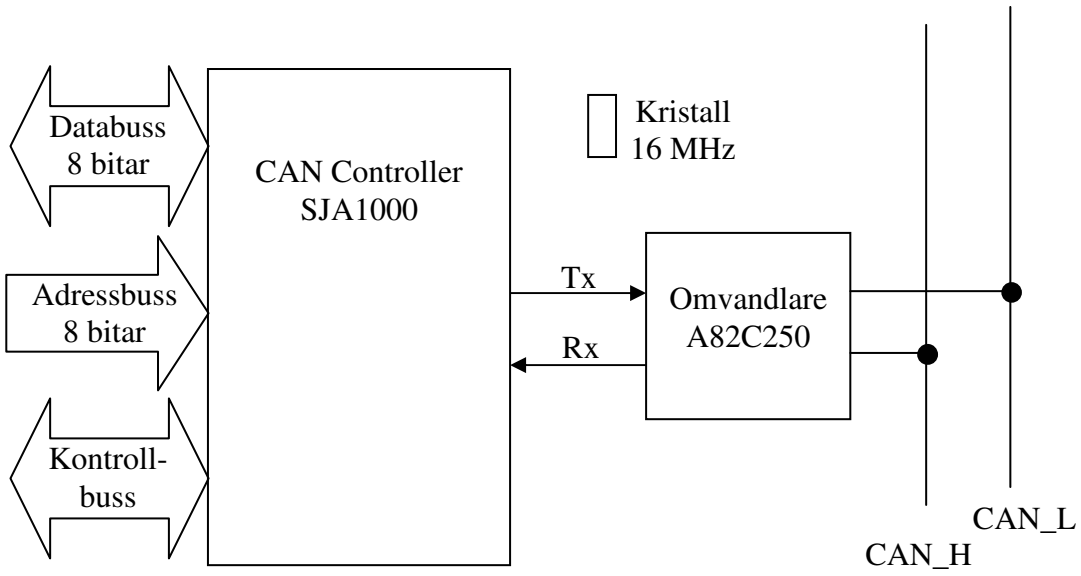
Figur 1. Referensmodell för fältbussar.

Hårdvara

CAN Controller

Grunden för en CAN-nod är att det finns en CAN-kontroller. I denna laboration använder vi en **SJA1000 Stand-alone CAN Controller** från Philips. Blockschemat för denna finns med som bilaga. Förutom SJA1000 behövs en omvandlare. Vi använder **A82C250 CAN Interface Controller** från Philips. Det är en sändare/mottagare (**CAN Transceiver**) som omvandlar signalerna mellan CAN-kontrollern och CAN-bussen. Dessutom behövs en kristall för att ge oscillatorn (klockan) en bestämd svängningsfrekvens. Vi använder en kristall med självsvängningsfrekvensen 8 MHz.

En CAN-kontroller består av en mängd register som används för att sända, ta emot och hålla kontroll. I denna laboration ska vi undersöka hur några av registren används.



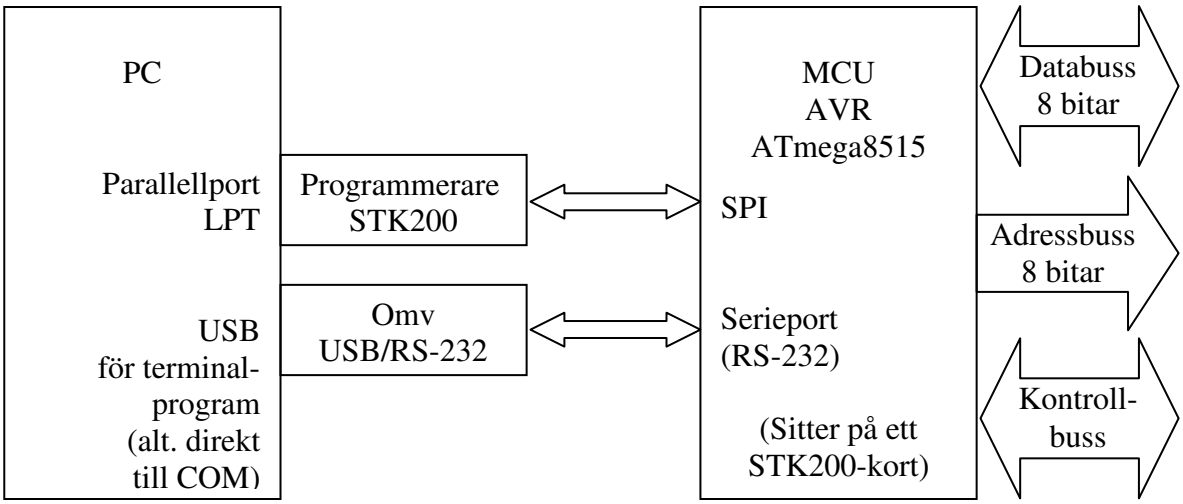
Figur 2. **CAN Controller**.

SJA1000 har två tillstånd (eng. *Modes*), **Basic CAN** och **PeliCAN**. Det förra innebär att det är mikroprocessorns uppgift att kontrollera om meddelanden på CAN-bussen ska tas om hand eller ej. I detta tillstånd används ramformatet **2.0A**, dvs. 11-bitars identifierare. **Full CAN** som i SJA1000 kallas **PeliCAN** innebär att ett acceptansfilter i CAN-kontrollern avgör om meddelanden ska behandlas eller ej. Detta frigör processorkraft så att mikroprocessorn kan ägna sig helt åt annat arbete. I **PeliCAN** används ramformatet **2.0B**, dvs. 29-bitars identifierare. Under denna laboration kommer vi endast att använda **Basic CAN**. Detta är redan inställt i programmet liksom att bithastigheten på CAN-bussen ska vara 125 kbps.

Mikroprocessor

För att få CAN-noden att inte bara sända och ta emot meddelanden, behövs en mer intelligent enhet, en mikroprocessor. Vi använder en **MicroController Unit** (MCU) av typen **AVR**

ATmega8515 från ATMEL. Det är en 8-bitars mikroprocessor i CMOS-teknik med RISC-arkitektur. Klockfrekvensen är vald till 8 MHz. (Maximalt klockfrekvens är 16 MHz.) Det är denna MCU som vi ska skriva program för. I den finns ett flashminne på 8 kB. Ett flashminne är ett snabbt **Electrical Erasable Programmable Read Only Memory** (EEPROM, E^2PROM). För att kunna programmera detta flashminne, finns ett **SPI Serial Interface for Program Download**. Vi kommer att använda parallellporten (LPT) på en PC för att sända program till flashminnet. För ändamålet krävs en speciell kabel med en s.k. programmerare. I detta fall en STK200. MCU är monterad på ett kort som också kallas STK200. Det finns också en traditionell serieport (RS-232) på STK200-kortet. Denna ansluter vi med hjälp av en adapter och kabel till någon av datorns USB-portar. (Givetvis kan RS-232 som alternativ anslutas direkt till en COM-port istället för att använda en USB-port .) Tanken är att programmet i MCU ska mata ut text på ett terminalfönster och mata in tecken från tangentbordet.



Figur 3. PC och MCU.

Uppgift A

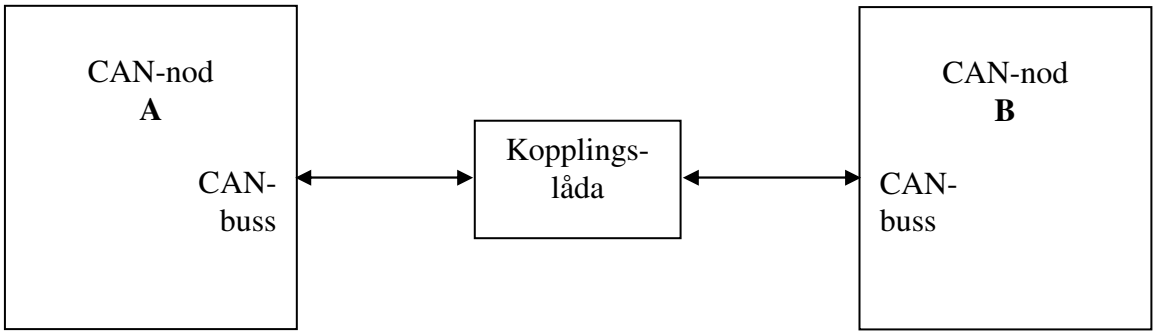
Denna laboration ska utföras i grupper om högst två studenter i varje. Två grupper ska samarbeta genom att koppla upp sina CAN-noder på samma buss.

1. Koppla ihop din PC med en MCU, enligt figur 3.

2. Koppla ihop MCU med SJA1000 som finns på ett specialtillverkat kort. Bandkablarna är märkta med **A** respektive **D**. Dessa ansluts till STK200-portarna som är märkta med **Port A** respektive **Port D**.

Koppla ihop de två CAN-noderna. Använd seriekablar med 9-poliga D-sub-kontakter och en kopplingslåda mellan dessa. Se figur 4 och figur 5. (Det är inte nödvändigt att använda just 9-poliga D-sub-kontakter för CAN-bussar, men för laborationsändamål är det praktiskt.)

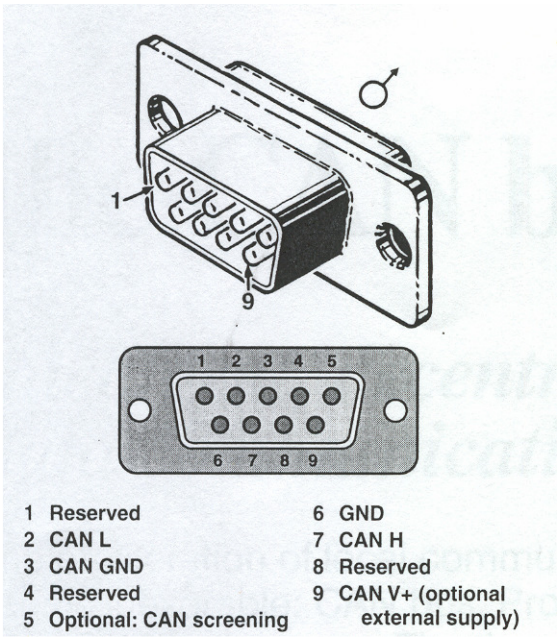
Observera att en av kopplingsdosans tre kontakter ovillkorligen måste vara ansluten till en PC för att få matningsspänning och signaljord. Denna kontakt är märkt med ”Power in”. I annat fall kommer uppgift C inte att fungera.



Figur 4. Sammankoppling av CAN-noderna.

3. Anslut en strömadapter till STK200-kortet och slå på strömmen.

4. Starta din PC och logga in.



Figur 5. 9-polig D-sub-kontakt för CAN-buss.

Program

Kermit

Terminalprogrammet **Kermit** ska användas för att skicka och ta emot meddelanden över CAN-bussen. Kermit startas och används från kommandofönstret **Terminal**. Starten görs på följande sätt:

```
~$ kermit
C-kermit> set line /dev/ttyUSB0
C-kermit> set carrier-watch off
C-kermit> set speed 9600
C-kermit> c
```

Om RS-232 på STK200-kortet kopplas direkt till PC:ns COM-port, ersätt **ttyUSB0** med **ttyS0**. (Beroende på vald port på PC:n ska **0** ersättas med **1**, eller annan siffra. Vid osäkerhet, fråga läraren.) Om allt är korrekt kopplat, visas från och med nu kommunikationen med programmet CANSJA som körs i MCU. Väl inne i Kermit följer man de instruktioner som programmet CANSJA ger. För att avbryta denna visning, tryck samtidigt på

[Ctrl]+[Alt Gr]+[?+\]

och sedan på

[C]

Att starta om från Kermitprompten görs som vanligt på följande sätt:

```
C-kermit> c
```

Kermit kan avslutas med följande:

```
C-kermit> exit
```

CANSJA

För att snabbt komma igång med laborationen finns det ett färdigt C-program som kallas CANSJA. Källkoden för detta finns i **cansja.c**. Filen finns listad i bilagorna.

Eclipse

Eclipce är en komplett programmeringsmiljö med projekthantering, GCC (kompilering och länkning), debugger och laddare (programuppladdning till flashminnet i MCU). För att GCC ska fungera tillverkar Eclipse själv en Makefile. Det finns att välja mellan flera olika programmerare (laddare) för olika typer av portar. Varje programmerare har sin inställning. Den som passar STK200 och kan användas i Ubuntu fungerar för närvarande endast för parallellporten. För att kunna hantera debugging direkt i hårdvaran krävs speciell typ av MCU och speciell typ av programmerargränssnitt, exempelvis **Joint Test Action Group** (JTAG).

Uppgift B

För att få igång på programmet CANSJA krävs att du först noga följer instruktionen.

1. Öppna det grafiska filhanteringssystemet Files (ikonen på skrivbordet) och gå till katalogen **Labbar** som finns i hemkatalogen. Kopiera arkivet **CAN-Lab.tar.gz** och klistra in i hemkatalogen.
2. Dekomprimera (höger-klicka > Extract Here) arkivet **CAN-Lab.tar.gz**.
3. Döp om **CAN-Lab** till **cansja**. (Namnet är viktigt för projekthanteringen i Eclipse).
4. Kopiera **cansja** till **workspace**. (Katalogen används för projekt av Eclipse.)
5. Starta Eclipse (ikonen på skrivbordet).
6. Skapa nytt projekt: File > New > **C project**
   Project Name: **cansja** (Namnet är viktigt för matchning mellan katalog och C-fil.)
   Välj: AVR Cross Target Application > **Empty Project**
   [Next >]
   Välj enbart **Release** (bocka för).
   [Advanced settings]
   Expandera AVR och välj **AVRDude**.
   Välj **STK200**.
   [Apply]
   Välj **Target Hardware**.
   [Load from MCU] (Ska ge **ATmega8515**.)
   Välj MCU Clock Frequency **8000000**.
   [Apply], [OK], [Next >]
   MCU Type: **ATmega8515**
   MCU Frequency (Hz): **8000000**
   [Finish]
7. Från Ecplise: Öppna upp katalogen **cansja** och sedan också filen **cansja.c**. (I detta läge kan programmet redigeras.)
8. Kompilera och länka: klicka på **hammaren** som finns på verktygsfältet.
   Öppna katalogen **Release** och kontrollera att den körbara filen **cansja.hex** har skapats.
   Observera också den nyskapade **makefile**.
   Ladda ner **cansja.hex** till MCU: klicka direkt på **AVR*** som finns på verktygsfältet.
9. Starta terminalprogrammet **Kermit**. Se avsnittet **Kermit** i detta häfte.
10. Gör reset genom att stänga av och sedan slå på strömmen med brytaren på STK200-kortet. När programmet CANSJA svarar, kontrollera att statusregistret visar **$0C_{hex}$**. Om inte, upprepa reset. (Programmet CANSJA visar några utvalda SJA1000-register i terminalens fönster efter reset.)

    Programmet CANSJA visar inte bara text i terminalfönstret utan också på LCD. I början av CANSJA skickas följande textsträng ut till den första raden på LCD:

    **** CAN NODE ****

11. Sedan ska du ange en Tx-identifierare (Tx = Transmitt) för din nod. Denna identifierare ska användas i de ramar som sänds. (I verkliga tillämpningar används inte endast en identifierare.) Identifieraren anges med tre tecken i ett hexadecimalt tal (000 – 7FF).

När du har angett en identifierare skickas ett meddelande ut på CAN-bussen av programmet CANSJA. Meddelandet lyder: **ON-LINE**. Eftersom CANSJA visar alla meddelanden på den andra LCD-raden, kommer den andra noden på bussen att visa t.ex.

**** CAN NODE ****
RX 0005 ON-LINE

På den andra LCD-raden står det exempelvis **RX 0005** som betyder att detta meddelande har Rx-identifieraren (Rx = Receive) lika med 005_{hex}. (Motparten sände med Tx-identifieraren 005_{hex} vilken togs emot av noden som en Rx-identifierare med samma värde.)

(Givetvis kan ingen av noderna visa text på den andra raden om CANSJA inte har kommit fram till möjligheten att sända/ta emot. Efter start av bägge noderna, starta om den första för att få se ”ON-LINE” även på den.)

Om du vill sända, tryck på [Enter] så att du får upp texten **Enter a message to send (maximum 8 characters)**. Skriv de tecken som du vill skicka och tryck på [Enter]. Om du skriver exempelvis **HEJSAN**, kommer programmet CANSJA att fylla ut med två asterisker, dvs. programmet sänder alltid åtta tecken, i detta fall **HEJSAN****.

Funktionen som hanterar inmatning från terminalen byter automatiskt från gemena tecken till versala.

Den nod som ska kunna ta emot ett meddelande, får inte vara i sändningsläge. Om programmet CANSJA väntar på åtta tecken att sända, tryck på [Enter] för att komma ut ur sändningsläget. (Då sänds ************.) Sedan kan noden ta emot nya meddelanden.

Provkör programmet CANSJA i bägge noderna.

Uppgift C

KORTSLUTNINGS-PROV	Pol: matningsspänning (5 V)	Pol: signaljord	Med varandra
CAN_L			
CAN_H			
Båda signalerna till samma pol			
Båda signalerna till olika poler	CAN_L till matningsspänning CAN_H till signaljord		
Båda signalerna till olika poler	CAN_L till signaljord CAN_H till matningsspänning		

Tabell 1. Kortslutningsprov.

Det påstås att CAN-bussen är så pass robust att den klarar kortslutningar. Gör kortslutningsprov med hjälp av kopplingsdosan. Kortslut signalerna CAN_L respektive CAN_H både till matningsspäningen och till signaljorden. Gör detta för varje signal och för båda samtidigt. Kortslut också signalerna CAN_L och CAN_H med varandra. Skicka ett meddelande för varje kortslutningsprov. Anteckna i tabell 1 om CAN-bussen fungerar (**OK** eller **FEL**) för olika kortslutningar.

Uppgift D

I de följande uppgifterna, D:1-6, ska gruppen utveckla programmet CANSJA. Två grupper ska arbeta parallellt med var sin källkod för var sin nod – en grupp med kod för nod **A** och en annan med kod för nod **B**. Arbeta i samma takt och testkör mot varandras noder efter varje övning innan nästa påbörjas.

1. Vi tänker oss att noderna använder endast var sin Tx-identifierare för meddelanden som sänds. Denna identifierare matas in från tangentbordet när programmet CANSJA startas och finns sedan kvar i variabeln **IdTx**. Det är funktionen **USART_ReceiveIdTx()** som hämtar in värdet på **IdTx** från terminalen. Funktionen **CAN_StrOut()** tar hjälp av **SJA_Write()** för att skriva in identifieraren och ett meddelande i **Transmit Buffer**. Det senare resulterar i att en ram skickas ut på CAN-bussen.

   Funktionen **SJA_Read()** används för att läsa i SJA1000-registren. För att se om det har kommit in nya meddelanden över CAN-bussen används följande sats i **main()**:

   ```
   if(SJA_Read(CAN_STATUS) & RBS)
   // CAN_STATUS är statusregistret
   // RBS = Receive Buffer Status
   ```

   När du programmerar och vill skriva text och värden i terminalfönstret, används funktionen **prints()** för teckensträngar och **printh()** för ett hexadecimalt tal med två tecken. Argumentet till **printh()** kan vara ett heltal < 256 eller ett hexadecimalt tal med två tecken, exempelvis 0xC3. (Den smarta funktionen **printf()** används tyvärr inte eftersom den förbrukar stora areor i flashminnet. Objektkoden kan väl rymmas i flashminnet, men under körning kommer **printf()** att behöva **FILE** som snart blir mycket stor.) Funktionerna **prints()** och **printh()** används exempelvis på följande sätt:

   ```
   unsigned int x1 = 255, x2 = 0x5A;
   char text[] = {'S', 't', 'o', 'p', 'p', '\0'};

   prints("Utmatning av identifierare: \n";
   prints("IdTx = "); printh(x1); printh(x2); prints("\n");
   prints(text);
   ```

   ```
   Utmatning av identifierare:
   IdTx = FF5A
   Stopp
   ```

   I terminalfönstret visas ovanstående som resultat av koden.

   Både terminalfönstret och LCD kan användas för debugging. De funktioner som med fördel kan användas för utmatning på LCD är **LCD_StrOut()** och **LCD_Convert()**. Den

förra fungerar likt **prints(),** dvs. kan användas för utmatning av textsträngar fastän på LCD. Hexadecimala tal matas ut på LCD med **LCD_Convert()**. Argumentet till denna funktion är av samma typ som till **printh()**. Ett exempel är som följer:

```
unsigned int x1 = 255, x2 = 0x5A;

LCD_Clean(LINE2);
LCD_StrOut(”IdTx = ”); LCD_Convert(x1); LCD_Convert(x2);
```

```
   *** CAN NODE ***
   IdTx = FF5A
```

På LCD visas ovanstående som resultat av koden.

Din uppgift är att mata ut den egna Tx-identifieraren i terminalfönstret varje gång som ett nytt meddelande visas på LCD.

Problemet är att vi bara kan skriva ut två hexadecimala tecken per heltal. Vi tar $7B5_{hex}$ = $(7 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0)_{dec}$ = 1973 $_{dec}$ som exempel. Om vi placerar 7 i en variabel och B5 i en annan, går det att mata ut dessa tre tecken med två **printh()**. Eftersom 7 anger sju st 256-vikter ($16^2 = 256$), kan vi använda följande kod:

```
Unsigned int IdTx = 0x7B5, x1, x2;

x1 = IdTx/256;
x2 = IdTx – x1*256;
```

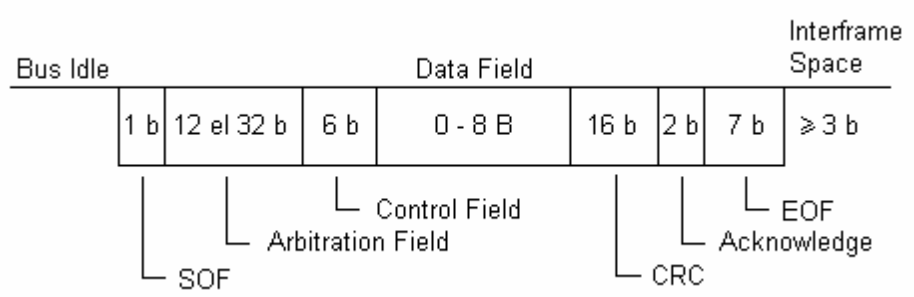
Heltalen **x1** och **x2** skrivs sedan ut med var sin **printh()**.

2. Visa också Rx-identifieraren för varje inkommande meddelande. För att lyckas med detta, krävs att vi känner till hur **Receive Buffer** är konstruerad. I denna buffert lagras nämligen identifieraren för inkommande ram. Några av registren i SJA1000 finns definierade i koden. Definitionerna finns i början av **cansja.c**. (Varje konstant i tabell 3 finns definierad i koden.)

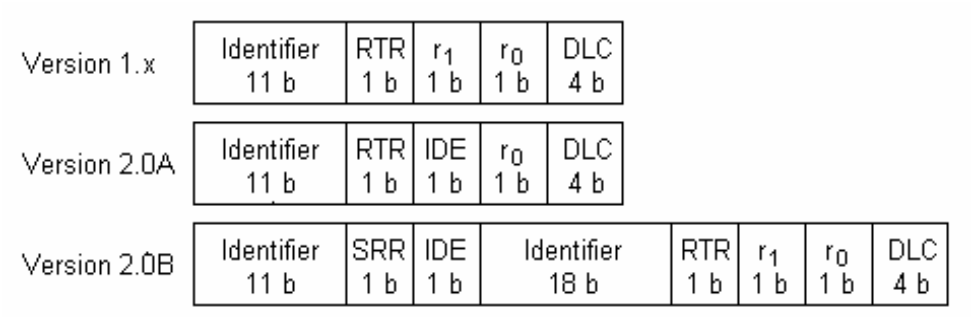
Konstant för register	Bitar							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
CAN_RX_ID	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3
CAN_RX_LEN	ID.2	ID.1	ID.0	RTR	DLC.3	DLC.2	DLC.1	DLC.0
CAN_RX_BUF0	b7	b6	b5	b4	b3	b2	b1	b0
CAN_RX_BUF1	b7	b6	b5	b4	b3	b2	b1	b0
CAN_RX_BUF2	b7	b6	b5	b4	b3	b2	b1	b0
CAN_RX_BUF3	b7	b6	b5	b4	b3	b2	b1	b0
CAN_RX_BUF4	b7	b6	b5	b4	b3	b2	b1	b0
CAN_RX_BUF5	b7	b6	b5	b4	b3	b2	b1	b0
CAN_RX_BUF6	b7	b6	b5	b4	b3	b2	b1	b0
CAN_RX_BUF7	b7	b6	b5	b4	b3	b2	b1	b0

Tabell 3. **Receive Buffer**.

Jämför **Receive Buffer** med ramformaten i figur 6 och figur 7.



Figur 6. Ramformatet på CAN-bussen.



Figur 7. Urskiljnings- och kontrollfält i olika nodversioner.

Vi använder ramformatet **2.0A**. Detta innebär att vi har en 11-bitars identifierare. Vi hittar också RTR-biten och de fyra DLC-bitarna i **Receive Buffer**. (IDE- och r0-biten finns inte i **Receive Buffer**.) De 11 bitarna i identifieraren finns i variablerna **CAN_RX_ID** (nr. 3 – 10) och i **CAN_RX_LEN** (nr. 0 – 2). Problemet är alltså att plocka ut bitarna och placera dessa i två variabler. Sedan ska samtliga 11 bitar visas som ett hexadecimalt värde i terminalfönstret.

För att kunna göra detta behöver vi C-operatorer som kan skifta bitar:
<< skiftar bitarna åt vänster, dvs. ger ett större värde
>> skiftar bitarna åt höger, dvs. ger ett mindre värde.

Exempel:

```
unsigned int temp1 = SJA_Read(CAN_RX_ID);
unsigned int temp2 = SJA_Read(CAN_RX_LEN);

temp1 = temp1 << 3 ;
// Skifta bitarna i temp1 tre steg åt vänster,
// dvs. skapa utrymme för tre bitar
// Vi får temp1 =
// ID.10 ID.9 ID.8 ID.7 ID.6 ID.5 ID.4 ID.3 0 0 0

temp2 = temp2 >> 5 ;
// Skifta bitarna i temp2 fem steg åt höger,
// dvs. ta bort RTR, DLC.3, DLC.2, DLC.1 och DLC.0.
// Vi får temp2 = 0 0 0 0 0 ID.2 ID.1 ID.0
```

Så måste vi göra för att få fram **temp1** med **temp2**. Sedan sätter vi ihop dem till Rx-identifieraren:

```
unsigned int IdRx = temp1 + temp2;
```

När väl **IdRx** har satts samman, kan värdet matas ut. Då ska givetvis samma teknik som behövdes för utmatning av **IdTx** användas, dvs. beräkna **x1** och **x2** för att sedan mata ut respektive med var sin **printh()**. Därför finns det två kommentarer i cansja.c om övning D:2, en första där **temp1** och **temp2** ska tas fram för att kunna sätta ihop dem till **IdRx**, samt en andra där **x1** och **x2** ska beräknas och matas ut.

3. Visa varje nytt meddelande i terminalfönstret tillsammans med de två identifierarna, **IdTx** och **IdRx**. (Låt utmatningen på LCD vara kvar i koden. Ändra inte på denna.)

   Vi läser av meddelandet i **Receive Buffer**. De åtta tecknen fås från **CAN_RX_BUF0 – CAN_RX_BUF7**. Några exempel:

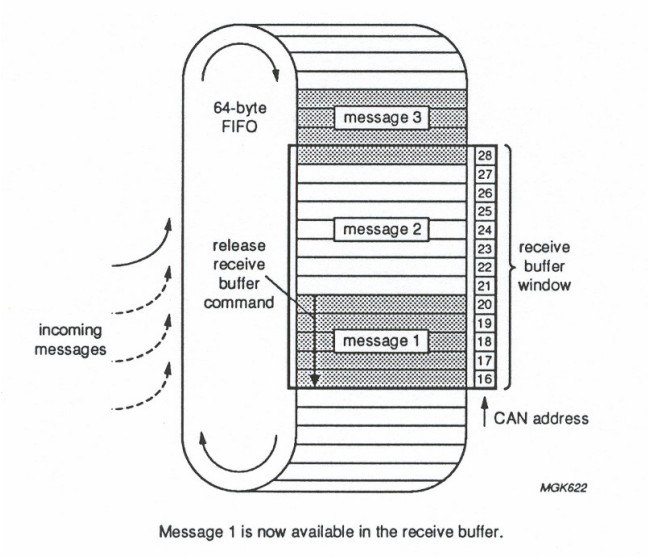
```
int i;
char R[9];

R[0] = SJA_Read(CAN_RX_BUF0);  // Sätt R[0] = första tecknet
R[i] = SJA_Read(CAN_RX_BUF0 + i);  // Passar bra för en loop
```

   Tänk på att ”överdimensionera” teckensträngen som används för meddelanden. Vi avläser åtta tecken men behöver ett nionde tecken för att lägga in ett NULL-tecken (**\0**). Detta NULL talar om för **prints()** var teckensträngen slutar.

Meddelandet ska alltså skrivas ut med hjälp av **prints()**. Du behöver också skriva ut någon form av ledtext så att användaren förstår att ett inkommande meddelande visas i terminalfönstret.

När vi har läst ett meddelande i **Receive Buffer**, ska denna buffert tömmas eftersom det kan stå flera meddelanden på kö. Inkommande meddelanden lagras i ett register av typen **First In First Out** (FIFO). Se figur 8. Tömningen görs med följande sats:

```
SJA_Write(CAN_CMD, RRB);   // RRB = Release Receiver Buffer
```



Figur 8. **Rx FIFO**.

4. Bygg ut CANSJA med ett acceptansfilter så att endast meddelanden som har utvalda identifierare visas i terminalfönstret. Dessutom ska Tx-identifieraren och Rx-identifieraren visas tillsammans med varje accepterat meddelande. (Annars ska de inte visas alls. På LCD ska alla inkommande meddelanden visas som tidigare.)

   Acceptansfiltret ska släppa igenom **IdRx** som har värden enligt tabell 4. Det ska alltså vara olika if-satser i nodernas kod. Exempelvis ska acceptansfiltret i nod **B** släppa in meddelanden som har **IdRx** lika med 000_{hex}, 005_{hex} eller $00A_{hex}$.

   Vid uppstart för test av den nya koden ska nod **A** använda **IdTx** = 000_{hex} och nod **B** använda **IdTx** = 002_{hex}.

   Kommentera noga if-satsen så att det tydligt framgår vilken nod acceptansfiltret är avsett för.

Nod	Tx-identifierare för vanliga meddelanden (IdTx)	Tx-identifierare för kvittenser (IdTxAck)	Rx-identifierare i acceptansfiltret (IdRx)
A	000	001	002, 007, 00C
B	002	003	000, 005, 00A

Tabell. 4. Identifierare under laborationen.

5. För komplett utmatning ska även **IdTxACK** visas i terminalfönstret. Värdet på **IdTxAck** ska sättas tidigt i koden, exempelvis redan i deklarationen av denna unsigned int. Skriv också in en kommentar som tydlig anger vad det är fråga om, exempelvis följande:

```
unsigned int IdTxAck = 0x001;    // För nod A
```

6. Bygg ut CANSJA så att varje meddelande som accepteras av respektive nod kvitteras med meddelandet **ACKNOW-j**. Sekvensnumret **j** ska vara ett tecken 0 – 9. Efter sekvensnummer 9 ska nästa sekvensnummer bli 0, dvs. en sekvensnumrering modulo 10.

   Kvittenserna från nod **A** ska ha identifieraren **IdTxAck** = 001_{hex} och kvittenserna från nod **B** ska ha **IdTxAck** = 003_{hex}.

   För att kunna sända, används variablerna enligt tabell 5, **Transmit Buffer**. Adresserna till SJA1000-registren finns listade i början av **cansja.c**. (Varje variabel i tabell 5 och är definierad som en adress.)

Konstant för register	Bitar							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
CAN_TX_ID	ID.10	ID.9	ID.8	ID.7	ID.6	ID.5	ID.4	ID.3
CAN_TX_LEN	ID.2	ID.1	ID.0	RTR	DLC.3	DLC.2	DLC.1	DLC.0
CAN_TX_BUF0	b7	b6	b5	b4	b3	b2	b1	b0
CAN_TX_BUF1	b7	b6	b5	b4	b3	b2	b1	b0
CAN_TX_BUF2	b7	b6	b5	b4	b3	b2	b1	b0
CAN_TX_BUF3	b7	b6	b5	b4	b3	b2	b1	b0
CAN_TX_BUF4	b7	b6	b5	b4	b3	b2	b1	b0
CAN_TX_BUF5	b7	b6	b5	b4	b3	b2	b1	b0
CAN_TX_BUF6	b7	b6	b5	b4	b3	b2	b1	b0
CAN_TX_BUF7	b7	b6	b5	b4	b3	b2	b1	b0

Tabell 5. **Transmit Buffer**.

Vi måste först placera in bitarna för **IdTxAck** i **Transmit Buffer**. Det är en motsatt procedur jämfört med att läsa en Rx-identifierare. Eftersom vi alltid sänder åtta tecken per meddelande ska DLC = 8_{dec} = 1000_{bin}. Skrivning i **Transmit Buffer** görs med funktionen **SJA_Write()**. Använd följande satser:

```
unsigned int IdTxAck = 0x001;  // Exempel för nod A

SJA_Write(CAN_TX_ID, IdTxAck >> 3);
SJA_Write(CAN_TX_LEN, (IdTxAck << 5) | 8);
```

När vi skiftar **IdTxAck** fem steg åt vänster, blir RTR = 0, DLC.3 = 0, DLC.2 = 0, DCL.1 = 0 och DLC.0 = 0. Med OR-operationen sätts sedan DLC.3 = 1, DLC.2 = 0, DCL.1 = 0 och DLC.0 = 0, dvs. vi får DLC = 8_{dec}.

För att skicka ett meddelande måste detta flyttas över till **Transmit Buffer** på SJA1000. Ett exempel:

```
WriteSja(CAN_TX_BUF0, 'A');  // Lägg tecknet A i buffert 0
```

Eftersom vi alltid sänder åtta tecken under denna laboration, ska också **CAN_TX_BUF1 – CAN_TX_BUF7** fyllas med tecken. **CAN_TX_BUF7** ska innehålla sekvensnumret. Siffrorna i ASCII-tabellen början med tecknet 0 som har värdet 48_{dec}. Sedan följer tecknen 1, 2, ..., 9 konsekutivt. För att överföra ACSII-tecken för siffrorna 0 – 9, används exempelvis följande kod:

```
int j;

SJA_Write(CAN_TX_BUF7, j + 48);  // ASCII-tecknet för j
```

Glöm inte bort att sekvensnumret (**j**) ska räknas upp modulo 10, dvs. sekvensen 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, ...

Till sist ska **IdTxAck** och tillhörande meddelande skickas ut på CAN-bussen. Använd följande sats:

```
SJA_Write(CAN_CMD, TXREQ);  // Sänd ut på CAN-bussen
```

I verkliga tillämpningar skriver man kompletta funktioner både för att sända och också ta emot meddelanden. Det finns sådana funktioner i programmet SJA1000, exempelvis **CAN_Check()**. Tanken är att laborationen ska visa på principer för att skriva och läsa i register i en CAN-kontroller. Därför använder vi inte dessa färdiga funktioner.

7. Provkör noga programmen i de bägge noderna mot varandra. Kontrollera att meddelanden med accepterade identifierare släpps in och ger kvittenser, att sekvensnumren räknas upp modulo 10 och att en kvittens inte ger en ny kvittens, dvs. att filtren fungerar.

VIKTIGT!
Ta bort projektet **cansja** och alla tillhörande filer i Eclipse när du är klar med laborationen. I annat fall får kommande grupp problem med att skapa projekt med samma namn. Gör följande:

Höger-klicka på projektets namn i fönstret Project Explorer > Delete > Bocka för **Delete project contents on disk (cannot be undone)** > OK

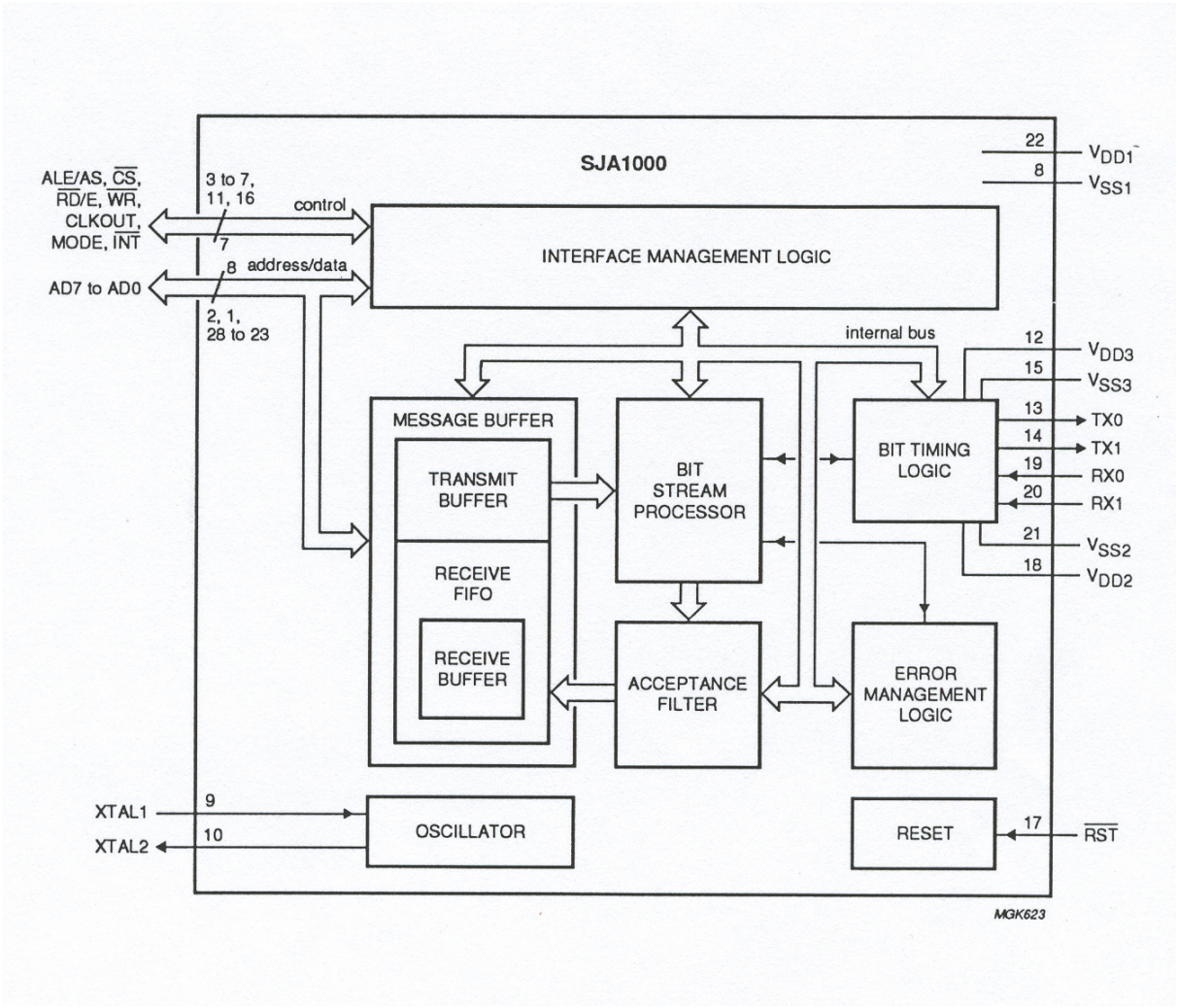
Frivillig uppgift

Utnyttja att en kopplingsdosa kan koppla ihop tre noder, eller varför inte ytterligare en kopplingsdosa och därmed ännu fler noder.

Ställ in filtren på lämpligt sätt så att det finns möjlighet att testa funktionen på bussen. Det får absolut inte bli kvittenser som kvitterar kvittenser.

Bilagor

Blockschema för CAN Controller SJA1000



Källkoden cansja.c

```
/*
**  Namn:             cansja.c
**  Källkod för nod: [A eller B?]
**  Version:          mega
**  Syfte:            studera protokollet för Philps CANSJA
**  Programvara:      AVR-GCC
**  Hårdvara:         MCU (AVR ATmega8515) på STK200-kort monterat på
**                    kopplingsbord med LCD, klockfrekvens 8 MHz
**                    kristall på kort)
**                    (ATmega8515 klara max 16 MHz)
**  LED:              10 st LED på MCU
**  LCD:              2-raders display på kopplingsbord, 16 tecken
**                    per rad
**  Design:           Jack Pencz
*/


#include <avr/io.h>
#include <util/delay.h>


/* Definitioner av funktioner för utmatning på RS232-port */
#define prints(s) USART_StrOut(s)   // Utmatning av sträng på terminal
via RS232-port
#define printh(h) USART_Convert(h)  // Utmatning av positivt heltal <
256 (eller 0xFF) av två hexadecimala tecken på terminal


/* Definitioner för LCD */
#define LINE1 1                  // Början av rad 1 på LCD
#define LINE2 168                 // Början av rad 2 på LCD
#define MSG1 "*** CAN NODE ***"  // Rubrik på LCD


/* Global konstant för CAN-buss */
unsigned char MSG2[] = {' ', 'O', 'N', '-', 'L', 'I', 'N', 'E'};


/* SJA1000-register (några av dessa) */
#define CAN_BASE 0
#define CAN_CTRL    0x00
#define CAN_CMD     0x01
#define CAN_STATUS  0x02
#define CAN_INT     0x03
#define CAN_AC      0x04
#define CAN_AM      0x05
#define CAN_TMG_0   0x06
#define CAN_TMG_1   0x07
#define CAN_OCR     0x08
#define CAN_TEST    0x09
#define CAN_TX_ID   0x0A
#define CAN_TX_LEN  0x0B
#define CAN_TX_BUF0 0x0C
#define CAN_TX_BUF1 0x0D
```

```
#define CAN_TX_BUF2 0x0E
#define CAN_TX_BUF3 0x0F
#define CAN_TX_BUF4 0x010
#define CAN_TX_BUF5 0x011
#define CAN_TX_BUF6 0x012
#define CAN_TX_BUF7 0x013
#define CAN_RX_ID   0x014
#define CAN_RX_LEN  0x015
#define CAN_RX_BUF0 0x016
#define CAN_RX_BUF1 0x017
#define CAN_RX_BUF2 0x018
#define CAN_RX_BUF3 0x019
#define CAN_RX_BUF4 0x01A
#define CAN_RX_BUF5 0x01B
#define CAN_RX_BUF6 0x01C
#define CAN_RX_BUF7 0x01D
#define CAN_CLKDIV  0x01F


/* Nyckelvärden och bitmasker för SJA1000*/
#define OWN_ID      0x0000      // CAN-ID för initiering, byts senare
ut mot IdTx från terminal
#define ACCEPTMASK  0xFF        // Acceptansmask
#define RESREQ      0x01        // Reset Request
#define RBS         0x01        // Receive Buffer Status
#define RRB         0x04        // Release Receive Buffer
#define TXREQ       0x01        // Transmit Request
#define TBA         0x04        // Transmit Buffer Access
#define COS         0x08        // Clear Overrun Status
#define BUSSTATUS   0x80        // Statreg bit 7, bus status
#define CANRX_INT   0x01
#define CANTX_INT   0x02
#define CANERR_INT  0x04
#define CANOVR_INT  0x08
#define CANWUP_INT  0x10


/* Definitioner för UBRR-registret i MCU för asynkron överföring med
normal hastighet, inte dubbel */
#define F_CPU 8000000UL  // Använd MCU-frekvensen 8 MHz (kristall på
kort)
#define USART_BAUDRATE 9600  // 9600 baud för terminal (Kermit)
#define BAUD_PRESCALE ((F_CPU/(USART_BAUDRATE * 16UL)) - 1)  // Bestäm
skalfaktor för RCX


void USART_Init(void)
{
 /* Initiera USART-kretsen i MCU för RS232-kommunikation

    Översikt av MCU-inställningar

    Inställningsordning:
      1. Överföringstyp
      2. Normal eller dubbel hastighet vid asynkron överföring
      3. Bithastighet
      4. Antal databitar
      5. Tillåt (enable) sändning och mottagning
```

```
      6. Paritet
      7. Antal stoppbitar

    UCSZ2 UCSZ1 UCSZ0 Databitantal
      0     0     0       5
      0     0     1       6
      0     1     0       7
      0     1     1       8
      1     1     1       9
    Intervallet 100-110 är reserverat.


    UPM1 UPM0 Paritet
      0    0  avstängd
      0    1  reserverad
      1    0  jämn paritet
      1    1  udda paritet


    USBS Stoppbitantal
      0        1
      1        2
    */

/* Sätt överföringstyp */
UCSRC &= ~(1 << UMSEL);   // Asynkron överföring
//UCSRC |= (1 << UMSEL);  // Synkron överföring

/* Sätt normal eller dubbel hastighet vid asykron överföring */
UCSRA &= ~(1 << U2X);     // Normal asynkron överföring
//UCSRA |= (1 << U2X);    // Dubbel hastighet vid asynkronöverföring

/* Sätt "signalhastighet" med enheten baud (antal signaltillstånd per
sekund) */
UBRRL = BAUD_PRESCALE;    // Ladda de 8 låga bitar i UBRR-registret
UBRRH = (BAUD_PRESCALE >> 8); // Ladda de 8 höga bitar i UBRR-registret
```

```
/* Tillåt mottagning och sändning */
UCSRB = (1 << RXEN) | (1 << TXEN); // Endast utan servicerutin för
mottagning
// UCSRB = ((1<<RXEN)|(1<<TXEN)|(1<<RXCIE)); // Endast med
servicerutin för mottagningsavbrott

/* Ramformat (default: 8 databitar, ingen paritet, 1 stoppbit)
För kommunikation med RCX: 8 databitar, udda paritet, 1 stoppbit */
UCSRC = (1 << URSEL) | (1 << UCSZ1) | (1 << UCSZ0) |(1 << UPM1) | (1 <<
UPM0);
}


void USART_Flush(void)
{
/* Rensar RS232-bufferten */

    while (UCSRA & (1<<RXC)) UDR;  // Rensa bufferten
}


void USART_SendByte(unsigned char u8Data)
{
/* Sändningsrutin RS232-port */

    while (!(UCSRA & (1<<UDRE)));  // Skicka nästa byte när
    sändningsbufferten blir tom
    UDR = u8Data;
}


void USART_CrLf(void)
{
/* Skickar "ny rad" till RS232-port */

    USART_SendByte(13);  // Skicka CR (Carriage return) till RS232-port
    USART_SendByte(10);  // Skicka LF (Line feed ) till RS232-port
}
```

```
void USART_StrOut(char *tknstr)
{
/* Sänder en teckensträng till terminal via RS232-port */
    unsigned char  chr;

    while (*tknstr != '\0')
    {
        chr = *tknstr++;
        if((chr == '\n') || (chr == '\r' )) USART_SendByte(13);
// Skicka CR (Carriage return) till RS232-port
        USART_SendByte(chr);  // Skicka tecken till RS232-port
    }
}


unsigned char USART_ReceiveByte(void)
{
/* Mottagningsrutin RS232-port */

    while(!(UCSRA & (1<<RXC)));  // Vänta tills mottagningsbufferten
    blir fylld
    return UDR;
}


unsigned int ToDec(unsigned char tkn)
{
/* Omvandling för hexadecimalt tal, från ACSII till decimalt tal */
unsigned int value, chr;

chr = (int) tkn;

if((48 <= chr) &&  (chr <= 57))
{
    value = chr - 48;  // Siffror 0-9
}
else if((65 <= chr) &&  (chr <= 70))
{
    value = chr - 55;  // Versaler A-F
}
else if((97 <= chr) &&  (chr <= 102))
{
    value = chr - 87;  // Gemener a-f
}
else
{
    value = 71;        // Error
}
    return (value);
}


void USART_Convert(unsigned char disp)
{
/* Anpassning av värde för visning på terminal via RS232-port
Hexadecimala tecken (eller positiva heltal < 256) omvandlas till
teckensträng
Exempel: LCD_Convert(0xA1) visar A1 på terminal
```

```
         LCD_Convert(255) visar FF på terminal
Bra för felsökning */
    int usart_H, usart_L, i;
    char text[] = {'\0','\0','\0'};

/* Delar upp disp i två separata hexadecimala tecken */
    usart_H = (int)disp/16;
    usart_L = disp - usart_H*16;
/* Omvandling från hexadecimala tecken till ACSII */
    for(i=0;i<10;i++)  // Urskilj tecknen 0-9
    {
        if(usart_H == i) text[0] = i + 48;
        if(usart_L == i) text[1] = i + 48;
    }
    for(i=0;i<6;i++)  // Urskilj tecknen A-F
    {
        if(usart_H == i+10) text[0] = i + 65;
        if(usart_L == i+10) text[1] = i + 65;
    }
    USART_StrOut(&text[0]);  // Skicka teckensträng till LCD
}


unsigned char USART_KbHit(void)
{
                        /* Känner av om tangent har tryckts ner på
terminalens tangentbord (via RS232-port) */

    if(bit_is_clear(PIND,0))
        return(1);           // Om databit 0 på PortD är satt,
returnera 1
    else
        return(0);           // Annars, returnera 0
}


unsigned int USART_ReceiveIdTx(void)
{
/* Läser in IdTx från RS232-port */
    unsigned char temp;
    unsigned int value, temp1, temp2, IdTx;

    USART_CrLf();
    USART_StrOut("Enter IdTx (000-7FF): ");

    temp = USART_ReceiveByte();   // Hämta tecken från terminal
    USART_SendByte(temp);         // Visa på terminalfönster (eko)
    value = ToDec(temp);          // Konvertering till decimalt värde
    IdTx = 256*value;             // Beräkna decimalt värde för X00

    temp = USART_ReceiveByte();   // Hämta tecken från terminal
    USART_SendByte(temp);         // Visa på terminalfönster (eko)
    value = ToDec(temp);          // Konvertering till decimalt värde
    IdTx = IdTx + 16*value;       // Beräkna decimalt värde för XX0

    temp = USART_ReceiveByte();   // Hämta tecken från terminal
    USART_SendByte(temp);         // Visa på terminalfönster (eko)
    value = ToDec(temp);          // Konvertering till decimalt värde
```

```
    IdTx = IdTx + 1*value;        // Beräkna decimalt värde för XXX

    if(IdTx > 2047)
    {  // BasicCan har 11-bitars identifierare, 0-2047
        USART_StrOut(" Error!");
        IdTx = 2047;  // Använd 1FF istället för otillåtet värde
    }


/* Utmatning på terminal via RS232-port: både text och det IdTx som
kommer att användas */
    USART_CrLf(); USART_CrLf();
    USART_StrOut("Using IdTx = ");
    temp1 = IdTx/256;
    temp2 = IdTx - temp1*256;
    USART_Convert(temp1);
    USART_Convert(temp2);
    USART_CrLf(); USART_CrLf();
    USART_StrOut("From now and on you have two choices:");
    USART_CrLf();
    USART_StrOut("1. Press [Enter] to write a message"); USART_CrLf();
    USART_StrOut("2. Don't press any key to wait for an incoming
message"); USART_CrLf();
    USART_CrLf();
    return(IdTx);
}
```

```
void LCD_Delay(void)
{
/* LCD-fördröjning anpassad för 8 MHz-klocka
 *
 * Eftersom
 *   #define F_CPU 8000000UL
 * och inte
 *   #define F_CPU 1000000UL
 * stämmer inte fördröjningen helt med antal angivna ms
 * Testat minumum är värdet 3
 * Värdet 10 ger god marginal
 */

     _delay_ms(10);
}


void LCD_Intro(unsigned char charline)
{
/* Går till början av LCD-rad = charline  */
    char  i, j;

    i = charline;
    j = i >> 4;    /* Swappa */
    j = j & 0x0F;

    PORTC = j;
    j = j | 0x10;  /* EN hög och RS låg */
    PORTC = j;
    j = j & 0x0F;  /* EN låg och RS låg */
    PORTC = j;

    LCD_Delay();
    i = i & 0x0F;
    PORTC = i;
     i = i | 0x10;
    PORTC = i;
    i = i & 0x0F;
    PORTC = i;
    LCD_Delay();
}
```

```
void LCD_Init(void)
{
/* Initieringsdata för LCD */

    LCD_Delay();
    LCD_Delay();
    LCD_Delay();
    DDRC = 0xFF;
    PORTC = 0x00;
    LCD_Delay();
    PORTC = 0x03;
    PORTC = 0x13;
    PORTC = 0x03;
    LCD_Delay();
    LCD_Delay();
    PORTC = 0x03;
    PORTC = 0x13;
    PORTC = 0x03;
    LCD_Delay();
    PORTC = 0x03;
    PORTC = 0x13;
    PORTC = 0x03;
    LCD_Delay();
    PORTC = 0x02;
    PORTC = 0x12;
    PORTC = 0x02;
    LCD_Delay();
    LCD_Intro(0x28);
    LCD_Intro(0x0C);
    LCD_Intro(0x01);
    LCD_Intro(0x06);
}
```

```
void LCD_CharOut(unsigned char tkn)
{
/* Visar tecken för tecken på LCD */
    unsigned char  i, j;

    i = tkn;
    j = i >> 4;         /* Swappa */
    j = j & 0x0F;
    j = j | 0x20;       /* RS hög */
    PORTC = j;
    j = j | 0x30;       /* EN hög och RS hög */
    PORTC = j;
    j = j & 0x2F;       /* EN låg och RS hög */
    PORTC = j;

    LCD_Delay();
    i = i & 0x0F;
    i = i | 0x20;       /* RS hög */
    PORTC = i;
    i = i | 0x30;       /* EN hög */
    PORTC = i;
    i = i & 0x2F;       /* EN låg */
    PORTC = i;
    LCD_Delay();
}


void LCD_StrOut(char *tknstr)
{
/* Visar textsträng på LCD */
    unsigned char  chr;

    while (*tknstr != '\0')
    {
        chr = *tknstr++;
        LCD_CharOut(chr);  // Skickar tecken för tecken till LCD
    }
}


void LCD_Clean(unsigned char line)
{
                        /* Rensar LCD-rad */

    LCD_Intro(line);                    // Gå till början av LCD-raden
    LCD_StrOut("               ");  // Rensa LCD-raden
    LCD_Intro(line);                    // Gå till början av LCD-raden
}
```

```
void LCD_Convert(unsigned char disp)
{
/* Anpassning av värde för visning på LCD
Hexadecimala tecken (eller positiva heltal < 256) omvandlas till
teckensträng
Exempel: LCD_Convert(0xA1) visar A1 på LCD
         LCD_Convert(255) visar FF på LCD
Bra för felsökning */
    int lcd_H, lcd_L, i;
    char lcdtext[] = {'\0','\0','\0'};

/* Delar upp disp i två separata hexadecimala tecken */
    lcd_H = (int)disp/16;
    lcd_L = disp - lcd_H*16;

/* Omvandlar från hexadecimala tecken till ACSII */
    for(i=0;i<10;i++)  // Urskilj tecknen 0-9
    {
        if(lcd_H == i) lcdtext[0] = i + 48;
        if(lcd_L == i) lcdtext[1] = i + 48;
    }
    for(i=0;i<6;i++)  // Urskilj tecknen A-F
    {
        if(lcd_H == i+10) lcdtext[0] = i + 65;
        if(lcd_L == i+10) lcdtext[1] = i + 65;
    }

    LCD_StrOut(&lcdtext[0]);  // Skickar teckensträng till LCD
}
```

```
unsigned char SJA_Read(unsigned char Adr)
{
/* Läser av SJA-registervärde (CAN-kontroller) */
    unsigned char R = 0;

    DDRD = 0xFF;
    DDRA = 0xFF;
    PORTA = Adr;

    PORTD |= _BV(5);        // ALE
    PORTD &= ~(_BV(5));

    DDRA = 0x00;
    PORTD &= ~(_BV(7));     // RD
    PORTA = 0xFF;           // PULL UP

    R = PINA;

    PORTD |= _BV(7);        // RD

    return(R);
}


void SJA_Write(unsigned char Adr, unsigned char Value)
{
/* Skriver in SJA-registervärde (CAN-kontroller) */

    DDRD = 0xFF;
    DDRA = 0xFF;

    PORTA = Adr;            // Address

    PORTD |= _BV(5);        // ALE
    PORTD &= ~(_BV(5));

    PORTA = Value;

    PORTD &= ~(_BV(6));     // WR
    PORTD |= _BV(6);
}
```

```
void SJA_Init(unsigned char *btr0, unsigned char *btr1)
{
/* Initierar SJA-register i CAN-kontroller */
    unsigned char B = 0;

    DDRD = 0xFF;  // Använd port D
    PORTD = 0xC0;

    B = SJA_Read(CAN_CTRL);
    B = B & RESREQ;
    while (B == 0)
    {
        SJA_Write(CAN_CTRL, RESREQ);
        B = SJA_Read(CAN_CTRL);
        B = B & RESREQ;
    }
    SJA_Write(CAN_AC,OWN_ID);
    SJA_Write(CAN_AM, ACCEPTMASK);
    SJA_Write(CAN_TMG_0, *btr0);
    SJA_Write(CAN_TMG_1, *btr1);
    SJA_Write(CAN_OCR, 0x1A);
    SJA_Write(CAN_CLKDIV, 0x47);
    SJA_Write(CAN_CTRL, 0x5E);
    SJA_Write(CAN_CMD, 0x0C);
}
```

```
void SJA_Dump(void)
{
/* Hämtar några SJA-registervärden (CAN-kontroller) och visar på
terminal via RS232-port */
    USART_CrLf(); USART_CrLf();
    USART_StrOut("SJA1000 Registers");
    USART_CrLf(); USART_CrLf();
    USART_StrOut("STATUS = "); USART_Convert(SJA_Read(CAN_STATUS));
    USART_CrLf();
    USART_StrOut("TX_ID  = "); USART_Convert(SJA_Read(CAN_TX_ID));
    USART_CrLf();
    USART_StrOut("TX_LEN = "); USART_Convert(SJA_Read(CAN_TX_LEN));
    USART_CrLf();
    USART_StrOut("TX_BUF = ");
    USART_Convert(SJA_Read(CAN_TX_BUF0)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_TX_BUF1)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_TX_BUF2)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_TX_BUF3)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_TX_BUF4)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_TX_BUF5)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_TX_BUF6)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_TX_BUF7)); USART_SendByte(32);
    USART_CrLf();
    USART_StrOut("RX_ID  = "); USART_Convert(SJA_Read(CAN_RX_ID));
    USART_CrLf();
    USART_StrOut("RX_LEN = "); USART_Convert(SJA_Read(CAN_RX_LEN));
    USART_CrLf();
    USART_StrOut("RX_BUF = ");
    USART_Convert(SJA_Read(CAN_RX_BUF0)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_RX_BUF1)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_RX_BUF2)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_RX_BUF3)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_RX_BUF4)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_RX_BUF5)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_RX_BUF6)); USART_SendByte(32);
    USART_Convert(SJA_Read(CAN_RX_BUF7)); USART_SendByte(32);
    USART_CrLf();
    SJA_Write(CAN_CMD, 0x04);  // Frigör Rx-bufferten
}
```

```
unsigned int SJA_CheckId(void)
{
/* Avläser inkommande rams RxId i SJA-registren (CAN-kontroller) */
    unsigned int IdRx = 0;
    unsigned char tmp1 = 0, tmp2 = 0;

    tmp1 = SJA_Read(CAN_RX_ID);
    tmp2 = SJA_Read(CAN_RX_LEN);

    if (SJA_Read(CAN_STATUS) & RBS)
    {
        IdRx = ((unsigned int) tmp1 << 3) + ((tmp2 >> 5) & 0x07);
    }
    return(IdRx);
}


void CAN_StrOut(unsigned int IdTx, unsigned char *B)
{
/* Skickar ut meddelande (en byte) på CAN */

    unsigned char datalength = 8, tmp = 0;

    do
    {
        tmp = SJA_Read(CAN_STATUS);
        tmp = tmp & TBA;
    } while (tmp != TBA);

    SJA_Write(CAN_TX_ID, IdTx >> 3);
      // Skriver in åtta IdTx-bitar (inte de tre msb) i ID.10 - ID.3
    SJA_Write(CAN_TX_LEN, (IdTx << 5) | datalength);
      // Skriver in tre IdTx-bitar (msb) i ID.2-ID.0 och ordlängden i
DLC3-DLC.0
    SJA_Write(CAN_TX_BUF0, B[0]);
      // Skriver in texten, ett tecken per register
    SJA_Write(CAN_TX_BUF1, B[1]);
    SJA_Write(CAN_TX_BUF2, B[2]);
    SJA_Write(CAN_TX_BUF3, B[3]);
    SJA_Write(CAN_TX_BUF4, B[4]);
    SJA_Write(CAN_TX_BUF5, B[5]);
    SJA_Write(CAN_TX_BUF6, B[6]);
    SJA_Write(CAN_TX_BUF7, B[7]);
    SJA_Write(CAN_CMD, TXREQ);
}
```

```
void CAN_Check()
{
/* Visar inkommande rams RxId och meddelande på LCD */
    unsigned char i = 0, c = 0, tkn[8];
    unsigned int IdRx = 0, temp1, temp2;

    IdRx = SJA_CheckId();  // Hämtar inkommande rams RxId från CAN-
kontrollerns register
    LCD_Intro(LINE2);  // Går till början av LCD-rad 2
    LCD_StrOut("RX ");
    temp1 = IdRx/256;  // Beräkna och skriv de hexadecimala tecknen,
två och två, för IdRx
    temp2 = IdRx - temp1*256;
    LCD_Convert(temp1);
    LCD_Convert(temp2);
    LCD_StrOut(" ");

    for (i = 0; i < 8; i++)
    {
        tkn[i] = SJA_Read(0x16 + i);  // När en ram har kommit in,
hämtas och visas meddelandet på LCD
        c = tkn[i];
        LCD_CharOut(c);  // Mata ut tecken för tecken på LCD
    }

    USART_CrLf();
}
```

```
void CAN_MessOut(unsigned int IdTx)
{
/* Matar in meddelande, 8 bytes, från terminal via RS232-port och
sänder på CAN */
    unsigned char tmp = 0, i = 0, txdata[9];

    tmp = USART_ReceiveByte();
    USART_CrLf();
    USART_StrOut("Enter a message to send (maximum 8 characters):");
      // Sträng till terminal via RS232-port
    USART_CrLf();

    for (i = 0; i < 8; i++)
    {
        tmp = USART_ReceiveByte();
          // Hämta tecken från terminal via RS232-port
        if (tmp == 13) for (; i < 8; i++) txdata[i] = '*';
          // Om < 8 tecken, fyll upp med "*"
        if (tmp > 96) tmp = tmp - 32;  // Byter till versaler
        txdata[i] = tmp;  // Lägger tecken i txdata
        USART_SendByte(tmp);  // Visar på terminalen (eko)
    }
    txdata[8]= '\0';  // NULL i sista elementet
    USART_CrLf();  // Ny rad på terminal via RS1232-port
    CAN_StrOut(IdTx, &txdata[0]);
      // Sänder meddelandet, txdata, på CAN-bussen
}
```

```
int main(void)
{
     unsigned char btr0 = 0x03, btr1 = 0x1C;
     unsigned int IdTx;

    /* Initieringar och registeravläsning */
     USART_Init();                  // Initierar USART
     USART_Flush();                 // Rensar RS232-bufferten
     LCD_Init();                    // Initierar LCD
     DDRA = 0xFF;  // Sätter Port A på STK200-kortet för bussar
till/från CAN-kontroller
     DDRB = 0xFF;  // Sätter Port B på STK200-kortet för LED:s (data
output, obs flera olika kontakttyper)
     // Sätter Port C på STK200-kortet för LCD (data output)
     DDRD = 0x00;  // Sätter Port D på STK200-kortet för bussar
till/från CAN-kontroller
     // Sätter Port E på STK200-kortet för NC
     SJA_Init(&btr0, &btr1); // Initierar CAN-kontrollerna SJA1000 till
125 kbit
     SJA_Dump();  // Visar CAN-registren på terminalen via RS232-porten
 // Status bör vara 0C, om inte starta om med Powerknappen
 // (Observera att några SJA1000-register inte kan avläsas i körläget)

     /* Visar text på LCD */
     LCD_Clean(LINE1); // Rensar LCD-rad 1 och gå till början av den
     LCD_StrOut(MSG1); // Matar ut titeln på LCD-rad 1
     LCD_Clean(LINE2); // Rensar LCD-rad 2 och gå till början av den

     /* Läser in IdTx och skickar en ram på CAN-bussen */
     IdTx = USART_ReceiveIdTx();
// Läser in Tx-identifieraren från terminalens tangentbord
     CAN_StrOut(IdTx, &MSG2[0]);
// Skickar dataram på CAN med texten "xxx ON-LINE ", där xxx = IdTx

     /* Evighetsloop */
     while(1)
     {
         if(SJA_Read(CAN_STATUS) & RBS)
// Kontrollerar om ram har kommit in till CAN-kontrollerns Rx
                        {
         CAN_Check();
// Hämtar in IdRx och meddelande från CAN-kontroller och visar på LCD
```

```
/* Övning D:2 - Plats för att beräkna IdRx utifrån registrevärdena */


/* Övning D:4 - Plats för filtret */


/* Övning D:1 - Plats för att beräkna x1 och x2 från IdTx
   samt att mata ut IdTx som två hexadecimala tal (x1 och x2) på
   terminalen */


/* Övning D:5 - Plats för att beräkna x1 och x2 från IdTxAck
   samt att mata ut IdTxAck som två hexadecimala tal (x1 och x2) på
   terminalen */


/* Övningen D:2 - Plats för att beräkna x1 och x2 från IdRx
   samt att mata IdRx som två hexadecimala tal (x1 och x2) på
   terminalen */


/* Övning D:3 - Plats för att avläsa meddelandet (data) i inkommande
   ram från CAN-kontrollern och att mata ut på terminalen */


/* Övning D:6 - Plats för att sätta samman kvittenserna "ACKNOW-j" och
   att skicka dessa till avsändaren över CAN-bussen */

         SJA_Write(CAN_CMD,RRB);
         // Frigör Rx-bufferten i CAN-kontrollern (SJA1000)
         }
         if(USART_KbHit()) CAN_MessOut(IdTx);
// Om det finns data att skicka, hämtar detta från terminalen (
// RS232-porten, flyttar över data till CAN-kontrollern (SJA1000)
   och skickar ut det på CAN-bussen
     }
}
```