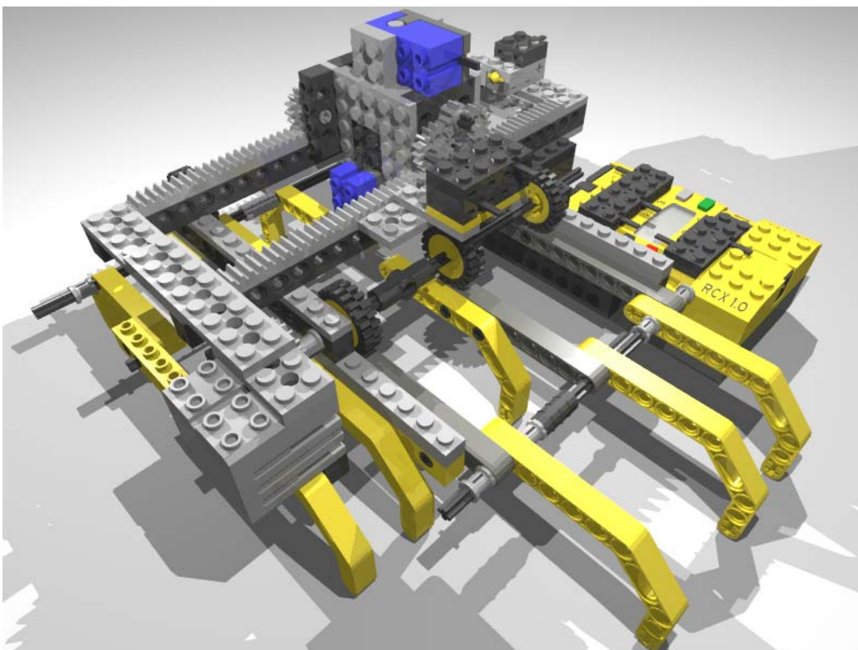
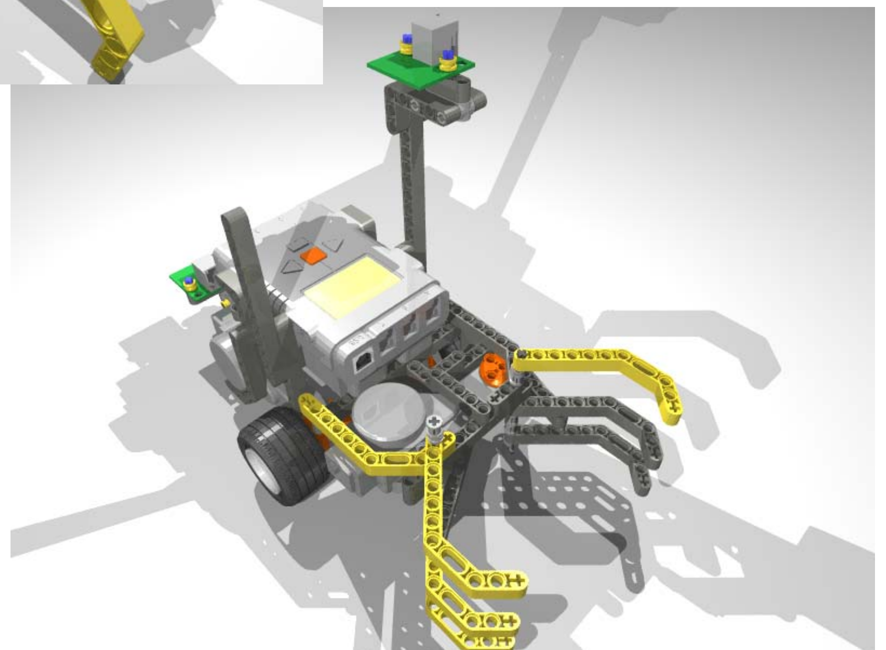
KTH Teknik och hälsa, Campus Telge　　　　Mekatronik Projekt

Scanbot & Servantbot





Södertälje 2008

PROJEKTRAPPORT

Skola:	STH
Tema:	Mekatronik projekt
Titel:	Scanbot & Servantbot
Grupp:	ERO C
Deltagare:	Jesper Stockenstrand Tomas Nordin Percy Villegas Tello Petter Jangenäs Fredrik Weng
Handledare:	Erik starbäck
Datum.	2008-05-12

Sammanfattning

I elektroteknik programmet, med inriktning robotik och mekatronik, vid KTH Campus Telge genomförs under första årskursen ett mekatronikprojekt. Detta projekt består i att bygga och programmera en robot bestående av minst två enheter baserade på Lego Mindstorms. Den ena i roboten ingående enheten skall baseras på en RCX-brick och den andra på en NXT-brick. Vidare ställs som krav att dessa två enheter på något sätt skall kommunicera med varandra.

Vi har som projekt valt att bygga en inläsningsenhet, baserad på en RCX-brick, som läser kommandon i form av binärkod. De kommandon som lästs in av inläsningsenheten skickas sedermera över till en NXT-brick baserad robot som utför dessa.

Inledningsvis genomfördes en faktainsamlig som syftade till att underlätta det kommande arbetet med konstruktion, montering och programmering. Därefter delades projektet upp på de tre delarna inläsningsenhet, robot och kommunikation. Gruppen delades och arbetade på tre fronter med de olika delarna. Slutligen sammanfogades de tre projektdelarna till ett alster och finjusterades till slutprodukten.

Resultatet har blivit en fullt fungerande robot med stora möjligheter till vidare utveckling.

Innehållsförteckning

1. Inledning
2. Problemdefinition
3. Mål
4. Avgränsning
5. Lösningsmetod
    5.1. Faktainsamling
    5.2. Konstruktion
        5.2.1. Scanbot
        5.2.2. Användargränssnitt
        5.2.3. Servantbot
        5.2.4. Kommunikation
        5.2.5. Sammanfogande
6. Resultat
7. Referenser

Bilagor:
A. Kravspec
B. Lecture slide, University of San Paolo
C. Kodmall
D. Kommandolista
E. Programkod Scanbot
F. Programkod Servantbot

1. Inledning

Denna rapport är skriven som en del i det mekatronik projekt vi har genomför under första året vid Elektroteknik programmet, med inriktning Robotik och Mekatronik på KTH Campus Telge. Projektets innehåll har var inledningsvis väldigt öppet och stora delar lämnades till projektgrupperna att själva fastställa. Bland de få styrningar som från början fanns med var de tyngsta punkterna att roboten som skulle byggas skall vara baserad på Lego Mindstorms. Båda de två Mindstorms systemen RCX och NXT skulle utnyttjas och de två systemen skulle även på något sätt samverka. Vi valde att bygga en inläsningsenhet, Scanbot, och en verkställande enhet, Servantbot.

2. Problemdefinition

En inläsningsenhet läser in ett "hålkort", varje binärkod motsvarar ett kommando som sedan skickas till en verkställande robot som utför kommandona. Den verkställande roboten ska kunna fungera autonomt. Till inläsningsenheten används en RCX-modul och till den verkställande roboten används en NXT-modul.

3. Mål

Vår robot skall ge en bild av hur två olika system kan samverka för att gemensamt lösa en uppgift. Roboten skall bestå av två enheter, en inläsningsenhet och en verkställande robot. Inläsningsenheten, Scanbot, skall vara baserad på en RCX och den verkställande roboten, Servantbot, på en NXT. Kommunikationen mellan de två enheterna skall vara direkt, d.v.s. icke via en dator. För en mer detaljerad kravspecifikation se bilaga A.

4. Avgränsning

Vi har valt att inte inkludera avgränsningar i detta projekt. Detta då det ej finns en kunds beställning som grund för vårat arbete och därmed ej någon budget att hålla sig till. Vidare är projektet av sådan art att det finns en väldigt stor flexibilitet i det och avgränsningar skulle kunna vara i vägen för denna flexibilitet.

5. Lösningsmetod

5.1 Faktainsamling

I det inledande skedet av projektet genomfördes en faktainsamlig för att underlätta det fortsatta arbetet. Ett av de först ställningstaganden vi ansåg att vi måste göra avsåg vilket programspråk och vilken utvecklingsmiljö vi skulle använda. Vi sökte efter tänkbara alternativ på internet och provade två av dem. PbLua, som är ett skriptspråk för Mindstorms NXT som är baserat på Lua. Samt RobotC som är en utvecklingsmiljö som tillåter programmering för Mindstorms NXT och RCX i C. Vi valde att använda RobotC främst på grund av att det i tester visat sig exekvera program mycket snabbare än andra alternativ, samt att alla i projektgruppen redan från början kunde programmera i C.

Tidigt i projektet tittade vi på en konstruktion av Servantbot med synchrodrive, synkron styrning och drift av samtliga hjul. Denna konstruktionslösning har fördelen

med en enkel kinematik då roboten ändrar riktning utan att själva roboten ändrar sin vinkel. Detta gör att man enkelt kan mäta in roboten i ett koordinat system och kartlägga dess position. Nackdelarna är också många.
Det visade sig vara svårt att med de tillgängliga delarna konstruera en sådan robot. Då endast tre roterande plattformar fanns tillgängliga provades en trehjulig konstruktion, vilket innebär att dessa måste sitta i en liksidig triangel för att rörelsemönstret ska fungera. Att bygga en liksidig triangel i lego går utmärkt men det fanns inte tillgång till några drev som passade i denna triangel. Det finns säkert andra sätt att bygga en trehjulig synchrodrive robot med lego men dessa innebär en högre komplexitet och därmed en lägre tillförlitlighet. Av dessa anledningar lades denna konstruktion ned.

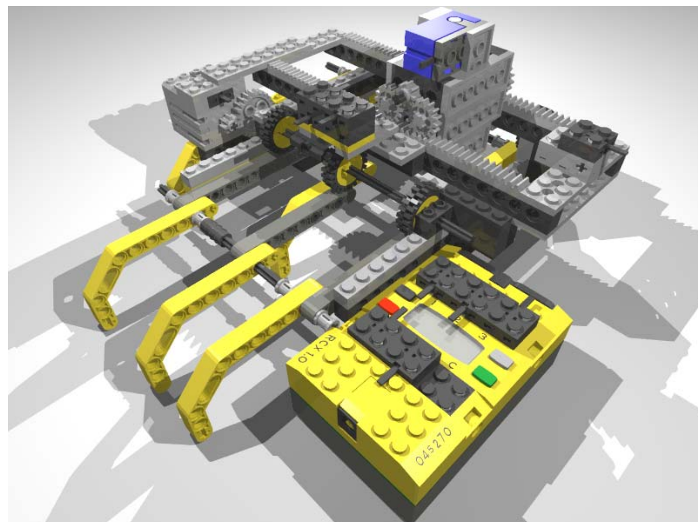
Att två robotar baserade på Lego Mindstorms RCX- respektive NXT-system skulle interagera på något sätt var ett av få, om inte det enda kravet på projektet innan vi började. Vi såg främst tre alternativ på interaktion – mekaniskt samspel, elektronisk kommunikation via PC samt direkt elektronisk kommunikation. Det senare föreföll vara det mest attraktiva och flexibla alternativet för oss. Att kommunicera direkt mellan RCX och NXT låter sig dock inte göras helt lätt, då NXT:n kommunicerar via sladd eller blue-tooth och RCX:n med IR. Problemet verkade i princip löst i och med ett inköp av IR-adapter från Mindsensors.com, som enkelt kopplas till NXT:n och vips kan NXT:n kommunicera medelst IR.

5.2 Konstruktion

5.2.1 Scanbot

Det har gjorts olika modeller, men till slut bestämde vi oss för en låg och smidig design. Hänsyn bör tas till att vi inte går en form- och designutbildning.

Scanbot är uppbyggd av en RCX robot, den har en ljussensor, en trycksensor, en varvräknare och två motorer. Den ena motorn driver inmatningen av papper och den andra driver ljussensorn som läser den binära koden.



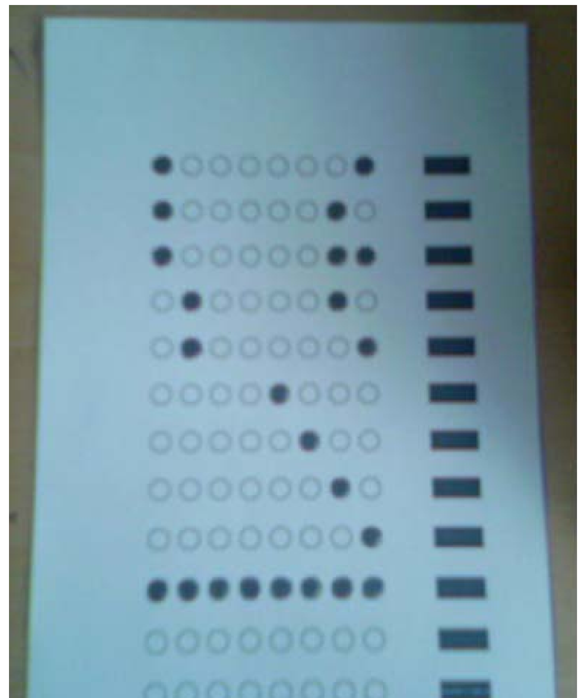
Pappret som matas in följer en bestämd mall (se bilaga 1), där binärkod representeras av punkter. Svart punkt är ett, vit punkt är noll. Avstånden mellan punkterna bestämdes efter mycket experimenterande, eftersom koden kan gå för snabbt och punkterna då inte läses. Ibland krånglar mekaniken, t.ex. kan en kugge missa sitt läge och detta får till följd att ljussensorn går olika långt

Att göra ett stabil och fin design tog sin tid. Inledningsvis byggdes en scanner där pappret inte matades in, utan låg på marken. Istället rörde sig ljussensorn både i x- och i y-led. Detta blev klumpigt och tog för mycket plats, dessutom var ett av målen

att pappret automatiskt matas in när man trycker på startknappen. Denna prototyp kunde således inte vara vår slutprodukt.

Under arbetet som ledde fram till Scanbot övervägdes en mängd olika designer som dock kunde avfärdas redan på skisstadiet. Slutligen så kom vi fram till slutdesignen och hänsyn har tagits för vidare utveckling av Scanbot. T.ex. kan man använda små modifikationer för bredare papper, eller lägga en extra motor för att göra Scanbot flyttbar etc.

Scanbot har en jämn yta och tre hjul som driver fram pappret och två andra som håller pappret nedtryckt. Där finns även raka Legobitar som håller pappret på plats före, under och efter processen och hindrar det från att glida åt sidorna.



När koden läses in från den fastställda mallen går scanner huvudet från höger till vänster. På mallen finns en fyrkantig svart rektangel på varje rad innan koden börjar, detta för att ljussensorn ska känna igen denna och därmed får programmet veta att där finns en rad av åtta punkter som representerar en byte. För att komma ut från den svarta rektangeln har vi en while där villkor är kör så länge det är svart. Kommer den till vit så kör den stora while loopen i Scan funktionen tills man kommer till svart punkt, alltså till första möjliga etta. Vart den stannade får vi fram genom att ha en räknarfunktion där det finns åtta if-satser med intervallvillkor som bygger på varvräknare.

Att den vänder beror på att vi har en slinga med villkor som bygger på varvräknare. Alltså har du kommit den sträcka vänd och gå tillbaka till trycksensorn.

Det är trycksensorn som nollställer varvräknaren och summan av varje byte. Programmet körs tills det läser 255 i sum variabel som är en variabel som plussar alla svarta punkter ”ettor”.

5.2.3 Servantbot

Servantbot är konstruerad för att vara enkel, tillförlitlig och svänga bra. Den är trehjulig med framhjulsdrift och har två direkt drivande motorer. Styrningen är differentiell, skillnaden i rotation mellan de drivande hjulen avgör hur mycket den svänger. Programmeringen av Servantbot möjliggör två typer av svängar, centrumsvängar där roboten svänger ett antal grader med- eller motsols runt centrum av den drivande axeln, samt svängar där en radie och ett antal grader anges. Vid den senare typen av sväng kör Servantbot således längs en tänkt cirkelbåge, med angiven radie, tills den svängt angivet antal grader. En stor del av konstruktionen är baserad på Castor Bot från nxtprogams.com.

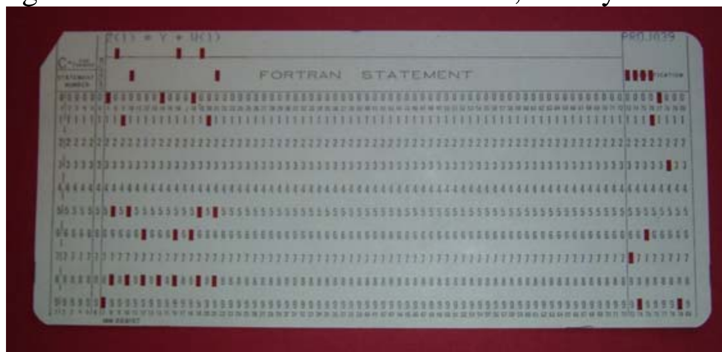
För att kunna påverka sin omgivning är den försedd med en gripklo. Konstruktionen med en gripklo ger roboten en något ogynnsam viktfördelning då lite för mycket vikt hamnar över de drivande axlarna. Detta gör att den inte klarar kraftiga inbromsningar bra, detta kompenseras bort med en lägre hastighet och låg markfrigång i fronten och är inte ett problem. Servantbot är programmerad så att klon kan öppnas eller stängas på kommando. När klon når ändlägena kommer programmeringen känna av att motorerna ej kan öppna/stänga klon mer och därvid lämna klon i det läge den då har. På detta sätt kommer klon även att gripa saker, kommando ges för att stänga klon och när klon gripit om föremålet så motorerna inte längre kan stänga klon mer, kommer klon att hållas i sitt läge.

Servantbots sensorbestyckning består av en magnetisk kompass, en ultraljudssensor, en IR-länk och rotationssensorer som är inbyggda i de tre motorerna. Kompassen används för att känna av vilken riktning Servantbot för tillfället står i. Programmeringsmässig uppdateras en global variabel med kompassens riktning från en egen task. Ultraljudssensorn används enbart av de rutiner som hanterar klon. Klon använder även rotationssensorn som finns inbyggd i dess motor. Rotationssensorerna i de två andra motorerna används för att styra robotens hastighet och även hur långt den åker vid rörelse rakt framåt eller bakåt. IR-länken används uteslutande för kommunikation med inläsningsenheten.

5.2.3 Användargränssnitt

Användargränssnitt

Användargränssnittet är konstruerat med de gamla hålkorten som förlaga. Hålkort var databärare som användes i datorns föregångare hålkortsmaskinen redan på 1800-talet. Hålkort användes för in- och utmatning av data i datorer ända fram till slutet av 1970-talet. Det vanligaste formatet var 80-kolumnerskortet, som rymde 80 tecken (bytes).



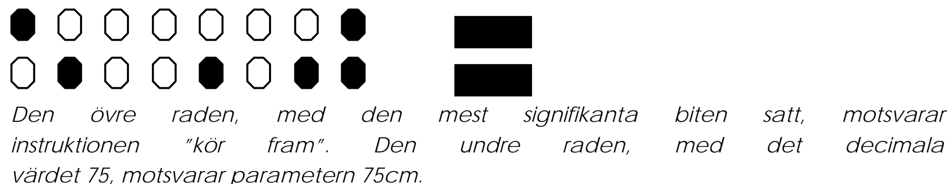
Typiskt hålkort som användes i datorer på 1960-talet och 1970-talet

Istället för att stansa ut hål i arken har vi bestämt oss för en mall där man fyller i ringar med en tuschpenna. Korten som används består av 32 rader med 8 ”bitar” i varje rad. En bit består av en tom ring som kan fyllas i med en svart tuschpenna. Ifylld ring står för en etta och en tom ring motsvarar en nolla.

Åtta vita ringar följt av en kantmarkering för styrning av scanner-huvudet. De åtta ringarna motsvarar en byte i binärkod.

På detta sätt har man 256 olika kombinationer av ettor och nollor, motsvarande en byte. På sidan av kortet finns en svart markering för varje rad som scanner känner av,

stannar frammatningen och läser in motsvarande rad. Kombinationerna är uppdelade i 128st instruktioner och parametrar i värdena 0-127. Instruktioner känns igen på att den mest signifikanta biten är satt, motsvarade alla decimala tal från 128-255.
En ifylld rad kan motsvara en instruktion eller en parameter till en instruktion. De flesta instruktioner kräver en parameter, exempelvis instruktionen ”kör fram” (1000 0001) som kräver en parameter motsvarande sträckan i cm. Instruktionen ”avsluta” (1111 1111), som alltid måste finnas med som sista instruktion, kräver ingen parameter.

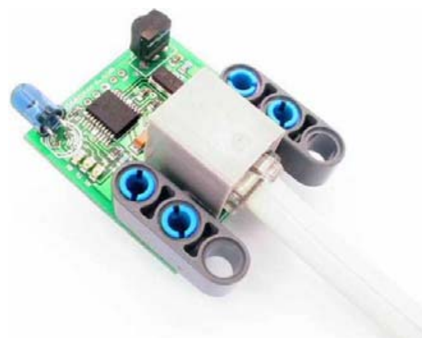


Den övre raden, med den mest signifikanta biten satt, motsvarar instruktionen ”kör fram”. Den undre raden, med det decimala värdet 75, motsvarar parametern 75cm.

Antalet rader kod som ryms på ett kort är litet (32st) men antalet möjliga instruktioner är stort (127st) vilket ger utrymme för ”färdiga” instruktioner med standardparametrar satta. T ex ”sväng 45 grader motsols”, ”sväng 90 grader medsols” osv. Om man använder sig av ”färdiga instruktioner” kommer antalet rörelser per kort öka markant.
Avståndet mellan punkterna är noga framtagen med hänsyn till scannerns upplösning. Vi fann att ett halvt A4 på längden skulle vara tillräckligt för att få plats med de 8 bitarna, kantmarkeringen och tillräckligt med avstånd mellan punkterna så att scannern skulle känna igen dem som enskilda punkter. Kortet tillåter även anteckningar i den vänstra marginalen som inte stör inläsningen. Efter justering med tanke på Lego’s fasta mått fastställdes kortets bred till 103 mm.
Scannern läser av rad för rad, bit för bit. Med hjälp av avståndet till kortets högerkant kan scannern avgöra vilket värde varje enskild markerad bit har. Varje rads binärtal omvandlas till ett decimalt tal. Alla instruktioner och parametrar sparas i minnet. När scannern har läst in instruktionen ”avsluta” (1111 1111) stannar den och skickar över den inscannade programkoden till den mobila roboten.
Kodbiblioteket utvecklas allt eftersom den mobila robotens rörelsemönster har programmerats. Eftersom enbart den mottagande roboten tolkar koden från korten kan man lätt lägga till nya instruktioner i mottagaren. Scannern kommer aldrig behöva modifieras. Man kan till och med tänka sig att låta scannern överföra binärkoden till en helt annan typ av robot för andra ändamål. Den enda reserverade instruktionen är ”avsluta”.

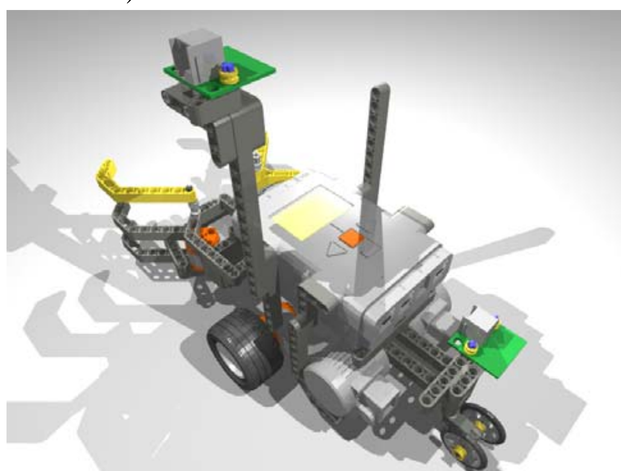
5.2.4 Kommunikation

För att lösa kommunikationen mellan Scanbot och Servantbot valde vi att använda en ”RCX to NXT Communication Adapter (NRLink-Nx)” från Mindsensors. NRLink kommer förprogrammerad med diverse makron som kan skickas till RCX:n för att utföra specifika kommandon såsom ”Run program 1”, ”Motor B on” eller ”Beep”. Några få av dessa makron ligger i ROM-area och kan ej ändras men det mesta ligger i EEPROM och kan skrivas över med egna makron. Det finns dock ytterligare EEPROM-area som är tomt där man kan lägga det man behöver utan att skriva över något som är fabrikskodat. Som framgår av denna rapport i övrigt låg det



dock ej i vårt intresse att styra RCX:n från NXT:n med direktkommandon, snarare tvärtom och dessutom bara skicka direkt-meddelanden till NXT:n, det vill säga, ett meddelande av storlek byte i stil med ”155”. Detta skulle plockas upp av NXT:n och sparas i en array.
På Mindsensors hemsida finns att ladda ned ett antal exempel på kod (skrivet med RobotC), eller snarare två exempel vari främst två grundläggande funktioner är författade för att som ovan beskrivet kunna styra RCX från NXT.

* void NRLinkCommand(byte NRLinkCommand)
Grundläggande funktion som används vid initiering av adaptern. Man kan med denna funktion exempelvis ändra från long range till short range (IR), rensa FIFO-buffer eller förbereda för körning av ett makro som ligger sparat i adapterns minne. Vid det sistnämda måste dock adressen för makrot läggas till och då körs allting hellre av nedan funktion. Funktionen ” NRLinkCommand” används av oss i RxTx_funcs-koden.



* void NRLinkRunMacro(byte NRLinkMacroAdd)
Som nämnt ovan, denna funktion körs för att till exempel starta motor A. Man kan om man känner till OP-koden för ”set-message” i RCX:n även använda denna funktion för att skicka direkt-meddelanden. Det gör inte vi. Om denna funktion används och samma kommando (OP-code) körs två gånger i rad så ignoreras det andra kommandot. Det har att göra med att adaptern på något sätt emulerar RCXn:s fjärrkontroll där åtgärder har vidtagits i händelse av att användaren trycker för länge eller med för kort intervall på samma knapp, den andra tryckningen ignoreras då.

Vi ville bara skicka ett ”rått” meddelande direkt till RCX:n, men insåg snart att man inte kan skicka ut en ”byte” i luften. Något måste läggas till för att klargöra vart byten är på väg.

Efter många timmar över google har vi bland annat funnit följande hemsida http://graphics.stanford.edu/~kekoa/rcx/#Protocol (RCX internals). Här finns ett avsnitt om det seriella protokollet med en länk till ett inlägg i ett forum av Dave Baum http://graphics.stanford.edu/~kekoa/rcx/protocol.html som beskriver hur varje paket till en RCX över IR ser ut. Detta tillsammans med några ”slides” från Universitet i Sao Paulo (bilaga B) har klargjort följande: Förutom ”formalian” med startbit, paritet och stopbit börjar varje paket med tre bytes som är någon slags rubrik, nämligen hex 55 FF 00. Detta följs av operationskoden på det man vill göra (i vårt fall ”set-message”) hex F7. Det som följer är 1-komplementet av OP-koden, meddelandet, 1-komplementet av meddelandet, checksumma (som är summan av OP-koden och meddelandet i en byte) samt 1-komplementet av checksumman. Resulterande i ett paket på 9 bytes. Ett exempel, vi skickar meddelandet hex 9B. Paketet att skicka blir då: 55 FF 00 F7 08 9B 64 92 6D
Eftersom vi har valt att skicka meddelanden ”råa” och utan hjälp av funktionen NRLinkRunMacro, måste vi skicka ett sådant paket varje gång med egna metoder. Vi

gör så med funktionen send_data i RxTx_funcs. Det är också med vetskapen om hur paketet ska se ut vi ”villkorar” bort eventuellt IR-brus som hamnar i Rx-buffern. Mer om detta nedan.

Vad vår kravspecifikation kräver är att RCX:n kan sända meddelanden till NXT:n, men vi skickar även ett ”kvitto” från NXT:n till RCX:n på att respektive byte har gått fram. Kvittot har samma innehåll som meddelandet. Så här ser vårt protokoll ut i princip: RCX skickar ett meddelande, väntar 750 ms på kvitto, om inget kvitto kommer – vänta 750 ms till och skicka igen. Om inget kvitto kommer efter ytterligare 750 ms upprepas sändningen varje 750 ms till dess att kvitto erhållits. NXT läser av statusflagga för Rx – buffer, om tom – vänta lite och läs sedan av igen. När något finns att hämta läses detta in, villkoras på att alla bytes fram till och med själva meddelandet ser ut som de ska, och om i så fall sparas meddelandet i global variabel för inkommande meddelande, ”Rx_data”. Buffer rensas och processen börjar om igen. I NXT:n sköts avläsningarna i en task och sysselsätter adaptern med kommunikation med NXT:n nästan hela tiden. Därför startar och stoppar vi denna task i funktionen build_commands som förutom att bygga en array med kommandon även skickar kvittot på att respektive meddelande har mottagits. Vårt protokoll kräver med andra ord ett samspel mellan dessa två funktioner.

5.2.5 Sammanfogande

När Scanbot och Servantbot var färdigbyggda med rudimentär programmering vidtog arbetet med att sammanfoga enheterna till ett system m.h.a det kommunikationsprotokoll som beskrivits ovan. Sammanfogandet gick väldigt smidigt tack vare de användarvänliga funktionerna som skrivits. På Scanbot var det bara att lägga till funktionen send_data(). I Servantbots program inkluderades en c-fil med samtliga för kommunikationen nödvändiga funktioner och ”ta emot”-funktionen tillkallas i början av main. Alla kommandon och parametrar som skickats från Scanbot lagras då i en array som vi sedan läser och tolkar i en enkel switch.

6. Resultat

Den färdiga produkten är en fullt fungerande robot med två separata enheter sammankopplade via en NRLink-Nx. De övergripande målen med projektet kan anses vara uppfyllda. Vad gäller Scanbot uppfyller den samtliga punkter i kravspecen. Kommunikationen mellan Scanbot och Servantbot har även det nått samtliga punkter i kravspecen. Servantbot har dock ej helt nått upp till de krav som ställdes i kravspecen. Den punkt i kravspecen som ej är uppfylld är ” 8. Felrutin om kommando ej kan utföras.”. Programmeringen i Servantbot är uppbyggd på ett sätt som gör adderingen av en felhanteringsrutin enkel, det har dock helt enkelt inte funnits tid att skriva denna och andra funktioner har prioriterats. Samtliga övriga punkter i kravspecen anses uppfyllda.
Om mer tid funnits skulle arbetet fortsatt med att förbättra precisionen i Servantbots rörelser samt, i stort sätt samtliga funktioner kan förbättras för att öka precisionen markant. Vidare kan nya funktioner lätta läggas till i Servantbots program och möjligheterna är därmed obegränsade.

7. Referenser

RobotC hemsida, http://www.robotc.net/, 080515

Mindsensors hemsida, http://www.mindsensors.com/, 080515

Castor Bot på nxtprograms hemsida, www.nxtprograms.com/castor_bot/, 080429

Kravspecifikation

Systemet ska bestå av en inläsningsenhet och en mobil robot.

Krav inläsningsenhet:

1. Uppbyggd av en RCX robot.
2. Läser pappersformat 105 X 297 mm.
3. Läser endast ett förutbestämt dokumentformat.
4. Autostart vid pappersinmatning.
5. Skickar data enligt förutbestämt protokoll.

Krav dataöverföring:

1. Skickar direkt RCX – NXT, ej via dator.
2. Handskaknings rutin vid överföring ska finnas.

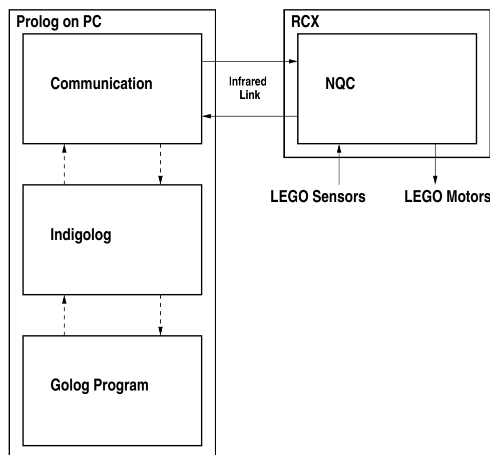
Krav Robot:

1. Uppbyggd av en NXT robot.
2. Hjul driven.
3. Skall ha ett gripverktyg
4. Ska kunna ta emot och lagra ett bestämt antal kommandon med parametrar
5. Starta när bestämt slutkommando skickas.
6. Återvända till startpunkt när kommandon är utförda.
7. Köra felrutin vid felaktiga kommandon.
8. Felrutin om kommando ej kan utföras.
9. Kommandon som skall kunna utföras:
   - 9.1 Kör ett avstånd rakt framåt/bakåt.
   - 9.2 Sväng ett antal grader medsols/motsols runt centrum på den drivande axeln.
   - 9.3 Sväng ett antal grader medsols/motsols längs en cirkelbåge med en viss radie.
   - 9.4 Kör till en punkt i ett definierat koordinatsystem
   - 9.5 Öppna/stäng gripverktyget
   - 9.6 Sök efter det närmaste objektet, när det är funnet kör till det och ta tag i det med gripverktyget
   - 9.7 Kör ett förutbestämt rörelse mönster baserat på ett tangogrundsteg (Salida)

Legolog: The Basic Idea

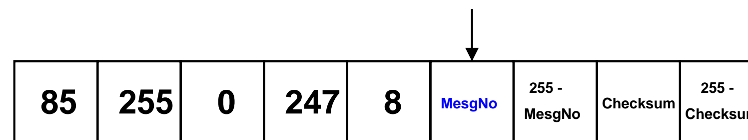
- Written in Prolog and NQC
- Communicates actions via infrared tower
- Prolog initiates all communication
  - Golog determines next action to execute and sends message to RCX; RCX must acknowledge within 3.5 seconds with sensing value
  - Golog can also "query" RCX to determine whether exogenous action has occurred (currently, only one exogenous action stored)
- Using Indigolog interpreter: concurrency, interrupts, exogenous actions, search operator

Legolog



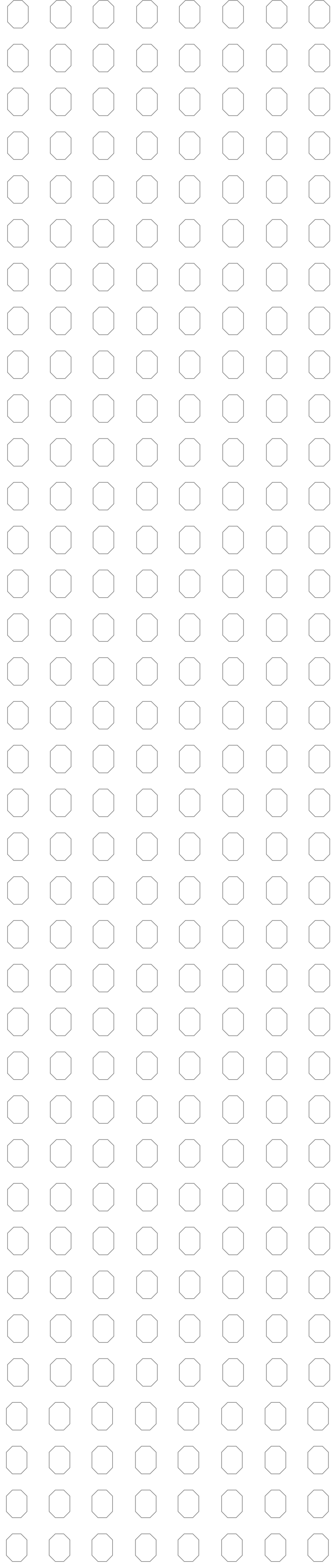
RCX User Messages

- RCX has simple error-checking protocol for communicating via infrared transmitter/receiver
- Messages are used to program RCX firmware, check battery level, etc.
- One particular message type—user message (our terminology)—allows numbers in the range 1 – 255 to be sent/received
- Legolog uses these for all communication
- User message packet format

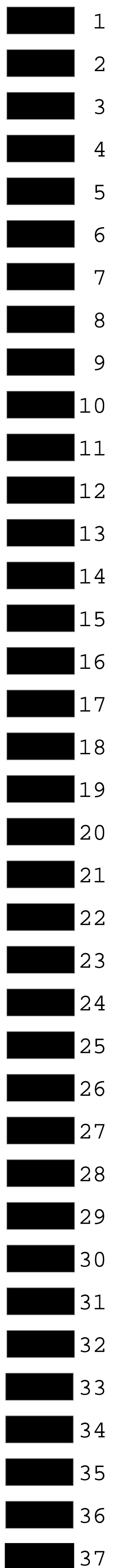


Legolog Protocol

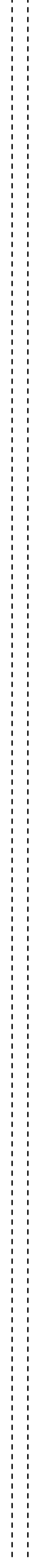
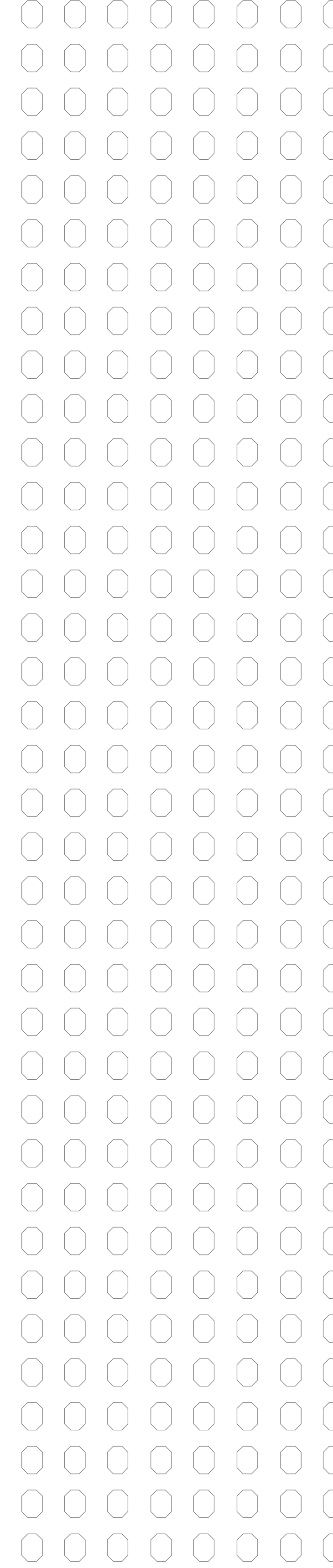
- **Desideratum:** send/receive arbitrarily large (positive) numbers
  - Allow multiple RCXs
  - Arbitrary sensing values
- How?
  - Send numbers $1 \leq n \leq 7$ bits at a time (least significant bits first)
  - Make use of a "continuation bit" to signal that more information is to follow
  - Also, a handful of special messages (exogenous request, continue, abort, request extra time, no exogenous action)
- Prolog initiates all communication (due to infrared tower "time-out")
  - Not a problem since RCX would need to wait for Golog anyway



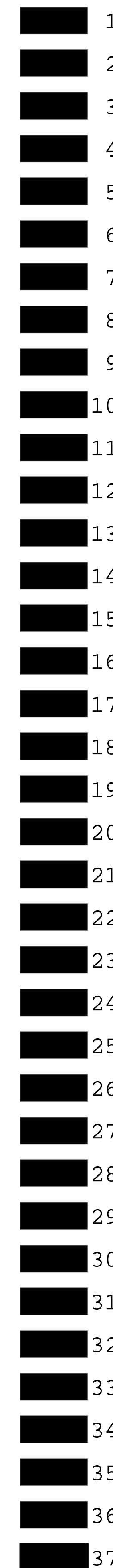
128 64 32 16 8 4 2 1
End Of File



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

128 64 32 16 8 4 2 1
End Of File



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

Scanbot & Servantbot

Kommandolista

Kod	Funktion	Parameter
129	Kör framåt	P1: Längd i cm
130	Kör bakåt	P1: Längd i cm
131	Centrumsväng medsols 0-120 grader	P1: Antal grader
132	Centrumsväng medsols 120-240 grader	P1: Antal grader överstigande 120
133	Centrumsväng medsols 240-360 grader	P1: Antal grader överstigande 120
134	Centrumsväng motsols 0-120 grader	P1: Antal grader
135	Centrumsväng motsols 120-240 grader	P1: Antal grader överstigande 120
136	Centrumsväng motsols 240-360 grader	P1: Antal grader överstigande 240
137	Lyxsväng medsols	P1: Antal grader (0-120) P2: Radie
138	Lyxsväng motsols	P1: Antal grader (0-120) P2: Radie
139	Tango!!	-
140	Kör till koordinater	P1: X koordinat P2: Y koordinat
141	Öppna klon	-
142	Stäng klon	-
143	Sök efter och tag närmsta objekt	-

File: G:\Slutgiltig verision\KillerBot.c

```
//*!!Sensor,    S2,             CmpsPort, sensorI2CCustomFast9V,      ,
//*!!Sensor,    S4,                sonar, sensorSONAR9V,      ,
//*!!Motor,  motorA,               motorA, tmotorNxtEncoderClosedLoop,
//*!!Motor,  motorB,               motorB, tmotorNxtEncoderClosedLoop,
//*!!
//*!!Start automatically generated configuration code.
const tSensors CmpsPort             = (tSensors) S2;   //sensorI2CCustomFast9
const tSensors sonar                = (tSensors) S4;   //sensorSONAR9V      /
const tMotor   motorA               = (tMotor) motorA; //tmotorNxtEncoderClos
const tMotor   motorB               = (tMotor) motorB; //tmotorNxtEncoderClos

#include "RxTx_funcs_01.c"                           // Include file contains
#include "driveFunk.c"                               // Include file contains
#include "KillerSwitch.c"                            // Include file contains

task main() {
  int i;

  nxtDisplayTextLine(1,"Fetching data");
  nxtDisplayTextLine(2,"from RCX");
  for(i = 0; i < LEN; i++)                             //Clears the array
    command[i] = 0;

  init_adapter();                                      //Must be in main for l
  i = build_commands();                                //Build command array..

  nxtDisplayTextLine(1,"Initieringar");
  nxtDisplayTextLine(2,"");
  nMotorEncoder[motorA]=0;
  nMotorEncoder[motorB]=0;
  nPidUpdateInterval=15;
  nI2CBytesReady[CmpsPort] = 0;

  cmpsCorr=cmpsHeading()*10;                             // This is the only t

  nxtDisplayTextLine(1,"Startar tasks");
  nxtDisplayTextLine(2,"                    ");

  StartTask(getheading);
  StartTask(position);

  nxtDisplayTextLine(1,"Startar");
  nxtDisplayTextLine(2,"KillerSwitch");

  KillerSwitch();

  nxtDisplayTextLine(1,"Program complete");
  nxtDisplayTextLine(2,"Klart, slut!");
  wait10Msec(400);

StopAllTasks();
}
/*****************************************************************************
********** Task that continously updates the globla variable currheading ****
*****************************************************************************

task getheading(){
  while(1){
    currheading = cmpsHeading();
    if ( currheading == -1 ) {
        nxtDisplayTextLine(6,"Error: Check connections");
    } else {
```

```
        nxtDisplayTextLine(6,"Read OK");
    }
    nxtDisplayTextLine(7,"Riktning = %d", currheading);


   }
}

/*****************************************************************************
******** Task that continously updates the globla variabls XposC and YposC **
*****************************************************************************

task position() {
  int EncMotorA, EncMotorB, EncAold=0, EncBold=0;
  int PosHeading;
  float distans, DeltaX, DeltaY, dEncA, dEncB;

  while(1){
    PosHeading=currheading;
    EncMotorA=nMotorEncoder[motorA];                          //Laser in vart
    EncMotorB=nMotorEncoder[motorB];                          //Laser in vart
    dEncA=EncMotorA-EncAold;
    dEncB=EncMotorB-EncBold;
    distans=abs(dEncA/(DistCm/10));

    if (nMotorRunState[motorA] != runStateIdle) {
        if (dEncA>0 && dEncB<0){                              // om Motorerna
            DeltaX=0;
            DeltaY=0;
        }
        else if (dEncA<0 && dEncB>0){                         // om Motorerna
            DeltaX=0;
            DeltaY=0;
        }
        else {
            if (PosHeading==-1){
            nxtDisplayTextLine(3,"Error: Task pos -1");
            DeltaX=0;
            DeltaY=0;
            }
            else {
                if (dEncA<0 && dEncB<0){
                      if (PosHeading>=180) PosHeading=PosHeading-180;
                      else PosHeading=PosHeading+180;
                }
                if (PosHeading==00){
                      DeltaX=0;
                      DeltaY=distans;
                }
                else if (PosHeading<90){
                      DeltaX=distans*sinDegrees(PosHeading);
                      DeltaY=distans*cosDegrees(PosHeading);
                }
                else if (PosHeading==90){
                      DeltaX=distans;
                      DeltaY=0;
                }
                else if (PosHeading<180){
                      DeltaX=distans*cosDegrees(PosHeading-90);
                      DeltaY=-distans*sinDegrees(PosHeading-90);
                }
                else if (PosHeading==180){
```

```
                        DeltaX=0;
                        DeltaY=-distans;
                  }
                  else if (PosHeading<270){
                        DeltaX=-distans*sinDegrees(PosHeading-180);
                        DeltaY=-distans*cosDegrees(PosHeading-180);
                  }
                  else if (PosHeading==270){
                        DeltaX=-distans;
                        DeltaY=0;
                  }
                  else {
                        DeltaX=-distans*cosDegrees(PosHeading-270);
                        DeltaY=distans*sinDegrees(PosHeading-270);
                }
              }
          }
      }
      else {
        DeltaX=0;
        DeltaY=0;
      }

      if ( PosHeading == -1 ) {
          nxtDisplayTextLine(6,"Error: Check connections");
      }
      else {
         nxtDisplayTextLine(6,"Read OK");
      }

      XposC=XposC+DeltaX;
      YposC=YposC+DeltaY;

      EncAold=EncMotorA;
      EncBold=EncMotorB;

      nxtDisplayTextLine(4,"X: %d", XposC);
      nxtDisplayTextLine(5,"Y: %d", YposC);
      wait10Msec(20);
    }
}

/*****************************************************************************
********************* Task that checks if motorC has stalled ****************
*****************************************************************************

task stallMonitoringTask()
{
  int nLastEncoder;


  nLastEncoder = nMotorEncoder[motorC];      // Check every 20 msec to see if
  while (true)                               // If not, increment a counter
  {
    int nEncoderDelta;

    wait1Msec(20);
    nEncoderDelta = (nMotorEncoder[motorC] - nLastEncoder);
    if ((nEncoderDelta > 2) || (nEncoderDelta < -2))
    {
      nStallCount = 0;
      nLastEncoder = nMotorEncoder[motorC];
```

```
    }
    else
    {
      if (nStallCount < 1000)
        ++nStallCount;
    }
  }
}
```

File: G:\Slutgiltig verision\KillerSwitch.c

```
// Function that reads the global array "command" and executes functions depe
// values read from the array

void KillerSwitch() {
    int i,j=0,sctl,p1,p2;
    for(i=0;i<LEN;i++){

      sctl=command[i];
      if (sctl<0) sctl=(sctl&0xFF);
      nxtDisplayTextLine(2,"Case: %d",sctl);
      wait10Msec(200);

      switch(sctl){
        case 129:                                          // Drives forward
          i++;
          p1= command[i];                                  // Parameter that
          Drive(F,p1);
          break;

        case 130:                                          // Drives backwar
          i++;
          p1=command[i];                                   // Parameter that
          Drive(B,p1);
          break;

        case 131:                                          // Turns around t
          i++;
          p1= command[i];                                  // Number of degr
          cTurn(R,p1);
          break;

        case 132:                                          // Turns around t
          i++;
          p1= command[i];                                  // Number of degr
          p1=p1+120;
          cTurn(R,p1);
          break;

        case 133:                                          // Turns around t
          i++;
          p1= command[i];                                  // Number of degr
          p1=p1+240;
          cTurn(R,p1);
          break;

       case 134:                                           // Turns around t
          i++;
          p1= command[i];                                  // Number of degr
          cTurn(L,p1);
          break;

      case 135:                                            // Turns around t
          i++;
          p1= command[i];                                  // Number of degr
          p1=p1+120;
          cTurn(L,p1);
          break;

      case 136:                                            // Turns around t
          i++;
          p1= command[i];                                  // Number of degr
          p1=p1+240;
```

```
          cTurn(L,p1);
          break;

      case 137:                                              // Turns along a
          i++;
          p1=command[i];                                     // Number of degr
          i++;
          p2= command[i];                                    // Radius
          Turn(R,p1, p2);
          break;

      case 138:                                              // Turns along a
          i++;
          p1=command[i];                                     // Number of degr
          i++;
          p2= command[i];                                    // Radius
          Turn(L,p1, p2);
          break;

      case 139:
          Tango();
          break;

      case 140:                                              // Goes to specef
          i++;
          p1=command[i];                                     // Target X coord
          i++;
          p2= command[i];                                    // Target Y coord
          Gogo(p1, p2);
          break;

      case 141:                                              // Opens the grab
          do_grabber(1);
          break;

      case 142:                                              // Closes the gra
          do_grabber(2);
          break;

      case 143:                                              // Finds the clos
          FindNGrab();
          break;

      case 255:
          i=LEN;                                             //Array max fr at
          Gogo(0,0);                                         // Returns Servan
          break;

        default:
          j++;
          nxtDisplayTextLine(1,"Meh, Default %d",j);         // Displays on th
          nxtDisplayTextLine(2,"sctl %d",sctl);              // reached the de
          wait10Msec(100);                                   // it has reached

        break;
        }
      }
}
```

File: G:\Slutgiltig verision\driveFunk.c

```
  // Definitions
#define L 0
#define R 1
#define F 0
#define B 1
#define DistCm 22.9                                    // Number of encoder tick
#define bMotorCStalled()   (nStallCount > 3)
#define close 1
#define open 2

  // Compass definitions
#define    CmpsID               0x02
#define    CmpsCommandReg       0x41
#define    CmpsReadResult       0x42
#define    CmpsReadHeading      0x49
#define    CmpsCAL              0x43
#define    CmpsEndCAL           0x44
#define    CmpsSampleUS         0x55
#define    CmpsSampleEU         0x45
#define    CmpsAuto             0x41
#define    CmpsPort             S2                     // Connect CMPS sensor tc

    // Global variables
int currheading;                                       // Current heading, this
int cmpsCorr=0;                                        // Compass correction val
int XposC=0, YposC=0;                                  // X and Y coordinates, c
int nStallCount = 0;

//*******************************************************************
//*******************   Function prototypes   ***********************
//*******************************************************************

void cTurn (int dir, int deg);                         // Turns around the robot
void Turn (int dir, int deg, int rad);                 // Turns along a circle
void Drive (int dir, int dist);                        // Drives a distance "dis
void Gogo (int NewX, int NewY);                        // Goes from current posi
//void GoHome (void);                                  // Returns to starting po
void Tango ();                                         // Makes the robot dance
int  cmpsHeading();                                    // Returns the current he
void FindNGrab();                                      // Searches for the neare
void do_grabber(byte action);                          //Opens or closes grabber
task position();                                       // Continusly updates Xpc
task getheading();                                     // Contiunsly updates cur
task stallMonitoringTask();



//*******************************************************************
//*******************   Function definitions   **********************
//*******************************************************************

  // Turns the robot around its axis
void cTurn (int dir, int deg) {
  int currHead, newHead, newHead2;
  currHead=currheading;

  if (dir==R) {                                        // If the function recive
    newHead=currHead+deg;                              // turns the robot clockw
    if ((currHead+deg)<=360) {                         // is larger than the new
          nSyncedMotors=synchAB;
          nSyncedTurnRatio=-100;
          motor[motorA]=20;
```

```
              while (currheading<newHead) {
              }
            motor[motorA]=0;
    }
    else {
            newHead=newHead-360;
              nSyncedMotors=synchAB;
              nSyncedTurnRatio=-100;
              motor[motorA]=20;
              while (currheading>newHead){
            }
              nSyncedMotors=synchAB;
              nSyncedTurnRatio=-100;
              motor[motorA]=20;
              while (currheading<newHead){
            }
          motor[motorA]=0;
      }
  }
  if (dir==L) {                                         // If the function recive
    newHead=currHead-deg;                               // turns the robot anti-c
    if (newHead<0) newHead=newHead+360;                 // is smaller than the ne

    if ((currHead-deg)>0) {
              nSyncedMotors=synchBA;
              nSyncedTurnRatio=-100;
              motor[motorB]=20;
              while (currheading>newHead) {
            }
            motor[motorB]=0;
    }
    else {
              nSyncedMotors=synchBA;
              nSyncedTurnRatio=-100;
              motor[motorB]=20;
              while (currheading<newHead){
            }

              nSyncedMotors=synchBA;
              nSyncedTurnRatio=-100;
              motor[motorB]=20;
              while (currheading>newHead){
            }
          motor[motorB]=0;
      }
  }
  wait10Msec(10);

  while (currheading>newHead+1 || currheading<newHead-1){    // When the robot
    if (newHead<10) {                                        // this part of t
      newHead2=newHead+50;
      if ((currheading+50)<newHead2) {
              nSyncedMotors=synchBA;
              nSyncedTurnRatio=-100;
              motor[motorB]=20;
            //  wait10Msec(1);
              motor[motorB]=0;
            }
      else if ((currheading+50)>newHead2){
              nSyncedMotors=synchAB;
              nSyncedTurnRatio=-100;
              motor[motorA]=20;
```

```
            //  wait10Msec(1);
              motor[motorA]=0;
            }
    }
    else if (newHead>350) {
      newHead2=newHead-50;
      if ((currheading-50)<newHead2) {
              nSyncedMotors=synchBA;
              nSyncedTurnRatio=-100;
              motor[motorB]=20;
            //  wait10Msec(1);
              motor[motorB]=0;
            }
      else if ((currheading-50)>newHead2){
              nSyncedMotors=synchAB;
              nSyncedTurnRatio=-100;
              motor[motorA]=20;
            //  wait10Msec(1);
              motor[motorA]=0;
            }
    }
    else {
      if (currheading>newHead) {
              nSyncedMotors=synchBA;
              nSyncedTurnRatio=-100;
              motor[motorB]=20;
            //  wait10Msec(1);
              motor[motorB]=0;
            }
      else if (currheading<newHead){
              nSyncedMotors=synchAB;
              nSyncedTurnRatio=-100;
              motor[motorA]=20;
            //  wait10Msec(1);
              motor[motorA]=0;
            }
      }
    }

}

  // Turn along a circle
void Turn (int dir, int deg, int rad) {
  float langd, TurnRatio;

  langd=((6.28*(rad+6)*deg)*DistCm)/360;
  TurnRatio=(100*(rad-6))/(rad+6);

  if (dir==R) {
    nSyncedMotors=synchAB;
    nSyncedTurnRatio=TurnRatio;
    nMotorEncoderTarget[motorA]=langd;
    motor[motorA]=50;
    while (nMotorRunState[motorA] != runStateIdle) {
    }
  }
  if (dir==L) {
    nSyncedMotors=synchBA;
    nSyncedTurnRatio=TurnRatio;
    nMotorEncoderTarget[motorB]=langd;
    motor[motorB]=50;
    while (nMotorRunState[motorB] != runStateIdle) {
```

```
      }
    }
  }

    // Drives "dist" cm in "dir" direction
  void Drive (int dir, int dist) {

    nSyncedMotors=synchAB;
    nSyncedTurnRatio=+100;
    nMotorEncoderTarget[motorA]=DistCm*dist;
    if (dir==B){
      motor[motorA]=-50;
    }
    else motor[motorA]=50;
    while (nMotorRunState[motorA] != runStateIdle) {
      }

  }

    // Simulates a short tango rutine >.<
  void Tango(){
       Turn(R,45,30);
       cTurn(L,45);
       Drive(F,30);
       cTurn(R,90);
       wait10Msec(200);
  }

    // Returns the current heading from the compass
    // The value is adjusted by the global variable "cmpsCorr"
  int  cmpsHeading()
  {
    byte replyMsg[2];
    int Heading, HeadingCorr;
    byte cmpsMsg[5];
    const byte MsgSize        = 0;
    const byte CmpsAddress    = 1;
    const byte ReadAddress    = 2;
    const byte CommandAddress = 2;
    const byte Command        = 3;

    cmpsMsg[MsgSize]          = 3;                        // Build the I2C me
    cmpsMsg[CmpsAddress]      = CmpsID ;
    cmpsMsg[CommandAddress]   = CmpsCommandReg ;
    cmpsMsg[Command]          = CmpsReadHeading;

    while (nI2CStatus[CmpsPort] == STAT_COMM_PENDING);      // Wait for I2C bus
    sendI2CMsg(CmpsPort, cmpsMsg[0], 0);                    // Send the message

    cmpsMsg[MsgSize]        = 2;
    cmpsMsg[CmpsAddress]    = CmpsID ;
    cmpsMsg[ReadAddress]    = CmpsReadResult ;

    while (nI2CStatus[CmpsPort] == STAT_COMM_PENDING);      // Wait for I2C bus
    sendI2CMsg(CmpsPort, cmpsMsg[0], 2);                    // Send the message

    while (nI2CStatus[CmpsPort] == STAT_COMM_PENDING);      // Wait for I2C bus
    if ( nI2CStatus[CmpsPort] != NO_ERR ) {                 // probably sensor
      return (-1);
    }
    readI2CReply(CmpsPort, replyMsg[0], 2);
```

```
    if ( replyMsg[0] == -1 ) {                              // -1 is reserved t
   Heading = 0;                                             // change it to zer
   }
   else {
   Heading = ( 0x00FF & replyMsg[0] );
   }
   Heading += ( (0x00FF & replyMsg[1]) <<8 );


   if (Heading >= cmpsCorr) {                               // Add the compass
     HeadingCorr=(Heading-cmpsCorr)/10;                     // This sets the 0/
   }                                                        // the robot is fac
   else HeadingCorr=(Heading+3600-cmpsCorr)/10;

   return (HeadingCorr);
 }

     // Drives the robot to the coordinates recived
 void Gogo(int NewX, int NewY) {
   int gDeltaX, gDeltaY, gDist, gAngle, gDir;
   long bapelsin;
   float gHeading;

   gDeltaX=NewX-XposC/10;                                   // Calculates the d
   gDeltaY=NewY-YposC/10;                                   // to the new coord

   bapelsin=gDeltaX*gDeltaX+gDeltaY*gDeltaY;
   gDist=sqrt(bapelsin);

   if (gDeltaX==0 && gDeltaY>0) {                           // Calculates the d
     gHeading=0;                                            // to face the new
     }
   else if (gDeltaX>0 && gDeltaY>0) {
     gHeading=atan(gDeltaX/gDeltaY);
     gHeading=radiansToDegrees(gHeading);
     }
   else if (gDeltaX>0 && gDeltaY==0) {
     gHeading=90;
     }
   else if (gDeltaX>0 && gDeltaY<0) {
     gHeading=atan(abs(gDeltaY/gDeltaX));
     gHeading=radiansToDegrees(gHeading)+90;
     }
   else if (gDeltaX==0 && gDeltaY<0) {
     gHeading=180;
     }
   else if (gDeltaX<0 && gDeltaY<0) {
     gHeading=atan(gDeltaX/gDeltaY);
     gHeading=radiansToDegrees(gHeading)+180;
     }
   else if (gDeltaX<0 && gDeltaY==0) {
     gHeading=270;
     }
   else if (gDeltaX<0 && gDeltaY>0) {
     gHeading=atan(abs(gDeltaY/gDeltaX));
     gHeading=radiansToDegrees(gHeading)+270;
     }
   else gHeading=-1;


   if (gHeading>currheading){
     gAngle=gHeading-currheading;
```

```
      if (gAngle>180) {
        gAngle=360-gAngle;
        gDir=L;
      }
      else gDir=R;
    }
  else if (gHeading<currheading){
    gAngle=currheading-gHeading;
      if (gAngle>180) {
        gAngle=360-gAngle;
        gDir=R;
      }
      else gDir=L;
    }

    cTurn(gDir,gAngle);                                    // Turns towards the
    Drive(F,gDist);                                        // Drices to the new
}


    //Searches for the closest object, drives to it and grabs it.
void FindNGrab()
{                                                          //Searches for the cl
  nSyncedMotors=synchAB;
  nSyncedTurnRatio=-100;

  int iCnt, ObjDist=250,ObjAng, tmpObjAng, tmpObjDist, l, r;

  do_grabber(1);
  for(iCnt=0;iCnt<502;iCnt++)
  {                                                        // Finds  and grab cl
    nMotorEncoder[motorA]=0;
    nMotorEncoderTarget[motorA]=1;
    motor[motorA]=10;
    while(nMotorRunState[motorA]!=runStateIdle)
    {}
    wait1Msec(20);
    if(SensorValue[sonar]<250)
    {
      tmpObjDist=SensorValue[sonar];
      wait10Msec(2);
      tmpObjAng=currheading;
      if(tmpObjDist<ObjDist)
      {
        ObjDist=tmpObjDist;
        ObjAng=tmpObjAng;
      }
    }
  }

  if(ObjAng-currheading<0)
  {
    cTurn(0,abs(ObjAng-currheading));
  }
  else
  {
    cTurn(1,abs(ObjAng-currheading));
  }


    Drive(0,ObjDist+1);
    do_grabber(2);
```

```
}


void do_grabber(byte action)
{
  static byte closed = 0;                            //Set to 1 if grabber is tc

  StartTask(stallMonitoringTask);

  switch (action) {
    case close:
      if(!closed) {
        motor[motorC] = -30;
        wait1Msec(200);                              //Give time to stallMonitor
        while(!bMotorCStalled())
          wait1Msec(10);
        motor[motorC] = 0;
        closed = 1;
      }
      break;
    case open:
      if(closed) {
        motor[motorC] = 30;
        wait1Msec(200);
        while(!bMotorCStalled())
          wait1Msec(10);
        motor[motorC] = 0;
        closed = 0;
      }
      break;
    default:
      break;
  }
  StopTask(stallMonitoringTask);
}
```

File: G:\Slutgiltig verision\RxTx_funcs_01.c

```
#define LEN 64

        //definitions for NRLink

const ubyte NRLinkID              = 0x02;
const ubyte NRLinkCommandReg      = 0x41;
const tSensors NRLinkPort         = S1;        // Connect NRLink sensor to th

const ubyte NRLinkDefault        = 0x44;
const ubyte NRLinkFlush          = 0x46;
const ubyte NRLinkMacro          = 0x52;

      //Function prototypes for communication

//Send commands to the adapter command register.
void NRLinkCommand(byte  NRLinkCommand);

//Write a macro on desired memory location, (store bytes).
void NRLinkWriteMacro(byte  Location, byte data);

//Send a byte of data to RCX.
void send_data(byte data);

//Initiate a read at specified adress. To read from Rx buffer.
//Use 0x42 and expect 7 bytes back if a read of RCX message is desired.
void init_read(byte adress, byte nr_bytes);

//Initiate adapter for use.
void init_adapter();

//Stores commands sent by RCX until end-of-commands is received.
//Using global array "command[]". Returns number of elements stored.
int build_commands();

//Continiously checking Rx buffer for data. Stores last received data in vari
//Stop this task when data is received or not required.
task read_rx();

          //Global variables
byte replyMsg[7];   //Used in communicate task
byte rx_data = 0;       //Used as message variable, read and clear.
byte command[LEN];    //Command array. (To be built by communication.)



/////////////////////////////////////////////////////////////////////////////
/////////Function and task definitions//////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////

void init_adapter()
{
  nI2CBytesReady[NRLinkPort] = 0;
  SensorType[NRLinkPort] = sensorI2CCustom9V;
  NRLinkCommand(NRLinkFlush);
  NRLinkCommand(NRLinkDefault);
}

//55 ff 00 f7 08 12 ed 09 f6  sending 0x12 as a message  http://www.crynwr.co
void send_data(byte data)
{
  NRLinkCommand(NRLinkFlush);   //Flush buffer. (New).
```

```
  NRLinkWriteMacro(0x42, 0x55); // write to Tx buffer (0x42) (pseudo)
  NRLinkWriteMacro(0x42, 0xff);
  NRLinkWriteMacro(0x42, 0x00); //Header ends here
  NRLinkWriteMacro(0x42, 0xf7);   // OP-code
  NRLinkWriteMacro(0x42, 0x08);   // not OP-code
  NRLinkWriteMacro(0x42, data);   //Message
  NRLinkWriteMacro(0x42, ~data);    //Not message
  NRLinkWriteMacro(0x42, 0xf7 + data);    //  Check-sum
  NRLinkWriteMacro(0x42, ~(0xf7 + data)); // Not check-sum
  wait1Msec(20);
  NRLinkWriteMacro(0x40, 0x09); // send the buffer , 9 bytes.
}

void init_read(byte adress, byte nr_bytes)
{
  byte NRLinkMsg[5];
  const byte MsgSize         = 0;
  const byte Address         = 1;
  const byte Register        = 2;

  // Build the I2C message
  NRLinkMsg[MsgSize]        = 2;
  NRLinkMsg[Address]        = NRLinkID;
  NRLinkMsg[Register]       = adress;     //Pointer to Rx FIFO or other adres

  while (nI2CStatus[NRLinkPort] == STAT_COMM_PENDING);
  {
    // Wait for I2C bus to be ready
  }
  // when the I2C bus is ready, send the message you built
  sendI2CMsg(NRLinkPort, NRLinkMsg[0], nr_bytes);  //Expecting 7 bytes back a
}

void NRLinkCommand(byte  NRLinkCommand)
{

  byte NRLinkMsg[5];
  const byte MsgSize         = 0;
  const byte Address         = 1;
  const byte CommandAddress  = 2;
  const byte Command         = 3;


  // Build the I2C message
  NRLinkMsg[MsgSize]        = 3;
  NRLinkMsg[Address]        = NRLinkID;
  NRLinkMsg[CommandAddress] = NRLinkCommandReg  ;
  NRLinkMsg[Command]        = NRLinkCommand;

  while (nI2CStatus[NRLinkPort] == STAT_COMM_PENDING);
  {
    // Wait for I2C bus to be ready
  }
  // when the I2C bus is ready, send the message you built
  sendI2CMsg(NRLinkPort, NRLinkMsg[0], 0);
}

void NRLinkWriteMacro(byte  Location, byte data)
{

  byte NRLinkMsg[5];
  const byte MsgSize         = 0;
```

```
  const byte Address          = 1;
  const byte EEPROMLocation   = 2;
  const byte EEPROMData       = 3;


  // Build the I2C message
  NRLinkMsg[MsgSize]         = 3;
  NRLinkMsg[Address]         = NRLinkID;
  NRLinkMsg[EEPROMLocation] = Location  ;
  NRLinkMsg[EEPROMData]      = data;

  while (nI2CStatus[NRLinkPort] == STAT_COMM_PENDING);
  {
    // Wait for I2C bus to be ready
  }
  // when the I2C bus is ready, send the message you built
  sendI2CMsg(NRLinkPort, NRLinkMsg[0], 0);
}

task read_rx()
{
  int b_time = 200;
  static int cnt = 0;

  NRLinkCommand(NRLinkFlush);       //Flush FIFO buffer.
  while(1)
  {
    do
    {
      init_read(0x41, 1);   //Point to status reg.
      while (nI2CStatus[NRLinkPort] == STAT_COMM_PENDING)
      {
        // Wait for I2C bus to be ready
      }
      readI2CReply(NRLinkPort, replyMsg[0], 1);
      wait1Msec(b_time);
    }while(replyMsg[0] & 0x01 == 0);        // While buffer empty.

    init_read(0x42, 7);
    while (nI2CStatus[NRLinkPort] == STAT_COMM_PENDING)
    {
      // Wait for I2C bus to be ready
    }

    readI2CReply(NRLinkPort, replyMsg[0], 7);         //Expected reply bytes

    if (replyMsg[0] == 0x55 && replyMsg[1] == 0xff && replyMsg[2] == 0 && rep
      rx_data = replyMsg[5];
      nxtDisplayTextLine(3,"Rxvalue %d: %d", cnt++, (int)rx_data);      //Rem
    }

    NRLinkCommand(NRLinkFlush);       //Flush FIFO buffer.
  }
}

int build_commands()
{
  int i = 0;    //Was not set to zero....
  do
  {
    rx_data = 0;
    StartTask(read_rx);
```

```
    while(rx_data == 0)
      wait1Msec(2);         //Wait for rcx message.
    StopTask(read_rx);
    command[i++] = rx_data;
    wait1Msec(40);    //Wait for RCX to "recover" from sending, before sendin
    send_data(rx_data); //Send acknowledge.
  }while(rx_data != 255);   //Where 255 is an end-of-commands indication.

  rx_data = 0;
  return i;
}
```

File: G:\Slutgiltig verision\KompassKalibrering.c

```
/***************************************************************************
****************************** Compass calibration *************************
****************************************************************************

#define   CmpsID              0x02
#define   CmpsCommandReg      0x41
#define   CmpsReadResult      0x42
#define   CmpsReadHeading     0x49
#define   CmpsCAL             0x43
#define   CmpsEndCAL          0x44
#define   CmpsSampleUS        0x55
#define   CmpsSampleEU        0x45
#define   CmpsAuto            0x41

#define   CmpsPort         S2                          // Connect CMPS ser

void cmpsCommand(byte cmpsCommand)
{
  byte cmpsMsg[5];
  const byte MsgSize         = 0;
  const byte CmpsAddress     = 1;
  const byte ReadAddress     = 2;
  const byte CommandAddress  = 2;
  const byte Command         = 3;

  // Build the I2C message
  cmpsMsg[MsgSize]           = 3;
  cmpsMsg[CmpsAddress]       = CmpsID;
  cmpsMsg[CommandAddress]    = CmpsCommandReg ;
  cmpsMsg[Command]           = cmpsCommand;

  while (nI2CStatus[CmpsPort] == STAT_COMM_PENDING);      // Wait for I2C bus
  sendI2CMsg(CmpsPort, cmpsMsg[0], 0);                     // Send the message
}


task main()
{

  int count, i;
  nI2CBytesReady[CmpsPort] = 0;

  nxtDisplayTextLine(6, "Kalibrerar");
  nxtDisplayTextLine(7, "kompassen");

  SensorType[CmpsPort] = sensorI2CCustomFast9V;
  cmpsCommand(CmpsCAL);

  for (i=0;i<20;i++) {
    nSyncedMotors=synchAB;                               // Motor A is set to
    nSyncedTurnRatio=-100;                               // Slave motor will d
    nMotorEncoderTarget[motorA]=120;                     // Sets motor encoder
    motor[motorA]=50;                                    // Runs motor A forwa
    wait10Msec(200);
  }

  cmpsCommand(CmpsEndCAL);

  nxtDisplayTextLine(6,"Kalibrering");
  nxtDisplayTextLine(7,"klar");
  wait10Msec(400);
  StopAllTasks();
```

```
}
```

File: G:\Slutgiltig verision\grupp3.1_mod.c

```
//*!!Sensor,    S1,                 Rot1, sensorRotation,      ,
//*!!Sensor,    S2,               Light2, sensorReflection,      ,
//*!!Sensor,    S3,               Calib3, sensorTouch,      ,
//*!!Motor,  motorA,                 MotA, tmotorNormal,
//*!!Motor,  motorB,                 MotB, tmotorNormal,
//*!!
//*!!Start automatically generated configuration code.
const tSensors Rot1                 = (tSensors) S1;   //sensorRotation     /
const tSensors Light2               = (tSensors) S2;   //sensorReflection   /
const tSensors Calib3               = (tSensors) S3;   //sensorTouch        /
const tMotor   MotA                 = (tMotor) motorA; //tmotorNormal       /
const tMotor   MotB                 = (tMotor) motorB; //tmotorNormal       /
//*!!CLICK to edit 'wizard' created sensor & motor configuration.

//*!!CLICK to edit 'wizard' created sensor & motor configuration.

//*!!Sensor,    S1,                 Rot1, sensorRotation,      ,
//*!!Sensor,    S2,               Light2, sensorReflection,      ,
//*!!Sensor,    S3,               Calib3, sensorTouch,      ,
//*!!Motor,  motorA,                 MotA, tmotorNormal,
//Send a byte of data to NXT. Playes uptone if ok, else downtone.
//Hangs forever in function if no reply is received from NXT.



void send_data(byte data);
void StartM();
void scan();
void Calibrate();
void SendArray();

int CommandCount;
int sum=0;
int StartPos;
int EndPos;
byte CommandList[50];
int varde;
//int array[10];
task main()
{

CommandCount=0;
  while(1){
    Calibrate();
    motor[MotB]=-100;

    if (SensorValue(Light2)<40){

      motor[MotB]=0;
      //wait10Msec(100);
       scan();


       while (SensorValue(Light2)<40){
            motor[MotB]=-100;

            }
      }
```

```
  if (sum==255){
    motor[MotB]=0;
    SendArray();

    sum =0;
    break;
      }


     }
 }


 void scan()
{


  sum = 0;
  StartM();

  while(SensorValue(Rot1)<EndPos)//(SensorValue(Light2)<40)
  {


  StartM();

    if (SensorValue(Light2)<40){
         motor[MotA]=0;
          //PlaySound(soundBlip);
         wait10Msec(10);

          varde=  SensorValue(Rot1)-StartPos;
         //varde=0;

  // while ( 13<varde<47){
     /// varde= SensorValue(Rot1)-StartPos;

    if (12<varde&&varde<16)
          sum=sum+1;
     //array[0]=0;

      if (16<varde&&varde<20)
          sum=sum+2;
          //array[1]=2;

      if (20<varde&&varde<24)
            sum=sum+4;
      //array[2]=8;
      if (24<varde&&varde<28)
            sum=sum+8;
        //array[3]=16;
      if (28<varde&&varde<32)
           sum=sum+16;
        //array[4]=32;
       if (32<varde&&varde<36)
               sum=sum+32;
         //array[5]=64;
        if (36<varde&&varde<40)
            sum=sum+64;
        //array[6]=128;
```

```
        if (40<varde&&varde<44)
           sum=sum+128;
        //array[7]=256;



      while (SensorValue(Light2)<40){

      StartM();
      }



    }

  }

  //har gjort en if sats
  if(sum!=0){
  CommandList[CommandCount]=sum;
  CommandCount=CommandCount+1;
}

  SetUserDisplay(sum, 0);
  Calibrate();
  SetUserDisplay(CommandCount, 0);
}

void Calibrate()
{
  while(SensorValue(Calib3)==0)
  {
    motor[MotA]=-100;
  }
  motor[MotA]=0;
  StartPos=SensorValue(Rot1);
  EndPos=StartPos+45;
  //wait10Msec(100);
  //NextPos=StartPos+16;
}

void SendArray()
{
  int i;
/*
  for(i=0;i<=CommandCount;i++)

  {
  //PlaySound(soundLowBuzz);
    SetUserDisplay(CommandCount-i, 0);
    send_data(CommandList[CommandCount]);
  }

*/

  for(i=0;i<CommandCount;i++)

  {
  //PlaySound(soundLowBuzz);
    SetUserDisplay(CommandCount-i, 0);
    send_data(CommandList[i]);      //i was commandcount
```

```
    if (CommandList[i] == 255){      //without this program did not terminate.
      break;
    }
  }

      //send_data(CommandList[CommandCount]);
  SetUserDisplay(0, 0);        //Set to zero, statement was not here.
  PlaySound(soundLowBuzz);
}

void StartM()
{

  motor[MotA]=50;
  wait10Msec(1);
  motor[MotA]=100;


}
//Consider make bool return type and return prematuerly if some error. Make a
void send_data(byte data)
{
  byte ack;

  message = 0;
  sendMessageOld(data);   //First attempt
  wait1Msec(750);       //Give some time to get a reply
  while(!message)      //If and while no message (acknowledge), send again unt
  {
    wait1Msec(750);   //Wait time between each send attempt.
    if(!message)
      sendMessageOld(data);
  }

  ack = message;

  if (ack == data)
    PlaySound(soundFastUpwardTones);
  else
    PlaySound(soundDownwardTones);
}
```