Örebro universitet
Institutionen för naturvetenskap och teknik
JPZ

DATORKOMMUNIKATION OCH NÄT

Laboration 1
RCX Byte Code
(RCX)

Laborationens förhållande till teorin

Laborationen **RCX Byte Code** visar intressanta principer för datakommunikation. RCX-enheten är en robot (realtidsprogram) med serverfunktion. Modellen klient/server förekommer i hög grad i lokala nät och på Internet. Modellen beskriver hur programmen klient och server kommunicerar. En server har till uppgift att besvara frågor som sänds från klienter. En klients begäran (eng. *request*) besvaras av servern med ett svar (eng. *response, reply*). Servrar kallas också hanterare och några exempel på sådana är filserver, applikationsserver, databasserver, FTP-server och webbserver.

Vidare beskrivs kommunikation med ramar vars fält har bestämda uppgifter: header för synkronisering, operationskod, data och även checksumma för mottagningskontroll. Operationskoder och data skrivs in i ramarna tillsammans med respektive 1-komplement. Dessutom tillämpas alternerande bytekod, dvs. upprepad operationskod måste anges med ett värde som skiljer sig med 8 jämfört med den senaste. Detta kan användas som en del i mottagningskontrollen. Du får också se ett enkelt sätt att beräkna och kontrollera checksummor. Till mottagningskontrollen hör också att skicka kvittenser till avsändaren. Svarskoden från en RCX-enhet är 1-komplementet till operationskoden.

Redovisning

Laborationen **RCX Byte Code** redovisas med en skriftlig rapport. I denna rapport ska du beskriva laborationen, redovisa dina svar och programmet som du har skrivit. 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
   Uppdatera denna (höger-klicka på innehållsförteckningen för ändamålet).
3. Bakgrund
   Beskriv laborationens koppling till teorin för datorkommunikation
4. Resultat
   Uppgift A: beskrivning av uppgiften, skiss över systemet (A) och svar på frågorna
   Uppgift B: beskrivning av uppgiften, skiss över systemet (B) och hänvisning till bilagan
5. Bilaga
   Programkod enligt uppgift B

Introduktion

Syftet med denna laboration är att lära känna ett enkelt protokoll för kommunikation, nämligen **RCX Byte Code**. Det är praktiskt fråga om programkoden som används i RCX-enheten i **LEGO MindStorms – Robotics Invention System**. RCX är förkortningen för **Robotics Command System**. RCX-enheten används inte bara för att roa vetgiriga barn och ungdomar utan också i utbildningen av tekniker och ingenjörer. RCX och efterträdaren NXT används bl.a. i projekt vid Institutionen för naturvetenskap och teknik.

Under laborationen används ett färdigt program skrivet i Visual Basic (VB) för att skicka operationskoder till RCX-enheten. Programmet kallas **LEGO V0.1**. Detta program visar checksumman och det svar som kommer från RCX-enheten. Övningen kommer att ge dig förståelse för dataramens uppbyggnad.

Du ska också skriva ett eget C-program som skickar operationskoder till RCX-enheten från en MicroController Unit (MCU), mikroprocessor. För ändamålet kommer en AVR-mikroprocessorn från Atmel att användas. De flesta mikroprocessorer som används för styrning och reglering har 8-bitars databuss. Det är datastorleken som gäller, även om Atmel tillverkar både 8- och 32-bitars mikroprocessorer.

Oavsett om VB- eller C-programmet används, görs kommunikationen med RCX-enheten via en IR-transceiver som hör till **LEGO MindStorms**. I det förra fallet ansluts IR-transceivern, det s.k. IR-tornet, till persondatorns USB-port via en USB-RS232-adapter och i det senare till RS232-porten för MCU.

Styrning av RCX-enheten från programmet LEGO V0.1

RCX-enheten

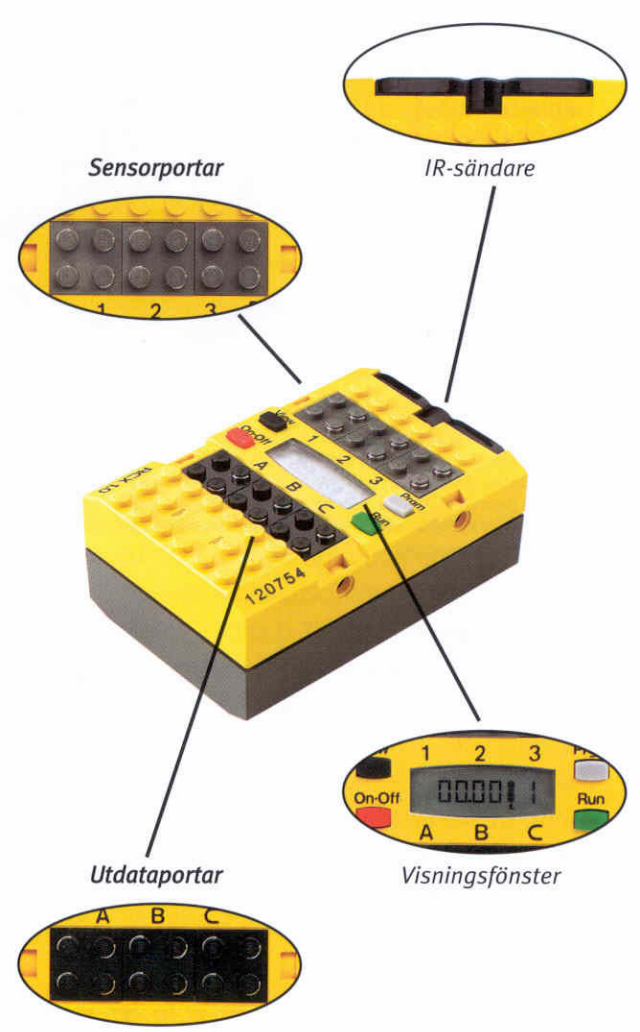
Den förvuxna LEGO-biten som kallas RCX-enhet innehåller ett komplett mikrodatorsystem som har ett eget operativsystem. Inne i RCX-enheten finns en 64-pinnars mikrokontroller från H8-familjen (Hitachi) som arbetar med klockfrekvensen 16 MHz.

Det finns minnen både i form av **Read Only Memory** (ROM) och **Random Access Memory** (RAM). ROM är på 16 kB och innehåller grundrutinerna som t.ex. laddare av operativsystemet (eng. *Bootstrap*). RAM är på 32 kB och används för dels operativsystemet (16 kB) och RCX-programmen (maximalt 16 kB). RCX-programmen skrivs med **RCX Byte Code** som är uppbyggt av tiotals kommandon/förfrågningar. Både operativsystemet och RCX-programmen laddas ned till RCX-enheten från en persondator över ett seriellt gränssnitt, numera i form av **Universal Serial Bus** (USB).

RCX-enheten drivs av sex AA/LR6-batterier (1,5 V per batteri). Man har omkring en minut på sig för att utföra ett batteribyte. Om bytet inte görs tillräckligt snabbt, kommer innehåller i RAM att raderas. Detta innebär att operativsystemet måste laddas ned på nytt till RCX-enheten.

För att spara ström har RCX-enheten en funktion för automatisk avstängning. RCX-enheten stängs av om den inte har använts under en bestämt tid. Denna tid är från början 15 min.

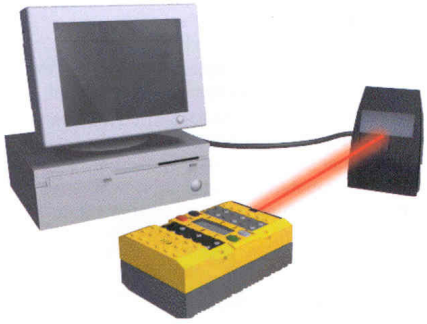
Tiden för den automatiska avstängningen kan ändras och ställas från 1 till 99 min.
(Inställningen 0 har en speciell betydelse enligt bilaga 1.)



Figur 1. RCX-enheten.
(I figuren står det IR-sändare men denna är egentligen en IR-transceiver.)

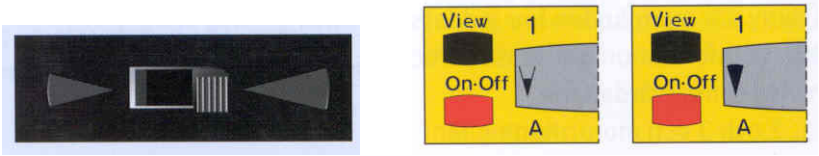
Det finns tre utportar (A, B och C) för styrdon som t.ex. motorer. Inportarna (1, 2 och 3) används för att ansluta tvåtråds givare (sensorer) som t.ex. fjädrande strömbrytare (switchar) och ljussensor. (Den ljussensor som följer med MindStorms har en röd lysdiod som sänder ut ljus och en fotodetektor som känner av om detta ljus reflekteras tillbaka.)

För att kunna skicka över operativsystemet och RCX-programmet från en persondator till en RCX-enhet, används trådlös överföring med IR-ljus. På RCX-enheten sitter en IR-transceiver kan kommunicera med IR-tornet. Den senare ansluts med en kabel till en serieport på persondatorn, i senare versioner till en USB-port. Det är endast **Transmit Data** (TxD) och **Receive Data** (RxD) som används av alla tillgängliga signaler. För att kunna använda en serieport i persondatorn, är signalerna **Request to Send** (RTS) och **Clear to Send** (CTS) hopkopplade inne i IR-tornet.



Figur 2. Trådlös överföring mellan persondatorn och RCX-enheten.

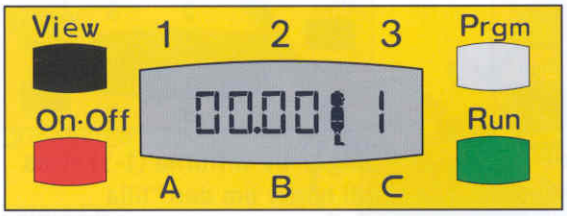
På IR-tornet finns en omställare för kort och långt avstånd till RCX-enheten. ”Långt avstånd” kan vara flera meter. RCX-enhetens **Liquid Crystal Display** (LCD) visar vilket avstånd som används.



a b

Figur 3. a. Omställare för kort (vänster) och långt (höger) avstånd.
b. LCD på RCX-enheten. Vänster delbild visar kort avstånd och höger visar långt.

På RCX-enheten finns det fyra knappar: **On·Off**, **Run**, **View** och **Prgm**. När RCX-enheten ställs om till läge **Run** ändras symbolen på LCD till en springande gubbe istället för en stående. I figur 4 är gubben stillastående, dvs. inget RCX-program körs för närvarande. Siffran till höger om gubben anger vilket RCX-program som körs. I figur 4 har RCX-enheten inställd på RCX-program 1. Man byter RCX-program med knappen **Prgm**. Det finns fem färdiga RCX-program (1 – 5). Dessutom är det tänkt att användaren ska kunna skriva egna RCX-program som laddas ned till RCX-enheten via IR-tornet. Sådan skriver över de färdiga programmen. (Återställning av operativsystemet och de färdiga programmen görs med den programvara som finns med i MindStorms.) Med knappen **View** kan man få upp mer information på LCD om sensorvärden (logisk nolla och etta) och motor-riktningar (medurs och moturs).

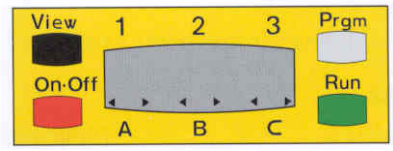


Figur 4. Knappar och LCD på RCX-enheten.

Motor-riktningen påverkas av hur kontakten har anslutits till RCX-enheten. Kontakten är en fyrkantig LEGO-bit som alltså kan vändas på fyra olika sätt. Därför rekommenderas att både motorer och sensorer ansluts enligt figur 5.

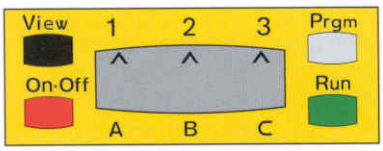
Figur 5. Rekommenderade riktningar för kontakter till sensorer och motorer.

LCD på RCX-enheten kommer att visa motor-riktningarna. En sådan visas med en pil, antingen moturs (vänster) eller medurs (höger), på respektive utport (A, B och C). I figur 6 visas alla pilar på samtliga utportar. Detta är naturligtvis inte möjligt under en normal programkörning.



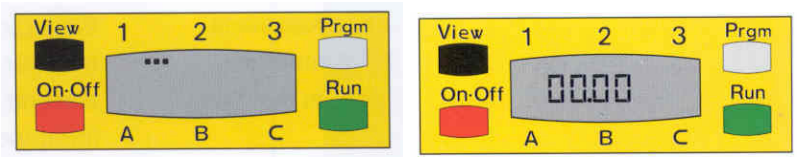
Figur 6. Visning av motor-riktningar.

Med knappen **View** väljs vilken av inportarna som man vill avläsa. Det som kommer att visas är det logiska värdet (logiska nolla eller logisk etta). På LCD markeras aktuell inport med en pil. I figur 7 visas pilar vid alla inportar. Detta är naturligtvis inte möjligt med en korrekt fungerande RCX-enhet.



Figur 7. Val av inport för visning av logiskt värde.

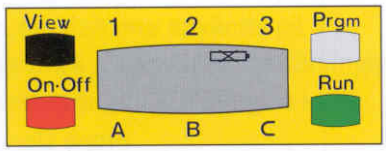
Under nedladdning av program visar LCD tre punkter. Annars visas tiden som RCX-enheten har varit påslagen (i antal minuter) sedan den senaste inställningen eller återställningen. Se figur 8.



a b

Figur 8. a. LCD under nedladdning av program.
b. LCD under normal drift.

Om LCD visar en symbol med ett överkryssat batteri, är det dags att byta batteri. Om strömmen inte räcker till, måste man i värsta fall ladda ned operativsystemet på nytt. Därför är det bra att snarast byta batteri vid en sådan varning. (Man har som tidigare nämnt, någon minut på sig för batteribytet, dvs. innehållet i RAM bevaras under en kort tid utan batteri.)



Figur 9. Dags för batteribyte.

RCX Byte Code

När man kommunicerar med RCX-enheten används ramar för att sända över operationskoder (kommandon/förfrågningar) och argument. Operationskoder finns listade i bilaga 1. Varje ram har ett huvud med kontrollinformation för synkronisering. Denna kontrollinformation är **55 FF 00$_{hex}$** vilket ger ett tydligt bitmönster för start av ram. Motsvarande bitmönster visas i figur 11.

Figur 10. Ramen för **RCX Byte Code**.

55	FF	00
01010101	11111111	00000000

Figur 11. Ramens huvud (header): hexadecimalt resp. binärt.

Operationskoden, data och checksumman skrivs in i ramen med sina 1-komplement (bitvis invertering). Varannan oktett är 1-komplementet av föregående oktett. Låt O vara oktett med operationskod, D_i vara oktett med data och C vara oktett med checksumma. Då skrivs operationskoden, data och checksumman in i ramen med

$$O\overline{O}D_1\overline{D_1}D_2\overline{D_2}\cdots D_n\overline{D_n}C\overline{C}$$

där beteckningen ”överstreck” betyder 1-komplement. Om beräkningen görs med decimala värden, har vi t.ex.

$$\overline{D_1} = 255 - D_1$$

Detta motsvaras med hexadecimala värden av

$$\overline{D_1} = FF - D_1$$

Checksumman beräknas som

$$C = O + D_1 + D_2 + \cdots + D_n$$

Om checksumman $C > 255_{dec}$ så används endast de åtta minst signifikanta bitarna.

Operationskoden kallas alternerande bytekod eftersom man för ett och samma kommando/förfrågan har två koder som skiljer sig med värdet 8. Man måste alternera mellan dessa. Låt O vara den första operationskoden som används och låt O vara det låga av de två

värdena som används för ett och samma kommando/förfrågan. Om samma kommando/förfrågan ska upprepas i en sekvens, används följande koder: O, $O + 8$, O, $O + 8$, osv. Denna regel upphör om något annat (eller några) kommando/förfrågan används mellan tillfällen för ett bestämt kommando/förfrågan.

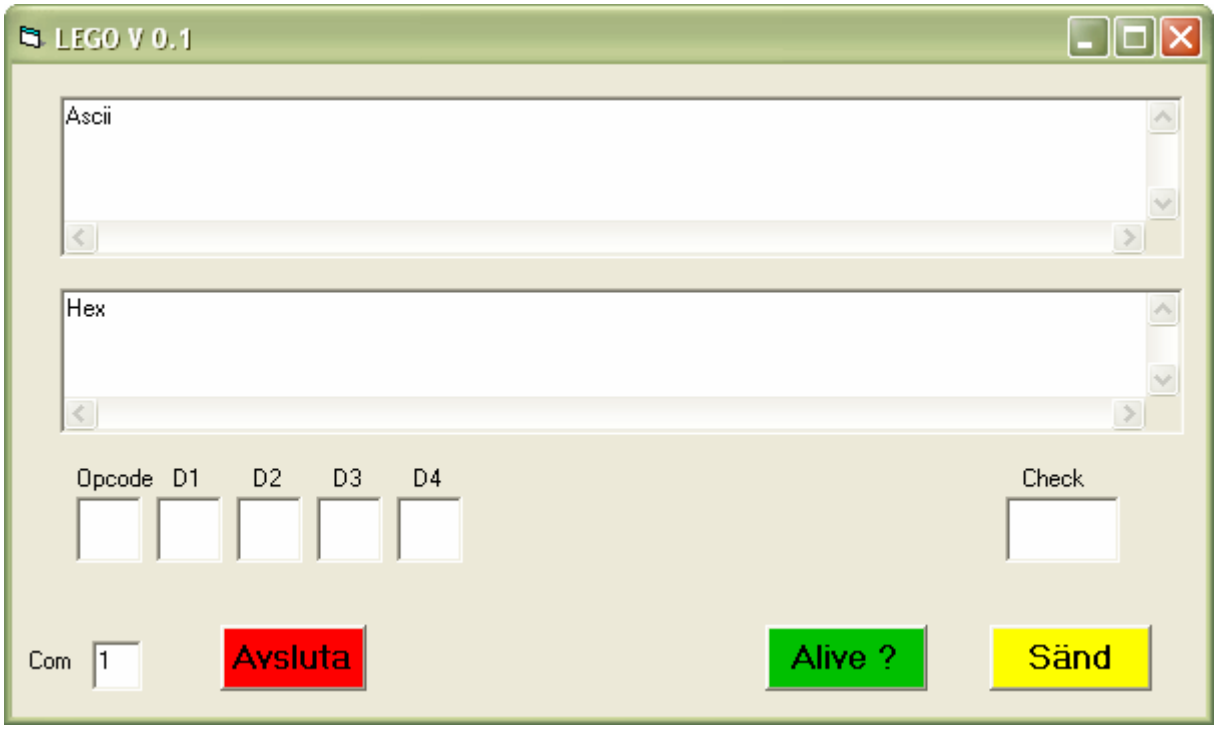
Man kan skilja mellan O och $O + 8$ genom att studera den fjärde minst signifikanta biten. Eftersom operationskoderna är heltal, har denna bit vikten $2^3 = 8$. Effekten av att skicka ett kommando/förfrågan med alternerande bytekod är att denna bit växlar mellan de logiska värdena noll och ett (åt endera hållet). Detta kan användas för att ge sekvensnummer åt RCX-ramarna. Mottagaren kan på så sätt kontrollera den fjärde minst signifikanta biten som alltid ska växla mellan logiska nolla och etta. För att detta ska fungera, måste man alltid använda koden för varje kommando/förfrågan som ger växling i den fjärde minst signifikanta biten. Detta gäller naturligtvis även om inte samma kommando/förfrågan upprepas. Dessvärre har undersökningar av RCX-enheten visat att denna finess inte används.

De flesta kommandon/förfrågningar har ett svar som skickas från RCX-enheten. Svaret ger alltid växling i den fjärde minst signifikanta biten (i svarskoden) jämfört med kommandot/förfrågan. Detta kan användas av avsändaren som kontroll, dvs. en slags kvittering. För övrigt är svarskoden 1-komplementet till kommandot/förfrågan.

Uppgift A

(Läraren demonstrerar uppgiften.)

Läraren ansluter ett IR-torn till en USB-port (via en USB-RS232-adapter) på en persondator. RCX-enheten placeras framför IR-tornet enligt figur 2. Läraren kör testprogrammet **LEGO V0.1** (**Lego.exe**).



Fältet **Ascii** visar innehållet i den ram som skickas ut med decimala värden. Direkt efter ramen visas svarsram från RCX-enheten. I fältet **Hex** visas utsänd ram och svarsram fastän med hexadecimala värden. Det är Hex-fältet som rekommenderas för följande uppgifter.

1. a. Skicka färdig ram som ger förfrågan **Alive**. Detta är en förfrågan för att testa förbindelsen med RCX-enheten. Alive skickas till RCX-enheten när du klickar på knappen på knappen **Alive?** Om allt är som det ska, svarar RCX-enheten på detta. Då upprepar programmet Lego.exe automatiskt förfrågan Alive igen (givetvis med $O + 8$) som bör leda till ytterligare ett svar från RCX-enheten. Det är alltså fråga om två Alive som bör resultera i var sitt svar.

   Eftersom det blir fyra ramar (förfrågan O, svar på O, förfrågan $O + 8$, svar på $O + 8$), blir det svårt att läsa av Hex-fältet. Tips: Varje ram börjar med 55FF00, som skrivs **55FF0** i Hex-fältet. Se också det kompletta ramformatet i figur 10. Inga data, D_i, använd för Alive-förfrågan eller Alive-svar.

   Klicka på knappen **Alive?** och anteckna innehållet (hexadecimala värden) för de fyra ramarna (1: Alive-förfrågan O, 2: Alive-svar på O, 3: Alive-förfrågan $O + 8$, 4: Alive-svar på O+8.)

	Header	Operation Code and Data	Checksum
1			
2			
3			
4			

b. Använd bilaga 1 med operationskoder. Anteckna de koder (O och $O + 8$) som används för **Alive**.

c. Anteckna de koder som används för svar till **Alive**, enligt bilaga 1.

d. Visa hur man beräknar 1-komplementet till operationskoden (**Alive**) i den första ramen. Tips: se sid. 7. Visa både formeln och värden på termerna, dvs. inte endast resultatet.

2. Det är också möjligt att skicka egna operationskoder till RCX-enheten. I programmet Lego.exe används fälten **Opcode** (kommando/förfrågan), **D1** (det första argumentet), **D2** (det andra argumentet), **D3** (det tredje argumentet) och **D4** (det fjärde argumentet). **Check** (checksumman) beräknas av Lego.exe, dvs. inget värde ska skrivas in av användaren. Operationskoden skickas till RCX-enheten med knappen **Sänd**.

   Tänk på att varje kommando/förfrågan har två koder (med skillnaden 8) och att man måste skifta mellan dessa. Om man inte växlar mellan dessa två koder, tar inte RCX-enheten emot kommandot/förfrågan.

   a. För att sända med de starkaste IR-signalerna ställ in RCX-enheten på **långt avstånd** till IR-tornet. Använd bilaga 1 för att ta reda på följande:

   - Kommandots namn

   - Hexadecimala koder för förfrågan och svar

   - Data (argumentet) för kort respektive långt avstånd

   Med inställning **långt avstånd** i RCX-enheten är det lämpligt att också ställa in **IR-tornet** på "långt avstånd". Se figur 3a.

   b. Visa hur man beräknar checksumman, *C*, i ramen som ställer in RCX-enheten på **långt avstånd**. (1-komplementet för C behöver inte visas.) Tips: se sid. 7 och tänk på att D_1 används, men inget annat D_i. Visa både formeln och värden på termerna, dvs. inte endast resultatet.

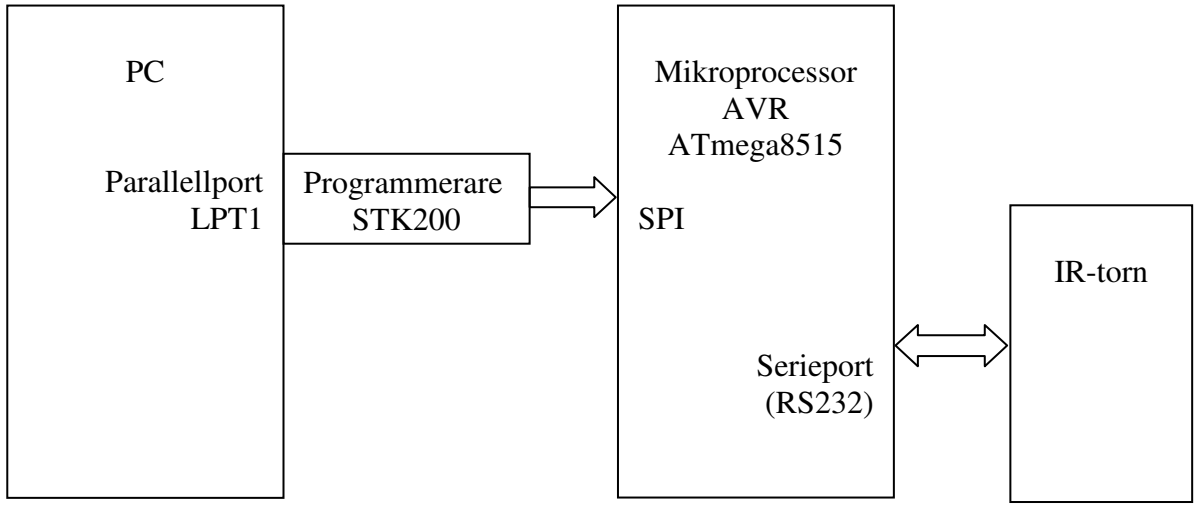
Uppgift B

(Eget arbete.)

Mikroprocessor

Du ska använda mikroprocessorn **AVR ATmega8515** från ATMEL. Det är en 8-bitars mikroprocessor i CMOS-teknik med RISC-arkitektur. Inom styr- och regler är detta ett vanligt förekommande dataformat. Med RISC-arkitektur blir sådana mikroprocessorer snabba. Programmet i mikroprocessorn ska i sin tur styra RCX-enheten. I mikroprocessorn finns ett flashminne på 8 KiB. Ett flashminne är ett snabbt **Electrical Erasable Programmable Read Only Memory** (EEPROM, E^2PROM).

För att kunna programmera flashminnet, finns ett **Serial Peripheral Interface** (SPI). Du kommer att använda parallellporten (LPT1) på en persondator för att sända program till flashminnet. För ändamålet krävs en speciell kabel med en s.k. programmerare som i detta fall är en STK200. Det finns också en traditionell serieport (RS232) på mikroprocessorn. Denna ansluter du med en kabel till IR-tornet.



Figur 12. Persondator, mikroprocessor och IR-transceiver.

Instruktion för att starta nytt projekt i Eclipse

1. Anslut en STK200-programmerare till datorn parallellport.
2. Koppla enligt Figur 12 i laborations-PM. För att kunna koppla ihop mikroprocessorn med IR-tornet, behövs en s.k. könbytare. Denna ansluts mellan kabeln från IR-tornet och RS232-porten på mikroprocessorn.
3. Anslut ström till mikroprocessorn från en adapter.
4. Öppna det grafiska filhanteringssystemet Files (ikonen på skrivbordet).
5. Gå till **Labbar**.
6. Dekomprimera (höger-klicka > Extract Here) arkivet **RCX-Lab.tar.gz**.
7. Döp om **RCX-Lab** till **lego**. (Namnet är viktigt för projekthantering i Eclipse).
8. Kopiera **lego** till **datorkom/workspace**. (Katalogen används för projekt av Eclipse.)

9. Starta Eclipse (ikonen på skrivbordet).
10. Skapa nytt projekt: File > New > **C project**
    Project Name: **lego** (Namnet är viktigt för matchning med 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]
11. Öppna upp katalogen **lego** och sedan också filen **lego.c**. (I detta läge kan programmet redigeras.)
12. Kompilera och länka: klicka på **hammaren** som finns på verktygsfältet.
    Öppna katalogen **Release** och kontrollera att den körbara filen **lego.hex** har skapats.
    Observera också den nyskapade **makefile**.
    Ladda ner **lego.hex** till MCU: klicka direkt på **AVR*** som finns på verktygsfältet.
13. Starta (On) RCX-enheten och starta om programmet i MCU genom att stänga av och sedan på med strömbrytaren.

    Stega menyn i programmet som finns i MCU med hjälp av switchen **SW.7**. Välj alternativet **Play sound no 2** med hjälp av switchen **SW.6**.

    Du använder följande switchar till MCU:
    SW.7 – stega i menyn som visas på LCD
    SW.6 – välj det alternativ som visas på LCD

    Tänk på att programmet inte är komplett än. Därför kommer SW.6 inte att ge val av alternativ.

14. Drag och släpp **legodemo.hex** till **Release**-katalogen. Ta bort **lego.hex**. Byt namn (höger-klicka > Rename) **legodemo.hex** till **lego.hex**. Ladda ner den nya **lego.hex** och provkör programmet i MCU igen. Nu ska **SW.6** kunna användas och därmed också kommunikationen med RCX-enheten.
15. Fortsätt med **B-uppgifterna** på nästa sida.

B-uppgifterna

I de följande uppgifterna 1 och 2 ska du komplettera filen **lego.c** så att dess kompilerade och länkade version **lego.hex** får samma funktioner som **legodemo.hex**.

1. Öppna filen **lego.c** från **Eclipse**.

   Skriv klart proceduren **SendFrame()** i **lego.c**. Denna ska kompletteras med sändning av huvud (**55_{hex} FF_{hex} 00_{hex}**), operationskod (**command**) och checksumma (**checksum**). Totalt ska

   $$55FF00O\overline{O}D_1\overline{D_1}D_2\overline{D_2}\cdots D_n\overline{D_n}C\overline{C}$$

   sändas för varje operationskod. Koden för sändning av data, D_i, är redan klar. Det finns kommentarer i källkoden som anger var kod ska skrivas in.

   Använd funktionen **USART_SendByte(*byte*)** för ändamålet. Om ***byte*** är ett värde, kan skrivsättet **0x*hh*** användas, där ***hh*** = två hexadecimala tecken, exempelvis 0xA8. (1 byte = 8 bitar = 2·4 bitar, dvs. två hexadecimala tecken ska användas.)

   Operationskoden, *O*, ges av variabeln **command**. Denna är ett av argumenten i anropet av **SendFrame()**. Det finns redan kod för **command + 8**. Kod för detta behöver alltså inte skrivas in. Tänk på att efter sändning av **command**, ska dess 1-komplement skickas med. Beräkning av 1-komplement i visas på sid. 6. Variabeln **command** har formatet 1 byte och denna passar utmärkt för sändning med funktionen **USART_SendByte(*byte*)**.

   Beräkningen av checksumman, *C*, är klar. Den ges av variabeln **checksum**. Du behöver endast skriva kod för att skicka **checksum** och dess 1-komplement. Checksumman sänds på liknande sätt som operationskoden.

   Kompilera, ladda och provkör

   Under provkörning ska alternativen **Play sound no 2**, **Set motor A pow**, **Motor A on** och **Motor A off** fungera.

2. Nu återstår kod för alternativet **Get battery power**. Sedan ska ditt legoprogram fungera som legodemo.

   Skriv klart proceduren **GetBattPower()** i **lego.c**. Denna ska kompletteras med sändning av huvud (**55_{hex} FF_{hex} 00_{hex}**), operationskod (**command**) och checksumma (**= command** eftersom data inte används). Det finns kommentarer i källkoden som anger var kod ska skrivas in. I detta fall ska

   $$55FF00O\overline{O}C\overline{C}$$

   sändas för varje begäran om batterispänning.

Kompilera, ladda och provkör.

Förklaring: Batterispänningen i enheten V beräknas med hjälp av svaret från RCX-enheten. Då används data D_1 och D_2 enligt formeln

$$U = \frac{256 \cdot D_2 + D_1}{1000}$$

Kod för beräkningen finns redan i **lego.c**.

I uppgifterna 3 och 4 ska du lägga till alternativ i menyn för annat ljud och annan motor (B eller C).

3. Lägg till ett annat ljud än alternativet **Play sound no. 2**. Du väljer själv något ljud. Glöm inte att den nya ljudtypen ska visas på mikroprocessorns LCD. Det finns kommentar i källkoden som anger var kod ska skrivas in.

   Kompilera, skriv och provkör.

4. Lägg till en annan motor i än alternativen **Set motor A pow**, **Motor A on** och **Motor A off**. Du kan välja motor **B** eller **C**. Glöm inte att det nya motorvalet ska visas på mikroprocessorns LCD. Det finns kommentar i källkoden som anger var kod ska skrivas in.

   Kompilera, ladda och provkör.

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

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

Operationskoder

Följande operationskoder beskrivs i denna bilaga:

Absolute value
Alive
Add to variable
And variable
Branch always far
Branch always near
Call subroutine
Clear message
Clear sensor value
Clear timer
Datalog next
Decrement loop counter far
Decrement loop counter near
Delete all subroutines
Delete all tasks
Delete firmware
Delete subroutine
Delete task
Divide variable
Get battery power
Get memory map
Get value
Get versions
Multiply variable
Or variable
Play sound
Play tone
Power off
Return from subroutine
Send message
Set datalog size
Set display
Set loop counter
Set message
Set motor direction
Set motor on/off
Set motor power
Set power down delay
Set program number
Set sensor mode
Set sensor type
Set time
Set transmitter range
Set variable
Sign variable
Start firmware download
Start subroutine download
Start task
Start task download
Stop all tasks
Stop task
Subtract from variable
Test and branch far
Test and branch near
Transfer data
Unlock firmware
Upload datalog
Wait

Absolute value

74/7c Request/Command

byte *index*	Destination variable index. 0..31.
byte *source*	Source type for operand. Only 0 and 2 allowed.
short *argument*	Argument for operand.

Compute the absolute value of the value specified by *source* and *argument* and store the result in variable *index*. The absolute value of the largest negative number, -32768, is defined to be the largest positive number, 32767.

83/8b Reply

void

Reply indicates success.

Alive

10/18 Request

void

Check whether or not the RCX is alive. If the PC receives a reply to this request, it assumes the RCX is alive and the connection is good.

e7/ef Reply

void

Reply indicates that the RCX is alive.

Add to variable

24/2c Request/Command

byte *index* Destination and first operand variable index. 0..31.
byte *source* Source type for second operand. Only 0 and 2 allowed.
short *argument* Argument for second operand.

Add the value specified by *source* and *argument* to the value of variable *index*, and store the result in variable *index*.

d3/db Reply

void

Reply indicates success.

And variable

84/8c Request/Command

byte *index* Destination and first operand variable index. 0..31.
byte *source* Source type for second operand. Only 0 and 2 allowed.
short *argument* Argument for second operand.

Compute the logical AND of the value specified by *source* and *argument* and the value of variable *index*, and store the result in variable *index*.

73/7b Reply

void

Reply indicates success.

Branch always far

72/xx Command

ubyte *offset* Offset for computing branch target address.
ubyte *extension* Extension to offset.

Branch to the target address specified by *offset* and *extension*.

If bit 0x80 of *offset* is 0, the target address computed as:

address of *offset* + *offset* + 128 × *extension*.

Otherwise, the target address computed as:

address of *offset* - *offset* - 128 × *extension* + 128.

Branch always near

27/xx Command

ubyte *offset* Offset for computing branch target address.

Branch to the target address specified by *offset*.

If bit 0x80 of *offset* is 0, the target address computed as:

address of *offset* + *offset*.

Otherwise, the target address computed as:

address of *offset* - *offset* + 128.

Call subroutine

17/xx Command

byte *subroutine* Index of subroutine to call. 0..7.

Call the subroutine with index *subroutine*. If the subroutine is not defined, do nothing.

The RCX only supports one subroutine return address per task. If one subroutine calls another subroutine, execution of all tasks stops when the original subroutine returns.

Clear message

90/xx Command

void

Clear the message buffer by setting it to zero. The message buffer stores a single byte and allows for communication between multiple RCX units.

See also: **send message**, **set message**.

Clear sensor value

d1/d9 Request/Command

byte *sensor* Index of sensor to clear. 0..2.

Clear the counter associated with the specified sensor by setting it to a value of zero.

26/2e Reply

void

Reply indicates success.

Clear timer

a1/a9 Request/Command

byte *timer* Index of timer to clear. 0..3.

Clear the specified timer by setting it to a value of zero.

56/5e Reply

void

Reply indicates success.

Datalog next

62/6a Request/Command

byte *source* Source type for next datalog entry. Only 0, 1, 9, and 14 allowed.
byte *argument* Argument for next datalog entry.

Set the next datalog entry to the value specified by *source* and *argument*. If the datalog is full, leave it unmodified.

95/9d Reply

byte *errorcode* Return value.

A return value of 0 indicates success, while a return value of 1 indicates that the datalog was full.

Decrement loop counter far

92/xx Command

ushort *offset* Offset for computing branch target address.

Decrement the current loop counter, then, if the loop counter is less than zero, pop the loop counter stack and branch to the target address specified by *offset*.

The branch target address is computed as:

address of first byte of *offset* + *offset*.

Note that *offset* is unsigned. Backward branching is not allowed with this operation.

Decrement loop counter near

37/xx Command

ubyte *offset* Offset for computing branch target address.

Decrement the current loop counter, then, if the loop counter is less than zero, pop the loop counter stack and branch to the target address specified by *offset*.

The branch target address is computed as:

address of *offset* + *offset*.

Note that *offset* is unsigned. Backward branching is not allowed with this operation.

Delete all subroutines

70/78 Request/Command

void

Delete all subroutines belonging to the current program.

87/8f Reply

void

Reply indicates success.

Delete all tasks

40/48 Request/Command

void

Stop execution and delete all tasks belonging to the current program.

b7/bf Reply

void

Reply indicates success.

Delete firmware

65/6d Request/Command

byte *key*[5] Key. Must be {1,3,5,7,11}.

If *key* is valid, stop execution and delete the firmware. Otherwise, do nothing. The key prevents the firmware from being accidentally erased.

Before the firmware may be replaced, it must be deleted.

92/9a Reply

void

Reply indicates success.

Delete subroutine

c1/c9 Request/Command

byte *subroutine* Index of subroutine to delete. 0..7.

Delete the specified subroutine. If the subroutine is currently being executed, the related task is stopped.

36/3e Reply

void

Reply indicates success.

Delete task

61/69 Request/Command

byte *task* Index of task to delete. 0..9.

Delete the specified task. If the task is currently running, execution of all tasks stops.

96/9e Reply

void

Reply indicates success.

Divide variable

44/4c Request/Command

byte *index* Destination and first operand variable index. 0..31.
byte *source* Source type for second operand. Only 0 and 2 allowed.
short *argument* Argument for second operand.

Divide the value of variable *index* by the value specified by *source* and *argument*, and store the result in variable *index*. If the *source* and *argument* specify a denominator of zero, the variable *index* is left unchanged.

b3/bb Reply

void

Reply indicates success.

Get battery power

30/38 Request

void

Request the battery voltage from the RCX.

c7/cf Reply

ushort *millivolts* Battery voltage.

Reply indicates the current voltage of the RCX battery, in mV. A fresh set of alkaline batteries typically has a reading of around 9.3V.

Get memory map

20/28 Request

void

Request the memory map from the RCX.

d7/df Reply

ushort *map*[94] Memory map.

Reply contains the user program memory map of the RCX. The memory map is an array of 94 16-bit big endian addresses, organized as follows:

Index	Description
0-7	Starting addresses, program 0, subroutines 0-7
8-15	Starting addresses, program 1, subroutines 0-7
16-23	Starting addresses, program 2, subroutines 0-7
24-31	Starting addresses, program 3, subroutines 0-7
32-39	Starting addresses, program 4, subroutines 0-7
40-49	Starting addresses, program 0, tasks 0-9
50-59	Starting addresses, program 1, tasks 0-9
60-69	Starting addresses, program 2, tasks 0-9
70-79	Starting addresses, program 3, tasks 0-9
80-89	Starting addresses, program 4, tasks 0-9
90	First datalog address
91	Next datalog address
92	First free address
93	Last valid address

Get value

12/1a Request

byte *source* Source type for value. Sources 2 and 4 not allowed.
byte *argument* Argument for value.

Request the value specified by *source* and *argument*.

e5/ed Reply

short *value* Return value.

Reply contains the requested value.

Get versions

15/1d Request

byte *key*[5] Key. Must be {1,3,5,7,11}.

Request the ROM and firmware versions from the RCX. If *key* is not valid, no reply is sent.

e2/ea Reply

short *rom*[2] ROM version number.
short *firmware*[2] Firmware version number.

Reply contains the ROM and firmware version numbers. Each version number is composed of two big endian shorts; the first is the major version number and the second is the minor version number.

The ROM always has major and minor version numbers 3 and 1. The firmware shipped with the Robotics Invention System has major and minor version numbers 3 and 9. When no firmware is installed, both firmware version numbers are 0.

Multiply variable

54/5c Request/Command

byte *index* Destination and first operand variable index. 0..31.
byte *source* Source type for second operand. Only 0 and 2 allowed.
short *argument* Argument for second operand.

Multiply the value of variable *index* by the value specified by *source* and *argument*, and store the result in variable *index*.

a3/ab Reply

void

Reply indicates success.

Or variable

94/9c Request/Command

byte *index* Destination and first operand variable index. 0..31.
byte *source* Source type for second operand. Only 0 and 2 allowed.
short *argument* Argument for second operand.

Compute the logical OR of the value specified by *source* and *argument* and the value of variable *index*, and store the result in variable *index*.

63/6b Reply

void

Reply indicates success.

Play sound

51/59 Request/Command

byte *sound* Sound type. 0..5.

Play the specified sound.

There are six avaiable sound types:

Index	Description
0	Blip
1	Beep beep
2	Downward tones
3	Upward tones
4	Low buzz
5	Fast upward tones

a6/ae Reply

void

Reply indicates success.

Play tone

23/2b Request/Command

short *frequency* Tone frequency.
byte *duration* Tone duration.

Play the sound specified by *frequency* and *duration*. The tone frequency is measured in Hz, and the tone duration is measured in 1/100ths of a second.

d4/dc Reply

void

Reply indicates success.

Power off

60/68 Request/Command

void

Turn off the RCX. If the power down delay is zero, do nothing.

97/9f Reply

void

Reply indicates success.

Return from subroutine

f6/xx Command

void

Return from subroutine. This opcode is intended to be used in the middle of a subroutine, since a return from subroutine opcode is automatically added to every subroutine that is downloaded.

Send message

b2/xx Command

byte *source* Source type for message. Only 0 and 2 allowed.
byte *argument* Argument for message.

Send the value specified by *source* and *argument* to other RCX units. The value is sent by broadcasting a **set message** request over the infrared link.

Set datalog size

52/5a Request/Command

short *size* New datalog size.

Allocate and initialize a datalog with space for *size* data entries. A single extra entry is always allocated to hold the current size of the datalog, which is initialized to one since this size value is stored as the first entry in the datalog.

a5/ad Reply

byte *errorcode* Return value.

A return value of 0 indicates success, while a return value of 1 indicates that there is insufficient memory to allocate a datalog of the requested size.

Set display

33/3b Request/Command

byte *source* Source type for device. Only 0 and 2 allowed.
short *argument* Argument for device.

Display the input/output value associated with the device specified by *source* and *argument*. Valid devices are:

Index	Description
0	Watch
1	Sensor 1
2	Sensor 2
3	Sensor 3
4	Motor A
5	Motor B
6	Motor C

This operation controls the same functionality as the View button on the face of the RCX.

c4/cc Reply

void

Reply indicates success.

Set loop counter

82/xx Command

byte *source* Source type for counter value. Only 0, 2, and 4 allowed.
byte *argument* Argument for counter value.

Push the loop counter stack, then set the topmost loop counter to the value specified by *source* and *argument*. There are four loop counters. If more than four loops are nested, the loop counter stack is not pushed before the topmost value is set.

Set message

f7/xx Request/Command

byte *message* Message value.

Set the value of the message buffer to *message*. This is the only request with no matching reply.

Set motor direction

e1/e9 Request/Command

byte *code* Bit field to specify a direction and up to three motors.

Set the direction of the motors according to *code*. The bits of *code* have the following meanings:

Bit	Description
0x01	Modify direction of motor A
0x02	Modify direction of motor B
0x04	Modify direction of motor C
0x40	Flip the directions of the specified motors
0x80	Set the directions of the specified motors to forward

If both bit 0x40 and bit 0x80 are 0, the directions of the specified motors are set to reverse. If both bit 0x40 and bit 0x80 are 1, the directions of the specified motors are flipped.

16/1e Reply

void

Reply indicates success.

Set motor on/off

21/29 Request/Command

byte *code* Bit field to specify an on/off state and up to three motors.

Set the on/off state of the motors according to *code*. The bits of *code* have the following meanings:

Bit	Description
0x01	Modify on/off state of motor A
0x02	Modify on/off state of motor B
0x04	Modify on/off state of motor C
0x40	Turn off the specified motors
0x80	Turn on the specified motors

If both bit 0x40 and bit 0x80 are 0, the specified motors are set to float, which allows the shafts of the specified motors to spin freely. Setting both bit 0x40 and bit 0x80 to 1 turns on the specified motors.

d6/de Reply

void

Reply indicates success.

Set motor power

13/1b Request/Command

byte *motors* Bit field to specify up to three motors.
byte *source* Source type for power level. Only 0, 2, and 4 allowed.
byte *argument* Argument for power level.

Set the power level of the motors specified by *motors* to the value specified by *source* and *argument*. The bits of *motors* have the following meanings:

Bit	Description
0x01	Modify power level of motor A
0x02	Modify power level of motor B
0x04	Modify power level of motor C

Valid power levels are between 0 and 7, inclusive. Other power levels are ignored.

e4/ec Reply

void

Reply indicates success.

Set power down delay

b1/b9 Request/Command

byte *minutes* Power down delay.

Set the power down delay (0..99) to the specified value, which is measured in minutes. A power down delay of 0 instructs the RCX to remain on indefinitely and causes the **power off** opcode to be ignored.

46/4e Reply

void

Reply indicates success.

Set program number

91/99 Request/Command

byte *program* Program number. 0..4.

Stop execution and set the current program number to the specified value.

66/6e Reply

void

Reply indicates success.

Set sensor mode

42/4a Request/Command

byte *sensor* Index of sensor to modify. 0..2.
byte *code* Packed sensor slope and mode.

Set the slope and mode of sensor number *sensor* to the value specified by *mode*, and clear that sensor's value. The bits of *mode* are split into two portions. Bits 0-4 contain a slope value in 0..31, while bits 5-7 contain the mode, 0..7. The eight modes, which control the value returned by the sensor, are:

Mode	Name	Description
0	Raw	Value in 0..1023.
1	Boolean	Either 0 or 1.
2	Edge count	Number of boolean transitions.
3	Pulse count	Number of boolean transitions divided by two.
4	Percentage	Raw value scaled to 0..100.
5	Temperature in °C	1/10ths of a degree, -19.8..69.5.
6	Temperature in °F	1/10ths of a degree, -3.6..157.1.
7	Angle	1/16ths of a rotation, represented as a signed short.

The slope value controls 0/1 detection for the three boolean modes. A slope of 0 causes raw sensor values greater than 562 to cause a transition to 0 and raw sensor values less than 460 to cause a transition to 1. The hysteresis prevents bouncing between 0 and 1 near the transition point. A slope value in 1..31, inclusive, causes a transition to 0 or to 1 whenever the difference between consecutive raw sensor values exceeds the slope. Increases larger than the slope result in 0 transitions, while decreases larger than the slope result in 1 transitions. Note the inversions: high raw values correspond to a boolean 0, while low raw values correspond to a boolean 1.

b5/bd Reply

void

Reply indicates success.

Set sensor type

32/3a Request/Command

byte *sensor* Index of sensor to modify. 0..2.
byte *type* Sensor type. 0..4.

Set the type of sensor number *sensor* to *type*, and set the mode of that sensor to a default value. Valid types and their default modes are:

Type	Description	Default Mode
0	Raw	Raw
1	Touch	Boolean
2	Temperature	Temperature in °C
3	Light	Percentage
4	Rotation	Angle

c5/cd Reply

void

Reply indicates success.

Set time

22/2a Request/Command

byte *hours* Current hour. 0..23.
byte *minutes* Current minute. 0..59.

Set the current time to *hours* and *minutes*.

d5/dd Reply

void

Reply indicates success.

Set transmitter range

31/39 Request/Command

byte *range* Transmitter range. 0 or 1.

Set the transmitter range. 0 indicates short range, 1 indicates long range. Other values are ignored.

c6/ce Reply

void

Reply indicates success.

Set variable

14/1c Request/Command

byte *index* Destination and first operand variable index. 0..31.
byte *source* Source type for second operand. All sources are allowed.
short *argument* Argument for second operand.

Set the value of variable *index* to the value specified by *source* and *argument*.

e3/eb Reply

void

Reply indicates success.

Sign variable

64/6c Request/Command

byte *index* Destination and first operand variable index. 0..31.
byte *source* Source type for second operand. Only 0 and 2 allowed.
short *argument* Argument for second operand.

Set the value of variable *index* to 0 if the value specified by *source* and *argument* is equal to 0, 1 if the value specified by *source* and *argument* is positive, or -1 if the value specified by *source* and *argument* is negative.

93/9b Reply

void

Reply indicates success.

Start firmware download

75/7d Request

short *address* Firmware entry address. Typically 0x8000.
short *checksum* Firmware checksum. Typically 0xc278.
byte *unknown* Unknown or unused value. Always 0.

Prepare the RCX for a firmware download starting at address 0x8000. The checksum is computed by taking the sum of all firmware bytes modulo 2^{16}. The specified address is used as the firmware entry point when the firmware is unlocked.

If firmware is already installed, this request is ignored and no response is sent.

82/8a Reply

byte *errorcode* Return value.

A return value of 0 indicates success, while any other return value indicates failure.

Start subroutine download

35/3d Request

byte *unknown* Unknown or unused value. Always 0.
short *subroutine* Subroutine index. 0..7.
short *length* Subroutine length.

Prepare the RCX for a download of subroutine number *subroutine* of the current program. Space for *length* bytes is allocated in the memory map by moving other data as needed.

c2/ca Reply

byte *errorcode* Return value.

A return value of 0 indicates success, a return value of 1 indicates that there is insufficient memory for a subroutine of the requested size, and a return value of 2 indicates that the subroutine index was invalid.

Start task

71/79 Request/Command

byte *task* Index of task to start. 0..9.

Start the specified task, or restart it if it is currently active. If the specified task is not defined, nothing happens.

86/8e Reply

void

Reply indicates success.

Start task download

25/2d Request

byte *unknown* Unknown or unused value. Always 0.
short *task* Task index. 0..9.
short *length* Task length.

Prepare the RCX for a download of task number *task* of the current program. Space for *length* bytes is allocated in the memory map by moving other data as needed.

d2/da Reply

byte *errorcode* Return value.

A return value of 0 indicates success, a return value of 1 indicates that there is insufficient memory for a task of the specified size, and a return value of 2 indicates that the task index was invalid.

Stop all tasks

50/58 Request/Command

void

Stops execution.

a7/af Reply

void

Reply indicates success.

Stop task

81/89 Request/Command

byte *task* Index of task to stop. 0..9.

Stops execution of the specified task.

76/7e Reply

void

Reply indicates success.

Subtract from variable

34/3c Request/Command

byte *index* Destination and first operand variable index. 0..31.
byte *source* Source type for second operand. Only 0 and 2 allowed.
short *argument* Argument for second operand.

Subtract the value specified by *source* and *argument* from the value of variable *index*, and store the result in variable *index*.

c3/cb Reply

void

Reply indicates success.

Test and branch far

95/xx Command

byte *opsrc1* Packed operator and source type for first value. Sources 4 and 8 not allowed.
byte *src2* Source type for second value. Sources 2, 4, and 8 not allowed.
short *arg1* Argument for first value.
byte *arg2* Argument for second value.
short *offset* Offset for computing branch target address.

Compare two values against one another and branch if the comparison is true. Bits 0-3 of *opsrc1* contain the source for the first value, while *arg1* contains the argument for the first value. The second value is specified by *src2* and *arg2*. The comparison operator, which is stored in bits 6-7 of *opsrc1*, has a value in 0..3. The meanings of these values are:

Value	Description
0	First value less than or equal to second value
1	First value greater than or equal to second value
2	First value not equal to second value
3	First value equal to second value

The branch target address is computed as:

address of first byte of *offset* + *offset*.

Note that *offset* is signed. Backward branching is allowed with this operation.

Test and branch near

85/xx Command

byte *opsrc1* Packed operator and source type for first value. Sources 4 and 8 not allowed.
byte *src2* Source type for second value. Sources 2, 4, and 8 not allowed.
short *arg1* Argument for first value.
byte *arg2* Argument for second value.
byte *offset* Offset for computing branch target address.

Compare two values against one another and branch if the comparison is true. Bits 0-3 of *opsrc1* contain the source for the first value, while *arg1* contains the argument for the first value. The second value is specified by *src2* and *arg2*. The comparison operator, which is stored in bits 6-7 of *opsrc1*, has a value in 0..3. The meanings of these values are:

Value	Description
0	First value less than or equal to second value
1	First value greater than or equal to second value
2	First value not equal to second value
3	First value equal to second value

The branch target address is computed as:

address of first byte of *offset* + *offset*.

Note that *offset* is signed. Backward branching is allowed with this operation.

Transfer data

45/4d Request

short *index*	Sequence number of data block.
short *length*	Length of data block.
byte *data*[*length*]	Data block.
byte *checksum*	Checksum of data block.

Download block number *index*, containing *length* data bytes, to the RCX. The bytes are stored as required by the most recent **start firmware download, start subroutine download**, or **start task download** operation.

Block sequence numbers start at 1 and increase by one with each successive block transferred. The special sequence number 0 indicates the last block of a transfer. Once this sequence number is received by the RCX, another start download operation is required before additional data blocks may be transferred.

The checksum is computed by taking the sum of all bytes in the data block modulo 256. It is only checked for packets that have a non-zero sequence number.

Note that, for task and subroutine downloads, the RCX does not check that the total number of bytes sent was equal to the amount of space allocated by the corresponding start download operation. If too few bytes are sent, a portion of the downloaded task or subroutine will contain invalid code; if too many bytes are sent, other tasks and subroutines may become corrupted.

b2/ba Reply

byte *errorcode* Return value.

A return value of 0 indicates success, a return value of 3 indicates a block checksum failure, a return value of 4 indicates a firmware checksum error, and a return value of 6 indicates an invalid or missing download start. If the block sequence number is out of order, no reply is sent.

When block checksum error number 3 is received, the PC may either retransmit the erroneous block or restart the download. A block with index 0 cannot be retransmitted. Moreover, a bug in the ROM requires that the PC increment its block sequence number when retransmitting firmware data in response to a block checksum error.

During a firmware download, duplicate transfer data requests are handled incorrectly. If the PC does not receive a transfer data reply, block sequence numbers might get out of sync.

Unlock firmware

a5/ad Request

byte *key*[5] Key. Must be {76,69,71,79,174} = {"LEGO®"}.

Activate the firmware after a successful download.

52/5a Reply

byte *data*[25] Return value. Always {"Just a bit off the block!"}.

Reply indicates success.

Upload datalog

a4/ac Request

short *first* Index of first entry to upload.
short *count* Number of entries to upload.

Upload *count* datalog entries, starting with entry number *first*.

The datalog entry with index 0 always contains the current size of the datalog, which is guaranteed to be at least one since the current size entry is considered to be part of the datalog. After the current size entry are *size* data entries, where *size* is specified with the **set datalog size** operation and is initially zero.

It is an error to read an entry outside the valid range of the datalog.

53/5b Reply

dlrec *data*[*length*] Requested datalog entries.

Reply contains the requested datalog entries, stored as an array of *length* dlrec duples, where *length* was specified as the *count* in the corresponding request. Each dlrec is organized as follows:

byte *type* Type of datalog entry.
short *value* Value of datalog entry.

The *type* of each dlrec specifies the appropriate interpretation of the corresponding *value*. Valid types are:

Type	Description
0xff	Current datalog size
0x00-0x1f	Variable value (source 0, variables 0..31)
0x20-0x23	Timer value (source 1, timers 0..3)
0x40-0x42	Sensor reading (source 9, sensors 0..2)
0x80	Clock reading (source 14)

If an error occurs while reading the datalog, *length* is set to zero.

Wait

43/xx Command

byte *source* Source type for delay. Only 0, 2, and 4 allowed.
short *argument* Argument for delay.

Wait for the delay specified by *source* and *argument*. The delay is in 1/100ths of a second.

Sources and Arguments

This section describes the available sources and arguments. Sources are like addressing modes. They specify where and how to get certain operand values. There are 16 sources available, of which 13 apply to the RCX:

Source	Name	Argument	Description
0	Variable	Variable index, 0..31.	Returns value of specified variable.
1	Timer	Timer index, 0..3.	Returns value of specified timer, in 1/100ths of a second.
2	Immediate	Immediate value.	Returns specified immediate value.
3	Motor State	Motor index, 0..2.	Returns state of specified motor. See below.
4	Random	Maximum value.	Returns random value, 0..max.
5	*Reserved*	*N/A*	*Cybermaster only.*
6	*Reserved*	*N/A*	*Cybermaster only.*
7	*Reserved*	*N/A*	*Cybermaster only.*
8	Current Program	Ignored.	Returns current program number.
9	Sensor Value	Sensor index, 0..2.	Returns value of specified sensor.
10	Sensor Type	Sensor index, 0..2.	Returns type of specified sensor.
11	Sensor Mode	Sensor index, 0..2.	Returns mode of specified sensor.
12	Raw Sensor Value	Sensor index, 0..2.	Returns raw value of specified sensor, 0..1023.
13	Boolean Sensor Value	Sensor index, 0..2.	Returns boolean value of specified sensor, 0..1.
14	Clock	Must be 0.	Returns minutes since power on.
15	Message	Must be 0.	Returns value of message buffer.

Motor state is encoded as a single byte. Bits 0-2 contain the motor power, 0..7. The remaining bits are used as follows:

Bit	Description	Notes
0x08	Forward flag	0 if forward, 1 if reverse.
0x40	Off flag	1 if off.
0x80	On flag	1 if on.

If both bit 0x40 and bit 0x80 are 0, the specified motor is set to float.

```
/*
**  Namn:            lego.c
**  Version:         mega
**  Syfte:           studera protokollet för Lego RCX
**  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 <avr/wdt.h>
#include <util/delay.h>


/* Definitioner för LCD */
#define LINE1 1            // Början av rad 1 på LCD
#define LINE2 168          // Början av rad 2 på LCD


/* 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 2400  // 2400 baud för RCX
#define BAUD_PRESCALE ((F_CPU/(USART_BAUDRATE * 16UL)) - 1) // Bestäm skalfaktor för RCX


/* Globala variabler */
unsigned char add8 = 0;     // Används i SendFrame och GetBattPower för att kunna sända O,
O + 8, O, ...


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);
    //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)
{
     /* Rensa RS232-bufferten */

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


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

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


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

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

```
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 marginal
      */

     _delay_ms(10);
}


void LCD_Intro(unsigned char instr)
{
     /* Radbyte på LCD */
    char  i, j;

    i = instr;
    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 */

    LCD_Delay();
    LCD_Delay();
    LCD_Delay();
    DDRC = 0xFF;  // Använd Port C på STK200 till output för LCD
    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);  // Skicka tecken för tecken till LCD
    }
}
```

```
void LCD_Clean(unsigned char line)
{
     /* Rensa 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 tal (eller positiva heltal < 256) omvandlas till tecken i en lista
        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'};

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

     /* Omvandling 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]);  // Skicka tecknen till LCD
}


void MenuDelays(int time)
{
     /* 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
      * Värdet 4 ger marginal
      */

     if(time == 4)
                _delay_ms(4);  // Lång fördröjning
     else
                _delay_ms(1);  // Kort fördröjning
}
```

```
void SendFrame(unsigned char command, unsigned char *parameters)
{
     /* Skapa och sänd en RCX-ram */
    unsigned char i = 0, checksum = 0;

    /* Sätt "+ 8" varannan byte */
    if(add8!= 0)
    {
     command = command + 0x08;
        add8= 0;  // Jämt antal ramar har strax skickats
    }
    else
    {
     add8= 1;  // Udda antal ramar har strax skickats
    }

    /* Övning B:1 - Plats för att skicka header och command (operationskod) */
    /* Sänd header */

    /* Sänd command och dess 1-komplement */

    checksum = command;

    /* Sänd parameters */
    while(parameters[i] > 0)  // Sänd parameters och respektive 1-komplement
    {
     USART_SendByte(parameters[i]);
        USART_SendByte(0xFF - parameters[i]);
        checksum = checksum + parameters[i];
        i++;
    }

    /* Övning B:1 -  Plats för att skicka checksum */
    /* Sänd checksum och dess 1-komplement */

}


void GetBattPower(void)
{
     /* Sänd GET BATTERY POWER */
    unsigned char command, checksum, H1 = 0, H2 = 0, H3 = 0, D1 = 0, D2 = 0, dummy;
    int U;
    char u[8];

    LCD_Clean(LINE2);                    // Rensa LCD-rad 2 och gå till början av den

    /* Sätt "+ 8" varannan byte */
    if(add8!= 0)
    {
     command = 0x30 + 0x08;
        add8= 0;  // Jämt antal ramar har strax skickats
    }
    else
    {
        command = 0x30;
        add8= 1;  // Udda antal ramar har strax skickats
    }

    checksum = command;
```

```
    /* Övning B:2 - Plats för att skicka header, command (operationskod) och checksum */
    /*Sänd header */

    /* Sänd command och dess 1-komplement */

    /* Sänd checksum och dess 1-komplement */

    /* Läs svar */
    wdt_enable(7);
    // Slå på watchdog: körbart även om RS232-kommunikation falerar

    H1 = USART_ReceiveByte();  // Läs byte på RS232-port
    H2 = USART_ReceiveByte();  // Läs byte på RS232-port
    H3 = USART_ReceiveByte();  // Läs byte på RS232-port
    while((H1 != 0x55)||(H2 != 0xFF)||(H3 != 0x00))  // Sök efter headern (H1, H2, H3)
    {
        H1 = H2;
        H2 = H3;
        H3 = USART_ReceiveByte();  // Läs nästa byte på RS232-port
    }
    dummy = USART_ReceiveByte();   // Hoppa över svarskoden code (S)
    dummy = USART_ReceiveByte();   // Hoppa över 1-komplementet (~S)
    D1 = USART_ReceiveByte();      // Läs första data (D1)
    dummy = USART_ReceiveByte();   // Hoppa över 1-komplementet (~D1)
    D2 = USART_ReceiveByte();      // Läs andra data (D2)
    dummy = USART_ReceiveByte();   // Hoppa över 1-komplementet (~D2)
    dummy = USART_ReceiveByte();   // Hoppa över checkcum (C)
    dummy = USART_ReceiveByte();   // Hoppa över 1-komplementet (~C)
    u[7] = dummy;                   // Enbart för att slippa kompileringvarning om dummy
    wdt_disable();                 // Stäng av watchdog

    /* Beräkna batterispänning och skicka till LCD */
    U = 256*D2 + D1;               // Batterispänningen i mV, [0, 99999]
    u[0] = (int) U/10000;                                      // Tiotals V
    u[1] = (int) (U - u[0]*10000)/1000;                        // Entals V
    u[3] = (int) (U - u[0]*10000 - u[1]*1000)/100;             // Tiondels V
    u[4] = (int) (U - u[0]*10000 - u[1]*1000 - u[3]*100)/10;   // Hundradels V
    if (u[0] != 0)
      u[0] = u[0] + 48; // ACSII för tiotals V (Annars, dvs om 0, ersätt med mellanslag)
    else
      u[0] = ' ';
    u[1] = u[1] + 48;              // ASCII för entals V
    u[2] = '.';                    // Decimalpunkt
    u[3] = u[3] + 48;              // ASCII för tiondels V
    u[4] = u[4] + 48;              // ACSII för hundradels V
    u[5] = ' ';                    // Mellanslag
    u[6] = 'V';                    // Enheten V
    u[7] = '\0';                   // Nulltecken (sista elementet)
    LCD_StrOut(&u[0]);             // Skicka teckensträng till LCD
}


void Menu(void)
{
     /* Programmets huvudmeny */
     unsigned char i = 0, *b;

     /* Evighetsloop nr 2 */
    while(1)
    {
     while(bit_is_set(PIND,7));             // Om SW.7 har tryckts ner, kort fördröjning
```

```
 MenuDelays(1);
i++;                                // Stega framåt i programmets meny
switch(i)
{
    case 1:
        LCD_Clean(LINE2);              // Rensa LCD-rad 2 och gå till början av den
        LCD_StrOut("Get batt power");  // Skriv alternativet på LCD-rad 2
        while(bit_is_set(PIND,7))
        {                              // Så länge SW.7 är satt
                  if(bit_is_clear(PIND,6))
                  {                            // Och, om också SW.6 har varit satt
                            GetBattPower(); // GET BATTERY POWER
                            MenuDelays(4);  // Lång fördröjning
                  }
        }
    break;

    case 2:
        LCD_Clean(LINE2);
         LCD_StrOut("Play sound 2");
         while(bit_is_set(PIND,7))
         {
                  if(bit_is_clear(PIND,6))
                  {
                            b = (unsigned char*) "\x02";  // Ljudtyp nr 2
                            SendFrame(0x51, b);           // PLAY SOUND
                  }
         }
    break;

    case 3:
        LCD_Clean(LINE2);
        LCD_StrOut("Set motor A pow");
        while(bit_is_set(PIND,7))
        {
                  if(bit_is_clear(PIND,6))
            {
                            b = (unsigned char*) "\x01\x02\x07";
                                // Motor A, Level 2, Power 7
                            SendFrame(0x13, b);  // SET MOTOR POWER
                            LCD_Clean(LINE2);
                            LCD_StrOut("Motor A is set");
                            MenuDelays(4);
            }
        }
    break;

    case 4:
        LCD_Clean(LINE2);
        LCD_StrOut("Motor A on");
        while(bit_is_set(PIND,7))
        {
                  if(bit_is_clear(PIND,6))
            {
                            b = (unsigned char*) "\x81";
                                 // Modify/turn on motor A
                            SendFrame(0x21, b);  // SET MOTOR ON
                            MenuDelays(4);
            }
        }
    break;
```

```
            case 5:
                LCD_Clean(LINE2);
                LCD_StrOut("Motor A off");
                while(bit_is_set(PIND,7))
                {
                            if(bit_is_clear(PIND,6))
                    {
                                    b = (unsigned char*) "\x41";
                                        // Modify/turn off motor A
                                    SendFrame(0x21, b);  // SET MOTOR ON
                                    MenuDelays(4);
                    }
                }
            break;

            /* Övning B:3 - Plats för att lägga till ljud */

            /* Övning B:4 - Plats för att lägga till motor */

            default:
                LCD_Clean(LINE2);  // Rensa LCD-rad 2 och gå till början av den
                LCD_StrOut("Press SW.7");  // Skriv (igen) på LCD-rad 2
                    i = 0;                 // Nollställ valt case
        }
    }
}


int main(void)
{
     /* Initieringar */
    USART_Init();   // Initiera USART
    USART_Flush();  // Rensa RS232-bufferten
    LCD_Init();     // Initiera LCD
                    // Använd Port A på STK200-kortet för NC
    DDRB = 0xFF;    // Använd Port B på STK200-kortet för LED:s (data output,
                    // obs STK200 tre olika kontakttyper)
                    // Använd Port C på STK200-kortet för LCD (data output,
                    // sätts i LCD_Init()
    DDRD = 0x00;    // Använd Port D på STK200-kortet för switcharna (data input)
                    // Använd Port E på STK200-kortet för NC

     /* Visa text på LCD */
     LCD_Clean(LINE1);            // Rensa LCD-rad 1 och gå till början av den
     LCD_StrOut("LEGO RCX I/O");  // Skriv titeln på LCD-rad 1
     LCD_Clean(LINE2);           // Rensa LCD-rad 2 och gå till början av den
     LCD_StrOut("Press SW.7");    // Skriv på LCD-rad 2


     /* Evighetsloop nr 1 */
     while(1)
     {
         Menu();          // Hantera programmets meny
          MenuDelays(1);  // Kort fördröjning
     }
     return 0;
}
```