CODEN: LUTFD2/(TFRT-5432)/1–46/(1991)

Programmoduler för styrning och identifiering av en industrirobot

Peter Nilsson

Institutionen för Reglerteknik
Lunds Tekniska Högskola
mars 1991

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
Mars 1991

Document Number
CODEN: LUTFD2/(TFRT–5432)/1-46/(1991)

Author(s)
Peter Nilsson

Supervisor
Anders Blomdell and Klas Nilsson

Sponsoring organisation

Title and subtitle
Programmoduler för styrning och identifiering av en industrirobot
(Software modules for control and identification of an industrial robot)

Abstract

Software for a robot environment consisting of microprocessor, ethernetprocessor, VME-bus, sensorball, Sun workstation and power electronics have been developed in the real-time language Modula-2. There have been different levels in the programming work. From programming close to the hardware to higher levels containing, for instance, the regulator. As an application example, a system identification have been made using the software and Matlab.

Key words
Industrial robots, real-time programming, system identification

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
Swedish

Number of pages
46

Recipient's notes

Security classification

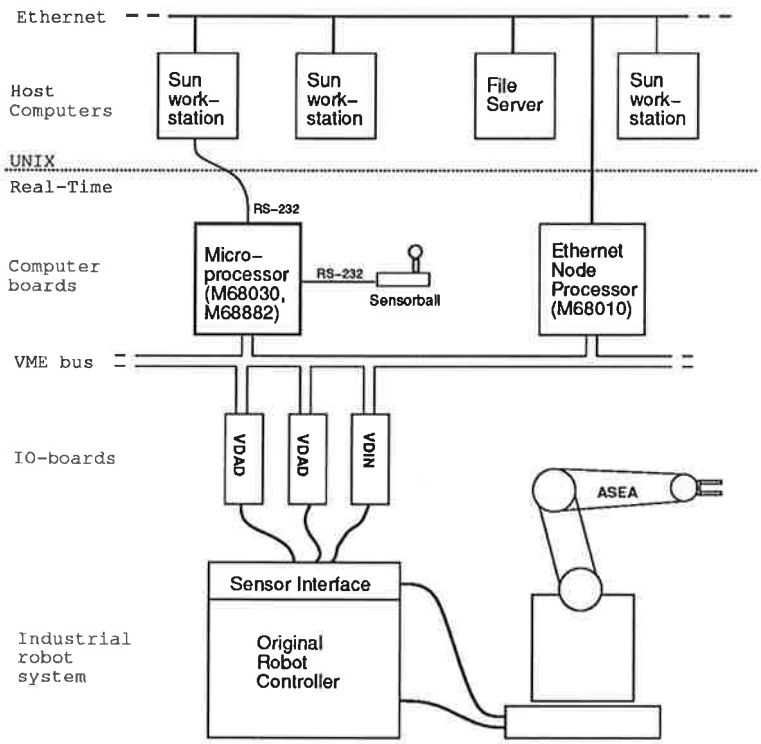
Innehåll

1 Inledning

Denna rapport beskriver mitt examensarbete med ett robotsystem vid Institutionen för Reglerteknik vid LTH. Arbetet har främst bestått av realtidsprogrammering, både på en hårdvarunära nivå och på en högre "modulariserande" nivå. Systemet som skulle kontrolleras bestod av många sammankopplade delar och programmerandet tvingade på mig insikter om modularisering och klara gränssnitt. Vidare har det ingående tillämpningsexemplet inom processidentifiering gett mig mer känsla för Bodediagram, poler och slikt. Jag vill tacka mina handledare Klas Nilsson och Anders Blomdell för deras stora hjälp och hjälpsamhet.

Målet har varit att kunna styra och experimentera med institutionens robot med hjälp av en värddator. Examensarbetet kan beskrivas i tre nivåer. På den lägsta nivån skapas gränssnitt mot omvärlden, dvs kommunikation med IO-enheter, styrutrustning, andra datorer mm. Programmen är skrivna i moduler på olika specialiceringsnivåer. Om man vill använda datorsystemet till något helt annan uppgift så använder man bara de lägsta modulerna. På nästa nivå skapas realtidsprocesser för bl a reglering, operatörskommunikation och datainsamling. På den översta nivån använder jag alltihop för att göra en processidentifiering av en robotaxel.

Förutom att prestanda och funktionalitet hos den utvecklade programvaran för realtidsstyrning och identifiering har verifierats, är slutsatsen att labmiljön bestående av Unix-baserade arbetsstationer och programvaror som utvecklingsmiljö, väl integrerad med en fristående realtidsdator, utgör en utmärkt bas för utveckling av avancerade styr- och reglersystem.



Figur 1: Översikt över systemet

2 Översikt över systemet

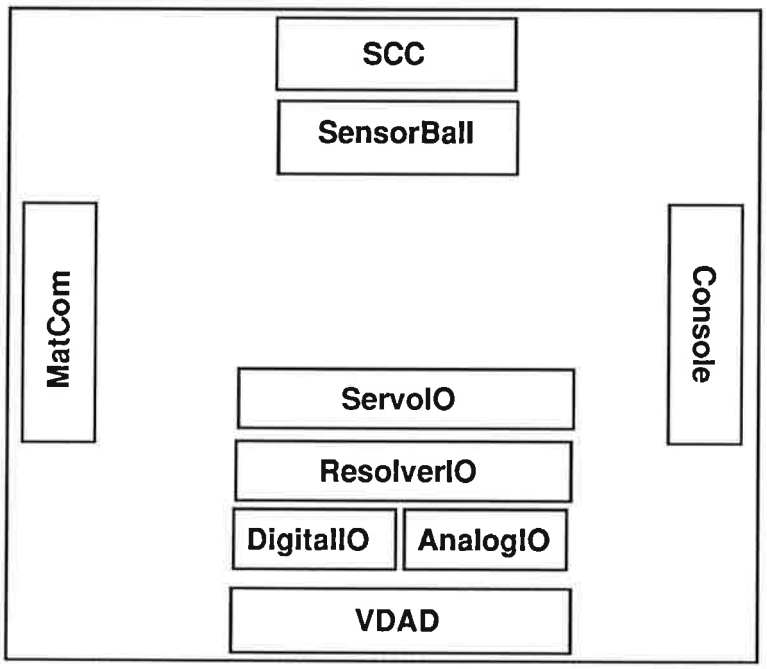
En översikt över systemet kan ses i bild 1, och ytterligare detaljer finns i (Braun, Nielsen och Nilsson, 1990).

Av det ursprungliga robotstyrsystemet används bara kraftelektroniken och nödstoppsslingan. Användaren kommer i kontakt med roboten via fönster på en Sun arbetsstation, styrspaken och nödstoppen. För uppstart och viss operatörskommunikation används en RS-232-linje kopplad till en av arbetsstationerna enligt figur 1. Även styrspaken använder seriell kommunikation. För större eller mer tidskrävande dataöverföringar används Ethernet. En Ethernet-processor är kopplad till VME-bussen och används för att skicka ner referensvärden till mikroprocessorn och för att hämta upp loggade data till Sun. Trafiken på VME-bussen består, förutom av refer-

enserna och loggdatan, av kommunikation med VDAD och VDIN korten. Är-värden kan antingen fås analogt med VDAD eller digitalt med VDIN. De digitala kanalerna på VDAD-korten används för att adressera aktuell robotled vid digital läsning, göra reset på resolvermätsystemet, styra gripdonets grip/släpp mm.

Det mesta av hårdvaran fanns tillgänglig i systemet när mitt arbete började. Det enda nya som jag tillfört är styrspaken. Den mjukvara jag fick var program för kommunikation med Sun-datorn via terminallinje och Ethernet. Språket jag använde var Modula-2. Det som finns nu är ett robotsystem speciellt utvecklat och lämpat för experimentering och identifiering. Programmeringen bestod av två delar, programvarusnitt mot omvärlden och realtidsprogrammering för robotstyrning (avsnitt 3 respektive 4).

Den största svårigheten för mig var att överblicka alla delar av systemet och hur dessa hängde ihop. Därnäst kom IO-delen där jag var tvungen att lämna den trygga, typade Wirth-världen, läsa hårdvarumanualer och dessutom tänka på tidskravet.



Figur 2: Programvarusnitt i mikroprocessorn

3 Programvarusnitt mot omvärlden

Innan huvudprocesserna kan skrivas måste hårdvaran bäddas in i en trevlig och lättanvänd förpackning så att man aldrig ska behöva tänka på sådant som adresser, bitar och avbrottshantering. Dessutom behövs en del verktyg, procedurer, för skilda uppgifter som anses ligga på en för låg nivå när realtidsprogrammeringen börjar. Hit räknas t ex initieringsfunktioner, omräkningsfunktioner och ringbuffert. Inbäddningen måste göras i nivåer så att programmen kan användas för vitt skilda uppgifter.

Med rubriken menas att vi befinner oss inne i värddatorn och att omvärlden är serieport, Ethernet, terminallinje och VME-buss, se figur 2 . Program för kommunikation med Sun-datorn medelst terminallinje och Ethernet är skrivna av folk på institutionen.

3.1 Serieporten — Styrspaken

Serieporten används för att kommunicera med den sexdimensionella styrspaken även kallad sensorbollen på grund av sin uppbyggnad med sex kraftgivare inuti en klotformig spak. Porten ger avbrott för "mottagit tecken", "bufferten tom" och olika felmeddelanden.

3.1.1 SCC–Serial Communication Controller

```
DEFINITION MODULE SCC;

 (* Serial Communication Controller *)

 (* This module transmits data to and from VME through Z8530       *)
 (* serial communication controller.                               *)

PROCEDURE InitSCC;

PROCEDURE TransmitData (     Data : CHAR );

PROCEDURE ReceiveData  ( VAR Data : CHAR );


END SCC.
```

I initieringsrutinen `InitSCC` definieras olika interna procedurer som avbrottshanterare för SCC-avbrotten. Detta göres genom anrop av `NewHandler` i modulen `Nucleus` (tillhörande den realtidskärna jag använt) med avbrottsproceduren och avbrottsvektorn som inparametrar, och med en variabel av typen `Nucleus.InterruptReference` som utparameter. Avbrottsproceduren kommer då också att anropas. Proceduren som tar emot normala tecken t ex, består då av en `LOOP` med följande innehåll; `Nucleus.Attach`, läsning av status, läsning av tecknet, inläggning i ringbuffert samt återställandet av avbrott. När något blir fel i styrspaken kan ett specialtecken erhållas. Då utförs en annan rutin där status läses, felmeddelande skrivs ut och error-bitar återställs. För att läsa inkomna tecken använder man `ReceiveData` som tar tecken ur ringbufferten (beskrivs senare). För att sända tecken, i vårt fall för att begära information från styrspaken, används `Transmitdata`. Denna procedur lägger bara sin byte (enquire-tecknet) på rätt adress. Svaret som man i bästa och vanligaste fall får från styrspaken följer ett protokoll innehållande

en del kontroller och checksummor. Det exakta protokollet kan hittas med hjälp av referenserna. Om allt går som det ska levererar styrspaken databytes via SCC som anger de tre translations och rotationsriktningarna. Denna modul är en helt allmän hantering av en serieport och för att kunna använda styrspaken hänger jag på modulen `SensorBall`. Implementationsmodulerna finns i appendix.

3.1.2 SensorBall

```
DEFINITION MODULE SensorBall;

(* Handler for the six DOF joystick, the so called sensor ball.

   The integers returned are in the range [-1,1] if interpreted as
   fixed point two complement numbers with a sign bit and remaining
   bits as fractional bits. The Button integer has one bit set for
   each button pressed according to: 0 => no buttons, 1 => button 1,
   2 => button 2, 4 => button 3, 7 => buttons 1,2,3, etc.

   The communication goes via an interrupt driven RS232 communication
   port, set to a baud rate of 9600.
*)

TYPE PositionDataType = RECORD
                           Xtransl,
                           Ytransl,
                           Ztransl,
                           Xrot   ,
                           Yrot   ,
                           Zrot   : INTEGER;
                        END;

PROCEDURE InitSensorBall;
  (* Initiate the sensor ball, the serial communication, and data buffers.
     Warning: Several interrupts with high hardware priority is serviced
     internally in the module during init. Software processes even with
     high software priority might therefore be delayed. This means you
     should not call this procedure during very critical control tasks
     with high CPU load and fast sampling (e.g. > 200 Hz)              *)

PROCEDURE ReadSensorBall ( VAR PositionData : PositionDataType;
                           VAR Buttons      : INTEGER;
                           VAR Error        : CARDINAL          );
```

```
(* Sends a read request to the sensor ball, awaits interrupt, receives
   data on interrupt, and returns joystick and key data. Error = 0
   indicates no error. Error > 0 usually indicates timeout due to not
   connected cables. A suitable rate of reading is 10 Hz  *)

END SensorBall.
```

På styrspakens låda finns knappar som gör att bara translations- eller bara rotationsriktningarna erhålles (de andra sätts till noll). En knapp har till funktion att bara det största av de sex värdena skickas då den är nedtryckt. En knapprad med 8 knappar skickas kodade som ett 8-bitars byte med högra knappen som LSB. När det inte går som det ska får programmet två problem att lösa:

1. Varför det blev fel. Spänningen till styrspaken kan t ex ha försvunnit eller inte kopplats in, varvid inget svar alls erhålles. Det kan också ha kommit en störning och då kan ett obekant tecken dyka upp.
2. Vilken åtgärd som programmet ska vidta vid de olika felen.

Om checksumman blir fel skickar proceduren automatiskt en ny begäran. Då får man tänka på att komma rätt i teckenströmmen. Men vad kan göras åt upprepade fel i kontrollsiffror eller checksummor, eller då inget svar erhålles alls? Vid felaktigt svar finns möjligheten att proceduren gör ett visst antal försök och sedan ger upp och meddelar vad som hänt i en error-parameter. Svårare är det då inget svar alls erhålles eftersom programmet då riskerar att hänga sig och inte komma vidare. För att ta hand om detta används en "Exceptionhandler". Detta är en timout-mekanism som gör att tiden som programmet får tillbringa i en viss region kan begränsas. Om programmet stannar för länge i en timeout-markerad region flyttas exekveringen helt abrupt ut ur denna. Detta löser inte bara hängningsproblemet dåinget svar alls erhålles, utan även problemet med fel i kontrollsiffror och checksummor. Överhuvudtaget verkar detta vara en bra metod vid undantagsfall för att slippa ta hänsyn till alla tänkbara fel i sin kod.

Nu kan styrspaken anses tillräckligt inbäddad för att kunna användas utan tanke på avbrott, omräkningar, checksumskontroller mm.

3.2 VME-bussen — VDAD, VDIN

Förutom datorkorten är även IO-korten VDAD och VDIN anslutna till VME-bussen. Ett VDAD-kort har 16 analoga ingångar, fyra analoga utgångar samt en digital port. Den digitala porten består av åtta bitar som bitvis kan konfigureras som in- eller utgångar. VDIN har en 16-bitars digital parallellport in.

3.2.1 VDAD mappning

För att komma åt kontrollbitar och in/ut-kanaler på VDAD och VDIN-korten deklareras variabler på de fixa adresser som bestäms av hårdvaran. Detta görs i Modula enligt

Variabel`[ADDRESS(`hexadecimal adress`)]` : typ;

Man bör ge variablerna passande namn och typer för att underlätta användandet. PBDDR betyder t ex Port B Data Direction Register. Namnet kunde varit utförligare men jag har valt att anknyta till hårdvarumanualens beteckningar. Vissa adresser, där de ensilda bitarna snarare än hela bytet är av intresse, har jag deklarerat som mängder (SET i Modula). Detta gör det lättare att skriva en del kod. Dessutom blir det vackrare eftersom man slipper bråka med binära översättningar. När man t ex önskar konfigurera en digital kanal till utgång ska rätt bit i den ovan nämnda PBDDR sättas till ett. Detta kan då skrivas

`INCL( VDAD[cardnr].PBDDR, wire );`

vilket betyder att `wire`, bit 0-7, inkluderas i PBDDR för det aktuella kortet. Viktigt att komma ihåg vid manipulering av bit-mängder är att Modula-kompilatorn numrerar bitarna i t ex `SET OF [0..7]` med bit noll som mest signifikanta bit och bit sju som den minst signifikanta i en byte. Även bitarna bör därför ges passande namn för att undvika misstag.

```
DEFINITION MODULE VDAD;

FROM SYSTEM     IMPORT BYTE, SHORTWORD, WORD, ADDRESS;
FROM Semaphores IMPORT Semaphore;

CONST Base      = 0EE0E00H;
      AdrOfDigPort1 = Base + 013H;           (* Address to PBDR on card 1 *)
      AdrOfDigPort2 = Base + 0100H + 013H;   (* Address to PBDR on card 2 *)
      NrOfCards = 2;
```

```
TYPE
  ByteSet    = SET OF [0..7];
  TwoByteSet = SET OF [0..15];
  VDADtype = RECORD
      pad1      : BYTE;                          PGCR    : BYTE;
      pad2      : BYTE;                          PSRR    : BYTE;
      pad3      : BYTE;                          PADDR   : ByteSet;
      pad4      : BYTE;                          PBDDR   : ByteSet;
      reserved1 : SHORTWORD;
      pad5      : BYTE;                          PIVR    : BYTE;
      pad6      : BYTE;                          PACR    : BYTE;

                                  .
                                  .
                                  .

  END; (* record *)

VAR
  VDAD [ADDRESS(Base)] : ARRAY [1..NrOfCards] OF VDADtype;
  AVDIN [ADDRESS(0EE0000H)] : SHORTCARD;
  Mutex                     : Semaphore;


PROCEDURE InitVDAD;
(* Initializes the VDAD cards *)

END VDAD.
```

En del initieringar såsom att sätta alla bitar i en port till utgångar görs i `InitVDAD`. Tack vare namngivningen och mappningen kan man enkelt, och kompakt skriva olika IO-rutiner. Proceduren `DAout` t ex som lägger ut ett värde `value` (spänning) på kort `cardnr`, kanal `channel` består bara av en rad:

```
VDAD[cardnr].DACOR[channel] := value + 2048;
```

Nu kan vi alltså glömma adresser och använda namnen för att skriva procedurer för analog och digital IO-hantering. Lägg märke till att deklareringen av VDAD i mappningen med basadresser gör att implementationen av procedurerna för analog och digital IO blir oberoende av antal kort och att kod och tidskrävande case- alternativt ifsatser inte behövs. Dock krävs att samtliga VDAD-kort har (genom bygling på kortet) givits konsekutiva adresser. Detta

för att de skall utgöra en `ARRAY[1..NrOfBoard]` av VDAD-kort.

3.2.2 AnalogIO och DigitalIO

Som ett skikt ovanpå VDAD-modulen som beskriver hårdvaran, har moduler för generell användning av IO-korten utvecklats. Dessa modulers snitt mot användaren är både hårdvaruoberoende och tillämpningsoberoende. Kortens egna skalningar av in- och utvärden används dock av effektivitetsskäl.

```
DEFINITION MODULE AnalogIO;

(* Analog IO via the VDAD boards from PEP computers.

   Initialize the VDADs in one and only one of the following ways:
   1. Call InitServoIO in the module ServoIO.
   2. Call InitResolver in the module ResolverIO.
   3. Import VDAD, call InitVDAD, and set up the control registers
      on the board. Refer to VDAD reference manual for details
   Alternatives 1 and 2 configures the analog ports to operate in
   the range +/- 10V.
*)

FROM VDAD IMPORT NrOfCards;

TYPE CardType     = [1..NrOfCards];
     ChannelType  = [0..15];
     DARange      = [-2048..2047];
     IntArray     = ARRAY ChannelType OF INTEGER;
     ExpGainType  = [0..2];
     OutRangeType = [0..1];

PROCEDURE ADin  (     cardnr  : CardType;
                      channel : ChannelType;
                  VAR value   : INTEGER      );

PROCEDURE DAout (     cardnr  : CardType;
                      channel : ChannelType;
                      value   : DARange      );

PROCEDURE MultiADin (     cardnr      : CardType;
                          LowChannel,
                          HighChannel : ChannelType;
                      VAR value       : IntArray     );
```

```
PROCEDURE SetInputGain( cardnr  : CardType;
                        ExpGain : ExpGainType );
  (* ExpGain = 0, 1, 2  =>  Input gain = 1, 10, 100 *)

PROCEDURE SetOutVoltage( cardnr     : CardType;
                         channel    : ChannelType;
                         Gain,
                         UniBiPolar : OutRangeType );
  (* Outvoltage range = Vref * (1 + Gain)  *)
  (* UniBiPolar: 0 = unipolar, 1 = bipolar *)

END AnalogIO.

DEFINITION MODULE DigitalIO;

(* Digital IO via VDAD boards and the VDIN board from PEP computers.

   Initialize the VDADs in one and only one of the following ways:
   1. Call InitServoIO in the module ServoIO.
   2. Call InitResolver in the module ResolverIO.
   3. Import VDAD, call InitVDAD, and set up the control registers
      on the board. Refer to VDAD reference manual for details
   Alternatives 1 and 2 configures the digital ports to be output
   on board number one, and input on board number two.

   The VDIN board requires no initialization. To read the 16 bit
   parallell input port, call DigInput.
*)

FROM VDAD   IMPORT NrOfCards;
FROM SYSTEM IMPORT BYTE;

TYPE CardType = [1..NrOfCards];
     WireType = [0..7];
     Bit      = [0..1];
     Byte     = [0..255];

PROCEDURE DigWireOutput (     cardnr : CardType;
                              wire   : WireType;
                              value  : Bit      );

PROCEDURE DigWireInput  (     cardnr : CardType;
                              wire   : WireType;
```

```
                          VAR value  : CARDINAL );

PROCEDURE DigByteOutput (     cardnr : CardType;
                              value  : BYTE     );

PROCEDURE DigByteInput  (     cardnr : CardType;
                          VAR value  : BYTE );

PROCEDURE DigInput( VAR value : SHORTINT );
  (* value = [-32768, 32767] *)

END DigitalIO.
```

Definitionsmodulerna får tala för sig själva. Vad som kan kommenteras är att `MultiADin` använder en mod på VDAD-kortet som gör att flera kanaler i följd kan läsas extra snabbt. Med `SetInputGain` och `SetOutVoltage` kan inspänning respektive utspänning förstärkas. Utspänning kan väljas uni- eller bipolär. Beträffande programmeringen kan sägas att ett hårt typat språk som Modula är klumpigt på denna nivå. När man har att göra med adresser och bitar vill man inte förknippa någon typ med dessa (se t ex implementationen av `DigWireInput` i appendix för ett exempel på klumpighet).

3.2.3 ResolverIO

Resolvermodulen använder IO-procedurerna för att kontrollera resolvrarna och ge tal som motsvarar motor- och robotvinklar. Robotvinklarna är en förskjutning av motorvinklarna ett helt antal varv, därför behövs parametern `addfact` som anger vilket varv motorn är på. Denna modul är tillämpningsberoende och IO-kortens skalning av signalerna finns kvar.

```
DEFINITION MODULE ResolverIO;

IMPORT VDAD;

FROM SYSTEM IMPORT ADR, ADDRESS;

TYPE FiveIntArray =  ARRAY [1..5] OF INTEGER;
     ResolverType =  [1..5];
     BitState = (Active, Inactive);
     (*$NONSTANDARD*)
```

```
     DigPort1Type      = PACKED RECORD
                              setbusy  : BitState;
                              gripper  : (undef1, Closed, Open, undef2);
                              reset    : BitState;
                              read     : BitState;
                              adr      : [0..7];
                         END; (* packed record *)
     (*$STANDARD*)
(* robotvalue = [-32768,32767] + addfact  *)
(* motorvalue = [-32768,32767]            *)
(* addfact    = +/- motorturns * 2^16     *)

VAR DigPort1[ADDRESS(VDAD.AdrOfDigPort1)] : DigPort1Type;

PROCEDURE InitResolver;

PROCEDURE InitDigPosition (     resolvernr  : ResolverType;
                           VAR motorvalue,
                               addfact     : FiveIntArray );

PROCEDURE InitAnPosition  (     resolvernr  : ResolverType;
                           VAR motorvalue,
                               addfact     : FiveIntArray );

PROCEDURE DigPosition (     resolvernr  : ResolverType;
                       VAR robotvalue  : INTEGER;
                       VAR motorvalue,
                           addfact     : FiveIntArray );

PROCEDURE  AnPosition (     resolvernr  : ResolverType;
                       VAR robotvalue  : INTEGER;
                       VAR motorvalue,
                           addfact     : FiveIntArray );

PROCEDURE Digital5Pos ( VAR robotvalue,
                           motorvalue,
                           addfact     : FiveIntArray );
PROCEDURE Analog5Pos  ( VAR robotvalue,
                           motorvalue,
                           addfact     : FiveIntArray );

(*PROCEDURE Speed (     resolvernr : ResolverType;
                   VAR value      : REAL          ); *)
```

```
END ResolverIO.
```

Modulen kan läsa av resolvrarna både analogt och digitalt. Den digitala avläsningen görs med hjälp av AVDIN-kortet. Den är snabbare och exaktare men kräver lite mer kod. Den digitala läsningen ger 16 och den analoga 12 bitar efter AD-omvandlingen i VDAD-kortet. Vid synkroniseringen används `InitAnPosition` eller `InitDigPosition` i vilka resolvermätsystemet nollställs. För att styra resolvermätsystemet används sju digitala kanaler på VDAD-korten, sex utgångar och en ingång:

- **reset**, som nollställer resolvermätsystemet för resolvern med adress **adr**.
- **read**, initierar en läsning.
- **setbusy**, startar en timer som sätter ready (se nästa bit) när datan har stabiliserat sig.
- **ready** (enda ingången), går hög när AVDIN kan läsas.
- **adr**, tre bitar för adressering av önskad resolver.

Utgångarna ligger i samma byte i mikroprocessorn. Om dessa deklareras i en PACKED RECORD, se ovan, kommer de i samma byte och de kan sedan nås med punktnotation. Detta är ett trevligt alternativ till bitmängder men tillhör inte standard för Modula-2. En läsning t ex initieras t ex så här smidigt:

```
DigPort1.read := Active;
```

Den exakta gången vid initiering och läsning kan studeras i implementationsmodulen i appendix. Den analoga läsningen är enklare, bara ett anrop av ADin från `AnalogIO` plus en omvandling från 12 till 16 bitar (en multiplikation med 16 som jag hoppas att kompilatorn utför som ett skift fyra steg). Tyvärr blir resolvrarnas värden inte en entydigt växande funktion över robotaxelns arbetsområde utan den slår om och ger alltså en sågtandsfunktion. Därför måste `addfact`-parametern finnas. Denna håller reda på hur många gånger omslag kommit. För att detektera omslaget måste värdet förra gången resolvern avlästes också skickas med. Om en alltför stor ändring skett från

förra gången tolkas detta som ett omslag. Beroende på åt vilket håll axeln vreds ökas eller minskas `addfact`.

Vanligt vid robotstyrning är att man vill ha alla fem resolvrarna lästa samtidigt. Det finns särskilda processer för detta (procedurnamnen innehåller en femma). Det finns hårdvaruunderstöd för att läsa flera resolvrar snabbt både analogt (`MultiADin`) och digitalt. Detta ger en väsentlig tidsvinst eftersom man också sparar in fyra proceduranrop. Vid digital läsning fås dessutom samtliga fem världen vid exakt samma tidpunkt.

3.2.4 ServoIO

Denna översta modul i kapslingen av snittet mot processen ger skalning till SI-enheter och procedurer för hantering av utsignaler och robotens mikrobrytare för synkronisering. Definitionen av denna modul är oberoende av hårdvarans skalningar, men den är av naturliga skäl processberoende och tillämpningsberoende

```
DEFINITION MODULE ServoIO;

(* Interface module to drive units and position sensors of the modified
   ASEA IRB-6 system, and joint compatible with that system.

   Upon init (by calling InitServoIO) the following happens:
   1. The scaling of RefOut is set to either MaxVel or MaxTorque
      depending on mode of operation. With the drives in PI or
      P-servo mode, the reference is a velocity reference, and
      InitServoIO should be called with MaxRef=MaxVel. With the
      drives in Ext mode, the reference is a torque reference, and
      InitServoIO should be called with MaxRef=MaxTorque.
   2. Analog or digital position sensor interface is selected by
      assigning the proper procedures to the procedure variables
      below. This way, the same procedures (i.e. the procedure
      variables absPos, incPos etc.) can be called for both analog
      and digital interface, and without any test of mode in the
      implementation module.
   3. The motor revolution counter in the resolver interface hardware is
      reset to zero.
   4. The internal counter in this module used for keeping track of
      the resolver-interface "turns" is reset to zero.
   The counters can also be reset by calling ResetServo. The zero position
   is then defined to be at motor angle zero IN THE CURRENT MOTOR TURN.
   This is needed for syncronization of the position measurement system.
```

```
   The incremental position values can be used to get the speed, or simply
   to reduce the magnitude of signals by using an incremental controller.

   The procedures with names containing a '5' can be used for fast
   sampling of joints 1..5. All values are then sampled at exactly the
   same time.

   To fully use this module, two VDAD boards and one VDIN board (PEP
   boards) is required. With fewer boards, be careful not to address
   a non-exsisting board, which will result in a bus-error exception.
*)

CONST MaxVel = 314;    (* rad/s => 3000 rpm *)
      MaxTorque = 1.3; (* Nm => 15 A motor current. *)

TYPE JointType      = [0..5];   (* 0 for external joint. 1..5 for IRB-6 *)
     AnalogOutputs  = [1..8];
     Array5Real     = ARRAY [1..5] OF REAL;
     Array5Longreal = ARRAY [1..5] OF LONGREAL;
     ReadMode       = ( analog, digital );
     absPosProc     = PROCEDURE ( JointType ) : LONGREAL;
     incPosProc     = PROCEDURE ( JointType ) : REAL;
     abs5PosProc    = PROCEDURE ()            : Array5Longreal;
     inc5PosProc    = PROCEDURE ()            : Array5Real;
     ZeroPosProc    = PROCEDURE ( JointType );

VAR absPos     : absPosProc  ;  (* absolute motorangle in radians       *)
    incPos     : incPosProc  ;  (* incremental motorangle in radians    *)
    abs5Pos    : abs5PosProc ;  (* 5 absolute motorangles in radians    *)
    inc5Pos    : inc5PosProc ;  (* 5 incremental motorangles in radians *)
    ResetServo : ZeroPosProc ;  (* resets the motorturn counter         *)


PROCEDURE InitServoIO ( AnOrDig : ReadMode; MaxRefOut : LONGREAL );


PROCEDURE RefOut ( joint : AnalogOutputs;  value : LONGREAL );
  (* Output value is limited to [-MaxRefOut, MaxRefOut].
     For joints not present, RefOut can be used as AnalogOut with
     built in scaling and saturation.                                  *)


PROCEDURE ZeroSense ( joint : JointType ) : CARDINAL;
```

```
(* Returns a value in the range [0,1] depending on the state of the
   syncronization switches on the robot. 1 means switch is closed. *)

END ServoIO.
```

I `InitServoIO` bestämmer man om inläsningen ska vara analog eller digital. absPos, incPos osv använder sedan de analoga eller digitala inläsningsprocedurerna som skapats tidigare. Beroende på valet i initieringen fungerar alltså de tillhandahållna procedurerna på helt olika sätt. Detta åstadkomms genom att använda procedur-variabler. Proceduren `absPos` t ex, som ska ge absoluta robotvinklar, är deklarerad som en procedur-variabel, se ovan. I initieringsproceduren blir tilldelningen

```
absPos := absPosAnalog;
```

om man valt analog läsning och naturligt nog

```
absPos := absPosDigital;
```

om man valt digital. Härigenom behövs ingen test under exekveringen av vilken procedur som ska anropas internt. Dessutom berörs inte användarprogrammet av vilket snitt som används mot processen (annat än vid initieringen). Vid initieringen anger man dessutom maximal vinkelhastighet. `RefOut` lägger ut styrsignal med hjälp av `DAOut` på rätt kort och kanal.

4 Realtidsprogram för robotstyrning

Nu är vi redo för nästa steg, att skriva program som ger oss möjlighet att styra roboten. Eftersom det finns flera parallella aktiviteter, t ex asynkron operatörskommunikation och synkron reglering, används realtidsprogrammering. I ett experimentellt system som detta läggs inte så stor vikt vid Man-Maskin kommunikationen. Den sparsmakade operatörskommunikationen innebär att man väljer olika åtgärder genom att trycka på en tangent, varefter man får någon liten ledtext om vad som ska matas in.

4.1 Kontrollstruktur

Någon process måste ha kommandot och här är det `OpCom` som bestämmer hur och om en process ska köras.

Regulatorn har en logisk variabel som stänger av och sätter på regleringen. Om regulatorn är på används referensen och är-värdet till att räkna ut ut styrsignal. Om regulatorn är avstängd läggs referensen ut direkt (styrsignalen är alltså vinkelhastigheten). Nolla som referens ger stillastående robot. Vid synkronisering ger man en måttlig hastighet med tecken enligt synkbrytarens status.

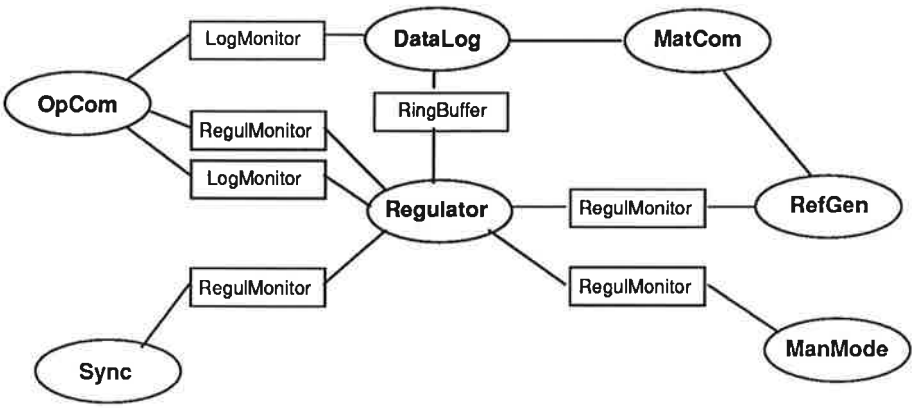
Man kan antingen köra manuellt med styrspaken eller generera referenser i Matlab. Om manuell mod väljs startas en process som utnyttjar det förut beskrivna gränssnittet med `SCC` och `SensorBall` och om Matlab-mod väljs används `MatCom` för att sända referenser till `RefGen` via Ethernet.

Processen `DataLog` som tömmer ringbuferten på loggade värden, hänger i ringbufferten tills det finns något att hämta. Då får den dessutom information från `OpCom`, om hur datan ska tas omhand. Denna information kunde istället ha placerat först i ringbufferten enligt något protokoll.

Synkroniseringsprocessen, `SyncRobot` ligger och väntar på en semafor då den inte används.

4.2 Datastruktur

En del data ska passera mellan processerna. Från `OpCom` till `Regulator` måste naturligtvis regulatorns parametrar och samplingsintervallet skickas. Regulatorparametrarna är ibland inte desamma för en regulator som för operatören. Vid PID-reglering t ex vill operatören mata in h, K, T_i, T_d och N



Figur 3: Realtidsprocesser och deras kommunikation

medan regulatorn vill ha h, K, T_i, a och b för att inte behöva utföra samma beräkningar varje period i denna tidskritiska sektion. Omformning görs då enkelt i monitorn. För att inte behöva läsa in alla parametrarna varje regulatorperiod finns en logisk variabel som man får ur samma procedur som ger referens och den logiska av/på . Denna procedur måste ju ända anropas varje period. Om variabeln är sann läses de nya parametrarna in.

Från `Regulator` till `DataLog` strömmar de loggade variablerna via en ringbuffert. De arrangeras i lagom stora matriser med en kolonn för varje variabel och skickas sedan med hjälp av modulen MatCom till Matlab på Sun via Ethernet.

4.3 Processer

I figur 3 ges en översikt över de ingående processerna och deras monitorer.

`OpCom` är en process för kommunikation mellan VME-datorn och operatören via ett X-fönster på SUN och tangentbord. I mitt system är den också huvudprocessen med ansvar för initieringar och start/stopp av körning. Först initieras bland annat realtidskärnan, en del monitorer och gemensam data såsom regulatorparametrar får default-värden. Vidare läggs alla utsignaler till noll. Man väljer sedan om man vill använda digital eller analog inläsning och om man vill styra manuellt eller om styrsignal ska genereras med t ex

Matlab. Därefter startas de övriga processerna och operatörsprocessen lägger sig att vänta på ett tecken från tangentbordet. Order som kan ges är synkronisera, visa och ändra regulatorparametrar, visa och ändra samplingstid, logga och stänga av regulatorer.

`Regulator` är en process för att reglera ett antal motorer som i detta fall sitter på en industrirobot. Själva regulatorn blir försvinnande liten med tanke på det totala antalet programrader som skrivits. Detta gör det lätt att experimentera med olika regulatorer. I regulatorn finns en sektion för loggning av intressanta värden. Sektionen, som bort göras till egen modul, ligger här eftersom regulatorn har högst prioritet och många intressanta variabler finns här. Tekniken med procedur-variabler kunde ha använts; När inga variabler ska loggas är logg-proceduren en tom procedur och när något ska loggas tilldelas den en lämplig procedur.

Målet har hela tiden varit att öka frekvensen hos regleringen. För att kontrollera att regleringen hinner med jämförs kärnans klocka med variabeln T som innehåller tidpunkten då nästa reglering ska börja. Om tiden T är passerad ökas T så att regleringen kommer ifatt. Om detta behövt göras ett visst antal gånger i följd ökas periodtiden en tidsenhet. Huvudprocessen tittar på samplingsintervallet då den får körtid och meddelar om förändring av samplingsintervallet skett.

`ManualMode` är processen som använder `ReadSensorBall`. Värdena från styrspaken skalas. För att ge större känslighet vid krypkörning kan en kvadratisk skalningsfunktion användas. De skalade värdena skickas sedan till regulatorn som referenser. Ingen kinematik har använts utan roboten styrs ledvis. Knappraden skulle kunna användas till hastighetsinställningar, dvs ändra skalningen, och/eller för att styra ett gripdon.

`SyncRobot` är en process som ställer roboten i synkläge vilket definierar målsystemets nolläge. När man synkroniserar ska regleringen stängas av och referensen, vinkelhastigheten, sättas till noll. När axeln ska vridas till synkläge läggs en måttlig vinkelhastighet ut. Varje axel har en digital synksignal som slår om i en viss vinkel. Det är viktigt, i riktiga tillämpningar, att vinkeln noll är på samma ställe efter varja synkning. Därför ska alltid nollställning av resolvern ske vid samma flank för varje axel (t ex omslag 0

till 1). Detta för att undvika hysteres. På grund av störningar behövs "filter" som kräver att omslaget ska vara fast ett visst antal sampel i följd. När en axel är synkad läggs referensen till noll, regleringen slås på och efter att ha synkat alla axlarna hänger sig processen på en semafor.

`DataLog` plockar data ur en ringbuffert som fylls på i regulatorprocessen. När en viss mängd data kommit skickas denna i form av en matris till Matlab på Sun via Ethernet. Varje variabel som samplas har en kolonn i matrisen. Datalogprocessen har inga tidskrav, om ringbufferten är rimligt stor, och har därför låg prioritet.

`RefGen` sköter kontakten med MatCom men kan också generera slump- och sinussekvenser själv. Pseudo-slumpsekvensen skapas med

slumptal := decimaldelen(slumptal * 147)

där slumptal initieras till ett tal mellan 0 och 1. För sinussekvensen kan period och amplitud väljas.

4.4 Processkommunikation

Som sig bör i realtidssammanhang måste variabler som används av flera processer skyddas. Brevlådor, monitorer och semaforer kan användas. Monitorer kan ses som strukturerad användning av semaforer och denna metod har jag använt. Dessutom kan en ringbuffert användas för enkelriktad data, t ex mellan regulatorn och loggprocessen.

`RegulMonitor` innehåller data som har att göra med reglering samt operationer på dessa. Regulatorn sitter som spindeln i nätet och behöver kunna kommunicera med nästan alla övriga processer. Den gemensamma datan utgörs av referensvärden, regulatorparametrar, samplingstid och en logisk on/off variabel. Operationerna består av procedurer som lägger in och tar ut denna gemensamma data. Regulatorparametrar räknas om till en, för regulatorn, effektiv form.

`LogMonitor` innehåller data som anger vilka variabler som ska loggas, vilken axel som ska loggas och om loggning ska ske varje sampel eller med annan

period. Operatörsprocessen ställer in loggordern och loggprocessen och loggsektionen i regulatorn läser den.

`RingBuffer` är en modul där flera ringbuffertar kan skapas. Vid initieringen anger man en vektor av valfri typ och längd som parameter. När man använder bufferten skickar man med vektorn (eller rättare, en pekare till den) för att ange vilken buffert man vill åt. Man är inte bunden till att data ska vara av en särskild typ. Den behöver inte ens vara av samma typ som vektorn som initierade bufferten. Det reserveras nämligen bara den plats som en sådan vektor behöver. Förutom de självklara funktionerna att lägga in och ta ut data finns två logiska funktioner, en som anger om mer data finns att hämta och en som anger om det finns plats för en variabel till.

```
DEFINITION MODULE RingBuffer;

FROM SYSTEM IMPORT BYTE;

PROCEDURE InitRing  ( VAR r : ARRAY OF BYTE );

PROCEDURE MoreData  ( VAR r : ARRAY OF BYTE )     : BOOLEAN;

PROCEDURE MoreSpace ( VAR r    : ARRAY OF BYTE;
                      Variable : ARRAY OF BYTE ) : BOOLEAN;

PROCEDURE PutData   ( VAR r     : ARRAY OF BYTE;
                          Value : ARRAY OF BYTE );

PROCEDURE GetData   ( VAR r     : ARRAY OF BYTE;
                      VAR Value : ARRAY OF BYTE );


END RingBuffer.
```

Modulen reserverar en, för ens behov, tillräckligt stor minnesarea. För att komma från typbekymren deklareras den som `ARRAY [0..Size] OF BYTE`. I början på vektorn som användaren vill ha som ringbuffert smyger man, i initieringsproceduren, in ett ringhuvud där det lagras lite administrativ information. Där finns ett skriv- och ett läsindex som anger i vilket byte skrivning respektive läsning ska påbörjas. Vidare finns buffertens storlek där så att man vet när omslag från slut till början av vektorn ska ske. Ett

ställningstagande måste göras om vad som ska ske vid läsning då inget finns att läsa. Jag valde att låta anropet bli hängande på en semafor. Motiveringen är att om man anropar bufferten så vill man ha något och kan vänta och eftersom `MoreData` finns får man faktiskt skylla sig själv annars. Om man inte hade hängt sig hade man varit tvungen att införa in NoMoreData-parameter. Den som verkligen ville ha något ur bufferten fick då lägga sig och titta på denna parameter i en realtids-vansinnig slinga. Inläggning och utplockning ur bufferten sker byte-vis och hela tiden kontrolleras att läs- och skrivindex inte kolliderar. Lägg märke till att man, eftersom all data behandlas byte-vis, kan deklarera vektorn som exempelvis CARDINAL och sedan ändå lägga in CHAR-deklarerade tecken. Detta kräver lite disciplin hos användaren men kan utnyttjas för inledande information, t ex text. Man slipper då koda detta till samma typ som datavariablerna.

5 Tillämpningsexempel - processidentifiering

För att sätta modulerna på prov har jag gjort en identifiering av den första axeln på den ASEA IRB-6 industrirobot som reglerats. De övriga är inte svårare att experimentera med, men resonanser märks tydligast på den första. Alla experiment utfördes i sluten loop med PI-regulator. Strömmen till motorn är insignal och vinkelhastigheten är utsignal. Båda dessa storheter, som har uttag på styrskåpet, kopplas till VDAD-kortet. Referensvärdena genererades i Matlab på Sun och skickades sedan via Ethernet till VME-datorn. Slumpsekvenser och svept sinus användes som referenser. Varje inspelning var omkring 4 sekunder lång. För att inte återkopplingen ska inverka och för att alla frekvenser ska exciteras bör roboten skaka ganska mycket vid inspelningen. När datan är loggad och uppskickad till Matlab, (se figur 4) börjar den andra delen av identifieringen, att vaska fram frekvensfunktioner och modeller för axeln ur mätserierna.
Matlab innehåller programpaketetet System Identification Toolbox och detta har används flitigt. Det första man bör göra är att kontollera att utsignalen åstakommits av insignalen (på ett linjärt sätt). För denna uppgift används ofta Koherensfunktionen

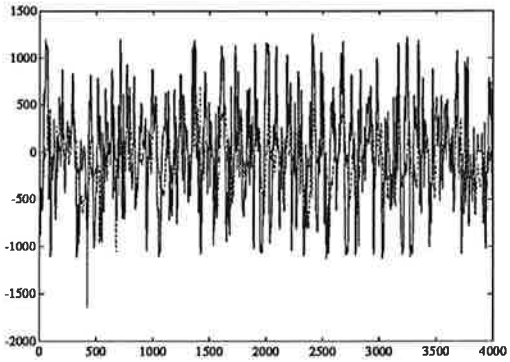
$$\Gamma_{yu} = \frac{|S_{yu}(i\omega)|^2}{S_{yy}(i\omega)S_{uu}(i\omega)}$$

där $S_{yu}(i\omega)$ är korspektrum mellan in- och utsignal och $S_{yy}(i\omega)$ och $S_{uu}(i\omega)$ är autospektrum för ut- och insignalen. Om denna funktion inte är nära ett tyder detta på att det inte finns någon linjär överföringsfunktion. Detta kan bero på olinjäriteter eller på mycket störningar på in- eller utsignalen. I figur 5 ser man att den inspelade datan är att lita på upp till ungefär 500 rad/s.

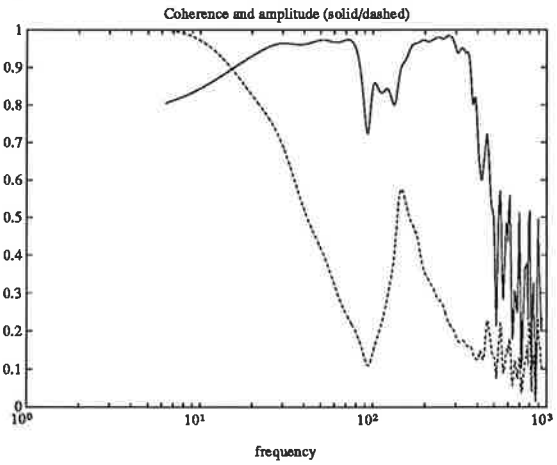
Man kan redovisa resultatet från en identifiering på två olika sätt, antingen som ett bodediagram eller som parametrar i en modell (icke-parametrisk respektive parametrisk identifiering). I den parametriska identifieringen kan man anpassa modellens parametrar antingen i tidsplanet eller i frekvensplanet.

5.1 Icke-parametriska metoder

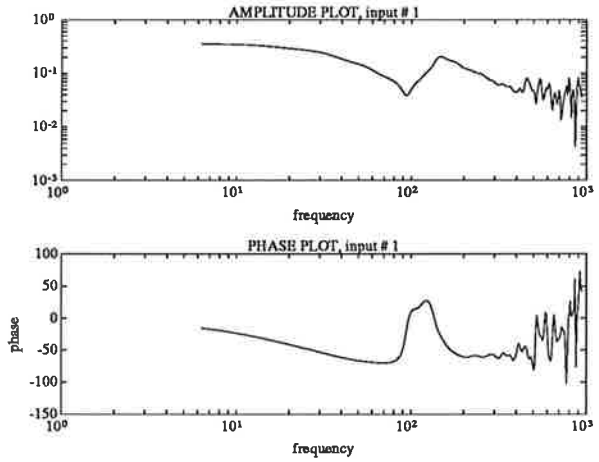
Överföringsfunkionen kan skattas med korsspektrum genom insignalens autospektrum. Denna spektralanalys ger Bodediagrammet i figur 6.



Figur 4: Slumpgenererad insignal och därtill hörande utsignal



Figur 5: Koherensfunktionen och amplitudfunktionen



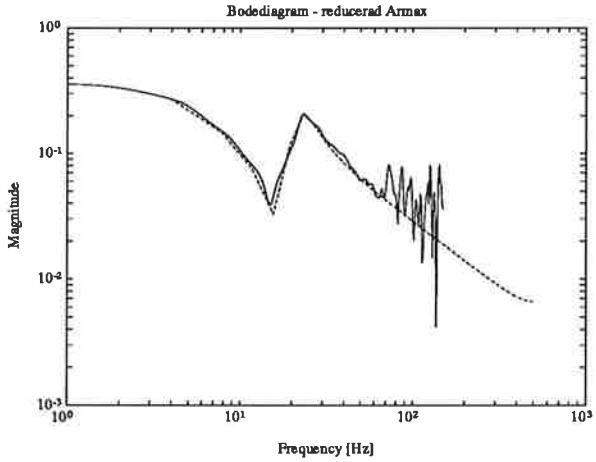
Figur 6: Bodediagrammet för axel 1

5.2 Parametriska metoder i tidsplanet

Här finns flera olika modeller att välja mellan, t ex:

ARX:	$A(q^{-1})y(t) = B(q^{-1})u(t-k) + e(t)$
ARMAX:	$A(q^{-1})y(t) = B(q^{-1})u(t-k) + C(q^{-1})e(t)$
OE:	$y(t) = \frac{A(q^{-1})}{B(q^{-1})}u(t-k) + e(t)$

När man har valt modell och låtit en algoritm anpassa sin tidsserie vill man gärna veta om modellen blivit tillräckligt bra. Ett sätt är att jämföra modellens frekvensfunktion med resultatet från en spektralanalys (icke-parametrisk metod). I Bodediagrammet bör då kurvorna följa varandra väl. Modellens rest utgör en funktion, residualen, och denna ska vara vitt brus (utom för tidsförskjutning noll). Vidare ska inte residualen bero av tidigare insignaler, dvs $E(u(t)e(t+T))$ ska vara noll för $T > 0$. Den sista och självklaraste metoden är simulering. Modellens utsignal jämförs med den verkliga. Man bör då använda olika mätserier för identifiering och verifiering. Efter lite

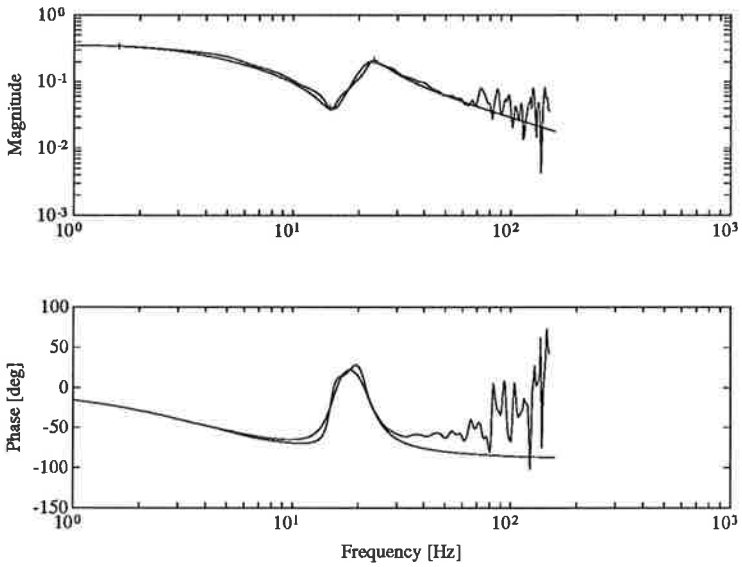


Figur 7: Spektralanalyserad data och reducerad Armax

experimenterande upptäcker man att en ARMAX-modell av ordning fem har en frekvensfunktion som följer den spektralanalyserade datan bra i ett Bodediagram. För att kontrollera om en onödigt hög modellordning har använts kan man omforma modellen till en balanserad realisering. I detta fall ser man då att vissa tillstånd, relativt sett, har mycket lite att göra med insignal-utsignalsambandet. Detta, tillsammans med att Bodediagrammet antyder lägre ordning, gör att man gärna försöker göra en reducering. Det visar sig att man med gott samvete kan stryka två tillstånd utan att det märks i Bodediagrammet. Figur 7 visar hur väl den tredje ordningens modell som erhållits genom att reducera en femte ordningens modell följer den spektralanalyserade datan.

5.3 Parametriska metoder i frekvensplanet

ARMAX krävde för hög ordning (utan reducering) på grund av att den viktade alla frekvenser lika mycket. Det vore därför önskvärt att kunna föra över lite kunskap från identifieraren till identifieringsalgoritmen så att denna förstod vad som var "viktigt". Detta kan göras med direkt minstakvadratanpassning i frekvensplanet. För detta har jag utvecklat ett interaktivt Matlab-makro i vilket ett Bodediagram ritas och ur detta väljs sedan



Figur 8: Spektralanalyserad data och modell från MK i frekvensplan

med musklickningar intressanta punkter såsom resonanser, antiresonanser och lågfrekvensnivån. Därefter väljs gradtal på A- och B-polynomen samt tidsfördröjning. Som resultat ger makrot den beräknade överföringsfunktionens frekvensfunktion i samma Bodediagram som man utgått från. Dessutom plottas pol-nollställesdiagrammet och polynomen skrivs ut. För min spektralanalyserade data klickar jag på resonansen, antiresonansen och på en låg frekvens. Antal nollställen väljs till två (ett komplexkonjugerat nollställe), antal poler väljs till tre (en komplexkonjugerat pol och integrator) och tidsfördröjningen tas till ett halvt sampel. Denna metod gav omedelbart ett bra resultat vilket kan ses i figur 8.

6 Referenser

KING K.N. (1988): *Modula-2 A Complete Guide*

SÖDERSTRÖM S. (1984): *Lecture Notes in Identification*

JOHANSSON R. (1990): *Modelling and Identification*, Kompendium

ERIKSSON P. (1989): *Kompendium i Digital Signalbehandling*

Motorola (1987): *MVME133A-20 VMEmodule, 32-Bit Monoboard Microcomputer, User's Manual*

PEP Modular Computers (1988): *VDAD, Universal Analog to Digital/Digital to Analog I/O Module for the VMEbus, User's Manual*

Oregon Software (1989): *Oregon Modula-2 Cross-Development System, M68000- Family Targets, Version 1.0, User's Manual*

The MathWorks Inc (1987): *Pro-Matlab* och *System Identification Toolbox*

BRAUN R., NIELSEN L. OCH NILSSON K. (1990): *Reconfiguring an ASEA IRB-6 Robot System for Control Experiments*

NILSSON K. (1990): *MatComm, A Matlab Interface to Real Time Control Systems*

NILSSON P. (1990): *Identifiering av robotaxel*

robot (mekanisk människa); efter KARELL ČAPEKS skådespel *R. U. R. Rossum's universal robots* (Praha 1921), som framställer ett samhälle med konstgjorda arbetare, robotar; av tjeckiska *robota*, arbete, fornslaviska *rab*, träl.

7 Appendix

Här följer

- implementationsmodulerna för de moduler som beskrevs i *Programvarusnitt mot omvärlden*,
- implementationsmodulen för ringbufferten som använts mellan `SCC` och `SensorBall` och mellan `Regulator` och `DataLog`,
- några Matlab-makron som jag gjort för identifieringsarbetet.

```
IMPLEMENTATION MODULE SCC;

IMPORT Nucleus;
IMPORT SYSTEM;
IMPORT Debug;
FROM   SYSTEM     IMPORT BYTE;
FROM   Console    IMPORT GetChar, PutChar, PutString, PutLn;
FROM   RingBuffer IMPORT InitRing, PutData, GetData;

TYPE
     RR1Bits = ( EOF, CRC, RxO, Parity, RC0, RC1, RC2, AllSent );
     RR1Set  = SET OF RR1Bits;
VAR
  SIOBWRO     [SYSTEM.ADDRESS(0FFFA0000H)] ,
  SIOBRRO     [SYSTEM.ADDRESS(0FFFA0000H)]  : BYTE;
  SIOBRXDATA  [SYSTEM.ADDRESS(0FFFA0001H)] ,
  SIOBTXDATA  [SYSTEM.ADDRESS(0FFFA0001H)]  : CHAR;
  TBEHandler, ESCHandler, RCAHandler, SRCHandler : Nucleus.InterruptReference;
  RR1                    : RR1Set;
  ch                     : CHAR;
  MoreDataToTransfer     : BOOLEAN;
  OutputData             : CHAR;
  Buffer                 : ARRAY [0..1000] OF CHAR;

(*-------------------------------------------------------------------------*)
PROCEDURE TransmitData( Data : CHAR );

BEGIN
  SIOBWRO := BYTE(01H);SIOBWRO := BYTE(13H);(* enable transmitter interrupt *)
  SIOBTXDATA := Data;                       (* transmit a character         *)
END TransmitData;
(*-------------------------------------------------------------------------*)
PROCEDURE ReceiveData  ( VAR Data : CHAR );

BEGIN
  GetData( Buffer, Data );
END ReceiveData;

(*-------------------------------------------------------------------------*)
PROCEDURE TBEinterrupt;   (* transmitter buffer empty *)

BEGIN
  WITH TBEHandler^ DO
    LOOP
      Nucleus.Attach( vector );
      SIOBWRO := BYTE(01H); SIOBWRO := BYTE(11H);  (* disable interrupt *)
      SIOBWRO := BYTE(38H);                        (* reset highest IUS *)
      IF MoreDataToTransfer THEN
        SIOBWRO := BYTE(01H); SIOBWRO := BYTE(13H);(* enable interrupt  *)
        SIOBTXDATA := OutputData;                  (* transmit data     *)
      END; (* if *)
    END; (* loop *)
  END; (* with *)
END TBEinterrupt;
(*-------------------------------------------------------------------------*)
PROCEDURE ESCinterrupt;   (* external status change *)

BEGIN
  WITH ESCHandler^ DO
    LOOP
      Nucleus.Attach( vector );
      PutString(' external status change '); PutLn;
    END; (* loop *)
  END; (* with *)
END ESCinterrupt;
```

```
(*-------------------------------------------------------------------------*)
PROCEDURE RCAinterrupt;   (* received character available *)

VAR  status  : BYTE;
     value   : INTEGER;

BEGIN
  PutString(' RCA handler starting'); PutLn;
  WITH RCAHandler^ DO
    LOOP
      Nucleus.Attach( vector );
      SIOBWRO := BYTE(01H);  status  := SIOBRRO;(* read  status           *)
      PutData( Buffer, SIOBRXDATA );            (* put character in buffer *)
      SIOBWRO := BYTE(38H);                     (* reset highest IUS       *)
    END; (* loop *)
  END; (* with *)
END RCAinterrupt;
(*-------------------------------------------------------------------------*)
PROCEDURE SRCinterrupt;   (* special received character *)

VAR status  : BYTE;

BEGIN
  WITH SRCHandler^ DO
    LOOP
      Nucleus.Attach( vector );
      PutString(' *** special received character *** '); PutLn;
      SIOBWRO := BYTE(01H);  status := SIOBRRO;(* read  status           *)
      RR1 := RR1Set(SIOBRRO);      (* RR1 contains an error set, see TYPE, *)
                                   (* could be used in debugging.          *)
      Debug.PutCard(Debug.StandardInOut,CARDINAL(ORD(CHAR(status))), 6);
      PutString(' Status: ' );        PutChar(CHAR(status)); PutLn;
      PutString(' specialtecknet: '); PutChar(SIOBRXDATA);   PutLn;
      SIOBWRO := BYTE(30H);                    (* reset error bits in RR1 *)
      SIOBWRO := BYTE(38H);                    (* reset highest IUS       *)
    END; (* loop *)
  END; (* with *)
END SRCinterrupt;
(*-------------------------------------------------------------------------*)
PROCEDURE InitSCC;

BEGIN
  InitRing(Buffer);
  MoreDataToTransfer := FALSE;
  Nucleus.NewHandler( TBEHandler, TBEinterrupt, 1000, 80H );
    (* transmitter buffer empty     *)
  Nucleus.NewHandler( ESCHandler, ESCinterrupt, 1000, 82H );
    (* external status change       *)
  Nucleus.NewHandler( RCAHandler, RCAinterrupt, 1000, 84H );
    (* received character available *)
  Nucleus.NewHandler( SRCHandler, SRCinterrupt, 1000, 86H );
    (* special received character   *)
  SIOBWRO := BYTE(30H) ;
  SIOBWRO := BYTE(10H) ;
  SIOBWRO := BYTE(09H) ; SIOBWRO := BYTE(40H) ;
  SIOBWRO := BYTE(0AH) ; SIOBWRO := BYTE(00H) ;
  SIOBWRO := BYTE(0EH) ; SIOBWRO := BYTE(82H) ;
  SIOBWRO := BYTE(04H) ; SIOBWRO := BYTE(44H) ;
  SIOBWRO := BYTE(03H) ; SIOBWRO := BYTE(0C1H);
  SIOBWRO := BYTE(05H) ; SIOBWRO := BYTE(0EAH);
  SIOBWRO := BYTE(0CH) ; SIOBWRO := BYTE(02H) ;
  SIOBWRO := BYTE(0DH) ; SIOBWRO := BYTE(00H) ;
  SIOBWRO := BYTE(0BH) ; SIOBWRO := BYTE(56H) ;
  SIOBWRO := BYTE(0EH) ; SIOBWRO := BYTE(81H) ;
  SIOBWRO := BYTE(01H) ; SIOBWRO := BYTE(11H) ;
```

```
  SIOBWRO := BYTE(0FH) ; SIOBWRO := BYTE(80H) ;
  SIOBWRO := BYTE(02H) ; SIOBWRO := BYTE(80H) ;
  SIOBWRO := BYTE(09H) ; SIOBWRO := BYTE(09H) ;
END InitSCC;

END SCC.
```

```
IMPLEMENTATION MODULE SensorBall;

FROM SCC      IMPORT InitSCC, TransmitData, ReceiveData;
(* FROM Timeouts IMPORT Timeout, Init, SetTimeout, RemoveTimeout; *)
(* IMPORT Exceptions; *)
FROM Console  IMPORT GetChar, PutChar, PutString, PutLn;
FROM Conversions IMPORT CardToString;

VAR  string                : ARRAY [0..20] OF CHAR; (* for debug *)

(*-------------------------------------------------------------------------*)
PROCEDURE GetData( VAR Value: INTEGER );

VAR Data : CHAR;

BEGIN
  ReceiveData( Data );
  Value := ORD(Data);
  IF Value > 127 THEN  Value := Value - 256;  END;
END GetData;
(*-------------------------------------------------------------------------*)
PROCEDURE ReadSensorBall ( VAR PositionData : PositionDataType;
                           VAR Buttons      : INTEGER;
                           VAR Error        : CARDINAL          );

CONST firsttry = 1;
      maxtries = 5;
      maxtime  = 1000;

VAR   n1, n2, n3,
      ID, checksum, sum  : INTEGER;
      error              : CARDINAL;
(*      t                : Timeout; *)
      DataOk             : BOOLEAN;
      ch                 : CHAR;

BEGIN (* ReadSensorBall *)
    Error := 0;
(*    error := Exceptions.Catch(); (* returns 0 the first time *)
    CASE error OF
    | Exceptions.NoError : (*PutString(' *** NoError ***');PutLn*);
    | firsttry..maxtries : PutString(' *** TimedOut ***');PutLn;
    ELSE
    END; (* case *)
    IF error > maxtries THEN
      Error := 1;
      PutString(' Fatal Error. Cannot read SensorBall. '); PutLn;
      GetChar(ch);
    ELSE
      t := SetTimeout(maxtime, error + 1);*)
      REPEAT
        TransmitData( CHR(5) );       (* send enquire to the Sensor Ball *)
        GetData( n1 );                (*   number of data   *)
        GetData( n2 );                (* -      -"-        *)
        GetData( n3 );                (* nr of data XOR 55H *)
        WHILE ((n1 + n2) <> 0) OR ((BITSET(n1) / BITSET(55H)) <> BITSET(n3)) DO
          n1 := n2;
          n2 := n3;
          GetData( n3 );
        END; (* while *)
        GetData( ID );                (*              *)
        IF n1 <> 1 THEN
          WITH PositionData DO
            GetData( Xtransl );       (* X translation *)
            GetData( Ytransl );       (* Y translation *)
            GetData( Ztransl );       (* Z translation *)
            GetData( Xrot    );       (* X rotation    *)
            GetData( Yrot    );       (* Y rotation    *)
            GetData( Zrot    );       (* Z rotation    *)
            sum := Xtransl + Ytransl + Ztransl + Xrot + Yrot + Zrot;
          END; (* with *)
          GetData( Buttons );         (*   8 buttons   *)
          sum := sum + ID + Buttons + n1;
          IF Buttons < 0 THEN  INC(Buttons, 256);  END; (* there are reasons *)
          GetData( checksum );
          DataOk := (checksum + sum) MOD 256 = 0;   (*   Checksum    *)
        ELSE (* sensor not ready *)
          PutString(' Sensor not ready '); PutLn;
          GetData( checksum );
          (*DataOk := (n1 + ID + checksum) MOD 256 = 0;*)(* for debug *)
          DataOk := FALSE;
        END; (* if *)
      UNTIL DataOk;
(*      RemoveTimeout(t);
    END; (* if *)*)
END ReadSensorBall;

(*-------------------------------------------------------------------------*)

PROCEDURE InitSensorBall;

BEGIN
(*  Init; (* Timeouts *) *)
  InitSCC;
END InitSensorBall;


END SensorBall.
```

```
IMPLEMENTATION MODULE VDAD;


FROM SYSTEM     IMPORT BYTE, SHORTWORD, ADR, CAST;
FROM Semaphores IMPORT InitSem;
FROM Console    IMPORT PutString, PutLn;
FROM Debug      IMPORT PutCard;

(*-------------------------------------------------------------------------*)
PROCEDURE InitVDAD;
(* Initializes the VDAD cards *)

VAR CardNr : CARDINAL;

BEGIN
  FOR CardNr := 1 TO NrOfCards DO
    WITH VDAD[ CardNr ] DO
      PGCR  := BYTE(11H);
      PSRR  := BYTE(18H);
      PBDDR := ByteSet(0FFH);    (* port B: all wires output *)
      PIVR  := BYTE(0F0H);
      PBCR  := BYTE(80H);
      PSR   := BYTE(00H);
                                 (* Timer *)
      TCR   := BYTE(40H);
      TIVR  := BYTE(0FH);
      CPRH  := BYTE(00H);
      CPRM  := BYTE(00H);
      CPRL  := BYTE(00H);
      TSR   := BYTE(00H);
                                 (* AnalogIO  *)
      PACR  := BYTE(80H);
(*      PACR  := BYTE(82H); *)
      PADDR := ByteSet(7FH);
      PADR  := ByteSet(00H);
      DACMR := TwoByteSet(0FF0H);
    END; (* with *)
  END; (* for *)
  (*InitSem(Mutex,1);    (* *)*)

END InitVDAD;
(*-------------------------------------------------------------------------*)
(*BEGIN
  PutString(' intoVDAD'); PutLn;
  PutCard( CARDINAL(ADR(VDAD[2].ADCRCA)), -8 ); PutLn;
  InitVDAD;*)
END VDAD.
```

```
IMPLEMENTATION MODULE AnalogIO;

FROM VDAD       IMPORT VDAD, TwoByteSet, ByteSet, Mutex;
FROM SYSTEM     IMPORT WORD, SHORTWORD, BYTE, CAST;
FROM Semaphores IMPORT Wait, Signal;

CONST EOC = 7; (* End Of Conversion *)
TYPE BITS = SET OF [0..15];

(*-------------------------------------------------------------------------*)
VAR x, y : REAL;

PROCEDURE ADin  ( cardnr : CardType; channel : ChannelType ; VAR value : INTEGER );

VAR k : CARDINAL;
BEGIN
  (*Wait(Mutex);*)
  VDAD[cardnr].ADCRCA := channel;
(*  x := y;
  FOR k := 1 TO 100 DO x := x + y; END; *)
  REPEAT  UNTIL EOC IN VDAD[cardnr].PADR;
  value := SHORTINT( BITS(VDAD[cardnr].ADCRCA) * BITS(0FFFH) ) - 2047;
  (*Signal(Mutex);*)
END ADin;
(*-------------------------------------------------------------------------*)
PROCEDURE DAout ( cardnr : CardType; channel : ChannelType; value : DARange );

BEGIN
  VDAD[cardnr].DACOR[channel] := value + 2048;
END DAout;
(*-------------------------------------------------------------------------*)
PROCEDURE MultiADin ( cardnr : CardType; LowChannel, HighChannel : ChannelType ;
                      VAR value : IntArray );

VAR channel : ChannelType;
    bla : SHORTWORD;
BEGIN
  (*Wait(Mutex);*)
  INCL( VDAD[cardnr].PADR, 4);  (* MCS on *)
  channel := HighChannel;
  VDAD[cardnr].ADCRCA := HighChannel;
  REPEAT  UNTIL EOC IN VDAD[cardnr].PADR;
  REPEAT
    value[channel] := SHORTINT( BITS(VDAD[cardnr].ADCRCA) * BITS(0FFFH) ) - 2047;
    REPEAT  UNTIL EOC IN VDAD[cardnr].PADR;
    DEC(channel);
  UNTIL channel = LowChannel;
  EXCL( VDAD[cardnr].PADR, 4);  (* MCS off *)
  value[channel] := SHORTINT( BITS(VDAD[cardnr].ADCRCA) * BITS(0FFFH) ) - 2047;
  (*Signal(Mutex);*)
END MultiADin;
(*-------------------------------------------------------------------------*)
PROCEDURE SetInputGain( cardnr : CardType; ExpGain : ExpGainType);
(* ExpGain = [0, 2] => Input gain = 1, 10, 100 *)

BEGIN
  VDAD[cardnr].PADR := VDAD[cardnr].PADR * ByteSet{7,6,5,4,3,2}
                     + ByteSet(BYTE(CHR(ExpGain)));
END SetInputGain;
(*-------------------------------------------------------------------------*)
PROCEDURE SetOutVoltage( cardnr : CardType; channel : ChannelType;
                         Gain, UniBiPolar : OutRangeType );
(* Outvoltage range = Vref * (1 + Gain)  *)
(* UniBiPolar: 0 = unipolar, 1 = bipolar *)

VAR DACMRcopy : TwoByteSet;

BEGIN
  DACMRcopy := VDAD[cardnr].DACMR;
  IF      Gain = 0         THEN   EXCL( DACMRcopy,  3 - channel );
  ELSE (* Gain = 1 *)             INCL( DACMRcopy,  3 - channel );
  END;
  IF      UniBiPolar = 0   THEN   EXCL( DACMRcopy, 15 - channel );
  ELSE (* UniBiPolar = 1 *)       INCL( DACMRcopy, 15 - channel );
  END;
  VDAD[cardnr].DACMR := DACMRcopy;
END SetOutVoltage;
(*-------------------------------------------------------------------------*)
BEGIN
  y := 1.0;
END AnalogIO.
```

```
IMPLEMENTATION MODULE DigitalIO;

FROM SYSTEM IMPORT ADR, CAST, BYTE, WORD, SHORTWORD;
FROM VDAD   IMPORT VDAD, AVDIN, ByteSet;
FROM Debug  IMPORT PutCard, PutLn;

(*-------------------------------------------------------------------------*)
PROCEDURE DigWireOutput( cardnr : CardType; wire : WireType; value : Bit );

VAR PBDRSet : ByteSet;

BEGIN
  INCL( VDAD[cardnr].PBDDR, wire  );
  PBDRSet := CAST(ByteSet, VDAD[cardnr].PBDR);
  IF value <> 0 THEN INCL(PBDRSet, wire );
  ELSE               EXCL(PBDRSet, wire );
  END; (* if *)
  VDAD[cardnr].PBDR := BYTE(PBDRSet);
END DigWireOutput;
(*-------------------------------------------------------------------------*)
PROCEDURE DigWireInput ( cardnr : CardType; wire : WireType; VAR value : CARDINAL);

BEGIN
  EXCL( VDAD[cardnr].PBDDR, wire );
  value := ORD(CHAR( ByteSet(VDAD[cardnr].PBDR) * ByteSet{wire} ));
  IF value <> 0 THEN  value := 1;  END;
END DigWireInput;
(*-------------------------------------------------------------------------*)
PROCEDURE DigByteOutput( cardnr : CardType; value : BYTE );

BEGIN
  VDAD[cardnr].PBDDR := ByteSet{0..7};
  VDAD[cardnr].PBDR  := value;
END DigByteOutput;
(*-------------------------------------------------------------------------*)
PROCEDURE DigByteInput ( cardnr : CardType; VAR value : BYTE );

BEGIN
  VDAD[cardnr].PBDDR := ByteSet{};
  value              := VDAD[cardnr].PBDR;
END DigByteInput;
(*-------------------------------------------------------------------------*)
PROCEDURE DigInput( VAR value : SHORTINT );
(* value = [-32768, 32767] *)

BEGIN
   value := 8000H - SHORTINT(AVDIN);
END DigInput;
(*-------------------------------------------------------------------------*)
END DigitalIO.
```

```
IMPLEMENTATION  MODULE ResolverIO;

FROM SYSTEM      IMPORT ADDRESS, WORD, BYTE;
FROM VDAD        IMPORT InitVDAD, VDAD, ByteSet, VDADtype, AVDIN, Mutex;
FROM DigitalIO   IMPORT DigWireInput;
IMPORT AnalogIO;
FROM AnalogIO    IMPORT ADin, MultiADin, DAout;
FROM Console     IMPORT CharAvailable, GetChar, GetString, PutChar, PutString, PutLn ;
FROM Conversions IMPORT CardToString, RealToString, IntToString;
FROM Semaphores  IMPORT Semaphore, InitSem, Wait, Signal;

VAR
    ready    : CARDINAL;
    string   : ARRAY [0..20] OF CHAR;
    ch       : CHAR;
    steps    : INTEGER;

(* robotvalue = [-32768,32767] + addfact  *)
(* motorvalue = [-32768,32767]            *)
(* addfact    = +/- motorturns * 2^16     *)
(*-------------------------------------------------------------------------*)
PROCEDURE InitResolver;

BEGIN
   InitVDAD;
   VDAD[1].PBDDR     := ByteSet{0..7};   (* Set all eight bits to OUT *)
   DigPort1.adr      := 0; (* Axis 7 not present: no reset noise on joints *)
   DigPort1.reset    := Inactive;
   (*DigPort1.setbusy := 1;*)
END InitResolver;
(*-------------------------------------------------------------------------*)
PROCEDURE InitDigPosition(     resolvernr  : ResolverType;
                           VAR motorvalue,
                               addfact     : IntArray );

VAR knl,a              : INTEGER;
    val                : INTEGER;
    k                  : CARDINAL;
BEGIN
    steps := 10;
    DigPort1.adr   := 7 - resolvernr;
    DigPort1.reset := Active;
    REPEAT steps := steps-1; UNTIL steps = 0; (* delay to get proper pulse *)
    DigPort1.reset := Inactive;
    DigPort1.adr   := 7 - resolvernr;
    DigPort1.read  := Active;
    DigPort1.setbusy := Active;
    REPEAT  DigWireInput(2, 2, ready)  UNTIL ready = 1;
    val := AVDIN - 7FFFH;                          (* val = [-32768,32767] *)
    DigPort1.read := Inactive;
    DigPort1.setbusy := Inactive;
    DigPort1.adr      := 0; (* Axis 7 not present: no reset noise on joints *)
    motorvalue[resolvernr] := val;
    addfact[resolvernr]    := 0;
END InitDigPosition;
(*-------------------------------------------------------------------------*)
PROCEDURE InitAnPosition(     resolvernr  : ResolverType;
                          VAR motorvalue,
                              addfact     : IntArray );

BEGIN
    steps := 10;
    DigPort1.adr   := 7 - resolvernr;
    DigPort1.reset := Active;
    REPEAT steps := steps-1; UNTIL steps = 0; (* delay to get proper pulse *)
    DigPort1.reset := Inactive;
    DigPort1.adr   := 0; (* Axis 7 not present: no reset noise on joints *)
    ADin(2, resolvernr, motorvalue[resolvernr]);
    motorvalue[resolvernr] := motorvalue[resolvernr]*16;(*from 12 to 16 bits*)
    addfact[resolvernr]:= 0;
END InitAnPosition;
(*-------------------------------------------------------------------------*)
PROCEDURE DigPosition (     resolvernr  : ResolverType;
                        VAR robotvalue  : INTEGER;
                        VAR motorvalue,
                            addfact     : IntArray );

VAR val    : INTEGER;
    k, a   : INTEGER;

BEGIN
    DigPort1.read    := Active;
    DigPort1.setbusy := Active;
    DigPort1.adr  := 7 - resolvernr;
    REPEAT  DigWireInput(2, 2, ready)  UNTIL ready = 1;
    val := AVDIN - 7FFFH;                          (* val = [-32768,32767] *)
    DigPort1.read     := Inactive;
    DigPort1.setbusy := Inactive;
    DigPort1.adr      := 0; (* Axis 7 not present: no reset noise on joints *)
    IF    (motorvalue[resolvernr]-val) >  40000 THEN
      INC(addfact[resolvernr], 65536);
    ELSIF (motorvalue[resolvernr]-val) < -40000 THEN
      DEC(addfact[resolvernr], 65536);
    END; (* if *)
    robotvalue             := val + addfact[resolvernr];
    motorvalue[resolvernr] := val;
END DigPosition;
(*-------------------------------------------------------------------------*)
PROCEDURE  AnPosition(     resolvernr  : ResolverType;
                       VAR robotvalue  : INTEGER;
                       VAR motorvalue,
                           addfact     : IntArray );

VAR val : INTEGER;

BEGIN
   ADin(2, resolvernr, val);       (* val = [-2048,2047] *)
   val := val * 16;                (* from 12 to 16 bits *)
   IF    (motorvalue[resolvernr] - val) >  40000 THEN
         INC(addfact[resolvernr], 65536);
   ELSIF (motorvalue[resolvernr] - val) < -40000 THEN
         DEC(addfact[resolvernr], 65536);
   END; (* if *)
   robotvalue             := val + addfact[resolvernr];
   motorvalue[resolvernr] := val;
END AnPosition;
(*-------------------------------------------------------------------------*)
PROCEDURE Digital5Pos  ( VAR robotvalue,
                             motorvalue,
                             addfact     : IntArray );
VAR val    : INTEGER;
    values : IntArray;
    i      : CARDINAL;

BEGIN
    DigPort1.read    := Active;
    FOR i := 1 TO 5 DO
      DigPort1.setbusy := Active;
      DigPort1.adr     := 7 - i;
      DigPort1.setbusy := Inactive;      (* pulse -> read next resolver *)
```

```
      REPEAT  DigWireInput(2, 2, ready)  UNTIL ready = 1;
      val := AVDIN - 7FFFH;    (* val = [-32768,32767] *)
      IF    (motorvalue[i]-val) >  40000 THEN
        INC(addfact[i], 65536);
      ELSIF (motorvalue[i]-val) < -40000 THEN
        DEC(addfact[i], 65536);
      END; (* if *)
      robotvalue[i] := val + addfact[i];
      motorvalue[i] := val;
    END; (* for *)
    DigPort1.read := Inactive;
    DigPort1.adr      := 0; (* Axis 7 not present: no reset noise on joints *)
END Digital5Pos;
(*-------------------------------------------------------------------------*)
PROCEDURE Analog5Pos  ( VAR robotvalue,
                            motorvalue,
                            addfact     : IntArray );
VAR values : AnalogIO.IntArray;
    i      : CARDINAL;

BEGIN
  MultiADin(2, 1, 5, values);
  FOR i := 1 TO 5 DO
    values[i] := values[i] * 16;                (* from 12 to 16 bits *)
    IF    (motorvalue[i] - values[i]) >  40000 THEN
          INC(addfact[i], 65536);
    ELSIF (motorvalue[i] - values[i]) < -40000 THEN
          DEC(addfact[i], 65536);
    END; (* if *)
    robotvalue[i] := values[i] + addfact[i];
    motorvalue[i] := values[i];
  END; (* for *)
END Analog5Pos;
(*-------------------------------------------------------------------------*)
(*PROCEDURE Speed(resolvernr:ResolverType; VAR value:REAL);
(* Reads the decimal value wich is proportional to the *)
(*  axle speed of the servo on resolvercard resolvernr *)
VAR val          : INTEGER;

BEGIN
   ADin(1,15,val);
   value:=(val-2047.0)*4.884E-3;
END Speed;*)

END ResolverIO.
```

```
IMPLEMENTATION MODULE ServoIO;

FROM ResolverIO  IMPORT InitResolver, InitDigPosition, DigPosition, DigPort1,
                        BitState, InitAnPosition, AnPosition, Analog5Pos,
                        Digital5Pos,FiveIntArray, IntArray;
FROM AnalogIO    IMPORT DAout;
FROM DigitalIO   IMPORT DigWireInput, DigByteInput, DigByteOutput;
FROM Console     IMPORT PutString, PutLn;
FROM Conversions IMPORT RealToString, IntToString, CardToString;

CONST pi = 3.14159;

VAR robotvalue,       (* proportional to the robot angle       *)
    motorvalue,       (* proportional to the robot motor angle *)
    TurnFact   : IntArray;
    DAoutScale : ARRAY AnalogOutputs OF LONGREAL;
    string     : ARRAY [0..20] OF CHAR;

(*-------------------------------------------------------------*)
PROCEDURE absPosDigital ( resolvernr : JointType ) : LONGREAL;

CONST AngleScale = 64.0 * pi / 65535.0;
VAR Angle : LONGREAL;

BEGIN
  DigPosition(resolvernr, robotvalue[resolvernr], motorvalue, TurnFact );
  Angle := FLOAT(robotvalue[resolvernr]) * AngleScale;
  RETURN Angle;
END absPosDigital;
(*-------------------------------------------------------------*)
PROCEDURE absPosAnalog ( resolvernr : JointType ) : LONGREAL;

CONST AngleScale = 64.0 * pi / 65535.0;
VAR Angle : LONGREAL;

BEGIN
  AnPosition(resolvernr, robotvalue[resolvernr], motorvalue, TurnFact );
  Angle := FLOAT(robotvalue[resolvernr]) * AngleScale;
  RETURN Angle;
END absPosAnalog;
(*-------------------------------------------------------------*)
PROCEDURE incPosDigital ( resolvernr : JointType ) : REAL;

CONST AngleScale = 64.0 * pi / 65535.0;
VAR Angle     : REAL;
    Increment,
    lastrobotvalue : INTEGER;

BEGIN
  lastrobotvalue := robotvalue[resolvernr];
  DigPosition(resolvernr, robotvalue[resolvernr], motorvalue, TurnFact );
  Increment := robotvalue[resolvernr] - lastrobotvalue;
  Angle := FLOAT(Increment) * AngleScale;
  RETURN Angle;
END incPosDigital;
(*-------------------------------------------------------------*)
PROCEDURE incPosAnalog ( resolvernr : JointType ) : REAL;

CONST AngleScale = 64.0 * pi / 65535.0;
VAR Angle     : REAL;
    Increment,
    lastrobotvalue : INTEGER;
BEGIN
  lastrobotvalue := robotvalue[resolvernr];
  AnPosition(resolvernr, robotvalue[resolvernr], motorvalue, TurnFact );
  Increment := robotvalue[resolvernr] - lastrobotvalue;
  Angle := FLOAT(Increment) * AngleScale;
  RETURN Angle;
END incPosAnalog;
(*-------------------------------------------------------------*)
PROCEDURE abs5PosDigital (): Array5Longreal;

CONST AngleScale = 64.0 * pi / 65535.0;
VAR Angle : Array5Longreal;

BEGIN
  Digital5Pos(robotvalue, motorvalue, TurnFact);
  Angle[1] := FLOAT(robotvalue[1]) * AngleScale;
  Angle[2] := FLOAT(robotvalue[2]) * AngleScale;
  Angle[3] := FLOAT(robotvalue[3]) * AngleScale;
  Angle[4] := FLOAT(robotvalue[4]) * AngleScale;
  Angle[5] := FLOAT(robotvalue[5]) * AngleScale;
  RETURN Angle;
END abs5PosDigital;
(*-------------------------------------------------------------*)
PROCEDURE abs5PosAnalog (): Array5Longreal;

CONST AngleScale = 64.0 * pi / 65535.0;
VAR Angle : Array5Longreal;

BEGIN
  Analog5Pos(robotvalue, motorvalue, TurnFact);
  Angle[1] := FLOAT(robotvalue[1]) * AngleScale;
  Angle[2] := FLOAT(robotvalue[2]) * AngleScale;
  Angle[3] := FLOAT(robotvalue[3]) * AngleScale;
  Angle[4] := FLOAT(robotvalue[4]) * AngleScale;
  Angle[5] := FLOAT(robotvalue[5]) * AngleScale;
  RETURN Angle;
END abs5PosAnalog;
(*-------------------------------------------------------------*)
PROCEDURE inc5PosDigital () : Array5Real;

CONST AngleScale = 64.0 * pi / 65535.0;
VAR Angle : Array5Real;
    Increment : INTEGER;
    lastrobotvalue : FiveIntArray;

BEGIN
  lastrobotvalue[1] := robotvalue[1];
  lastrobotvalue[2] := robotvalue[2];
  lastrobotvalue[3] := robotvalue[3];
  lastrobotvalue[4] := robotvalue[4];
  lastrobotvalue[5] := robotvalue[5];
  Digital5Pos(robotvalue, motorvalue, TurnFact);
  Increment := robotvalue[1] - lastrobotvalue[1];
  Angle[1] := FLOAT(Increment) * AngleScale;
  Increment := robotvalue[2] - lastrobotvalue[2];
  Angle[2] := FLOAT(Increment) * AngleScale;
  Increment := robotvalue[3] - lastrobotvalue[3];
  Angle[3] := FLOAT(Increment) * AngleScale;
  Increment := robotvalue[4] - lastrobotvalue[4];
  Angle[4] := FLOAT(Increment) * AngleScale;
  Increment := robotvalue[5] - lastrobotvalue[5];
  Angle[5] := FLOAT(Increment) * AngleScale;
  RETURN Angle;
END inc5PosDigital;
(*-------------------------------------------------------------*)
PROCEDURE inc5PosAnalog () : Array5Real;

CONST AngleScale = 64.0 * pi / 65535.0;
```

```
VAR Angle : Array5Real;
    Increment : INTEGER;
    lastrobotvalue : FiveIntArray;

BEGIN
  lastrobotvalue[1] := robotvalue[1];
  lastrobotvalue[2] := robotvalue[2];
  lastrobotvalue[3] := robotvalue[3];
  lastrobotvalue[4] := robotvalue[4];
  lastrobotvalue[5] := robotvalue[5];
  Analog5Pos(robotvalue, motorvalue, TurnFact);
  Increment := robotvalue[1] - lastrobotvalue[1];
  Angle[1] := FLOAT(Increment) * AngleScale;
  Increment := robotvalue[2] - lastrobotvalue[2];
  Angle[2] := FLOAT(Increment) * AngleScale;
  Increment := robotvalue[3] - lastrobotvalue[3];
  Angle[3] := FLOAT(Increment) * AngleScale;
  Increment := robotvalue[4] - lastrobotvalue[4];
  Angle[4] := FLOAT(Increment) * AngleScale;
  Increment := robotvalue[5] - lastrobotvalue[5];
  Angle[5] := FLOAT(Increment) * AngleScale;
  RETURN Angle;
END  inc5PosAnalog;
(*-------------------------------------------------------------*)
PROCEDURE InitDigPos ( resolvernr : JointType );

BEGIN
  InitDigPosition( resolvernr, motorvalue, TurnFact );
  robotvalue[resolvernr] := motorvalue[resolvernr];
END InitDigPos;
(*-------------------------------------------------------------*)
PROCEDURE InitAnPos ( resolvernr : JointType );

BEGIN
  InitAnPosition( resolvernr, motorvalue, TurnFact );
  robotvalue[resolvernr] := motorvalue[resolvernr];
END InitAnPos;
(*-------------------------------------------------------------*)
PROCEDURE Init( AnOrDig : ReadMode );

VAR i : CARDINAL;

BEGIN
  InitResolver;
  CASE AnOrDig OF
    |analog  : absPos     := absPosAnalog;
               incPos     := incPosAnalog;
               abs5Pos    := abs5PosAnalog;
               inc5Pos    := inc5PosAnalog;
               ResetServo := InitAnPos;
    |digital : absPos     := absPosDigital;
               incPos     := incPosDigital;
               abs5Pos    := abs5PosDigital;
               inc5Pos    := inc5PosDigital;
               ResetServo := InitDigPos;
  END; (* case *)
  FOR i :=  MIN(JointType) TO MAX(JointType) DO
    DAoutScale[i] :=  4095.0 / (2.0*MaxVel);
  END;
END Init;
(*-------------------------------------------------------------------*)
PROCEDURE RefScale( joint : JointType; MaxRefOut : LONGREAL );
BEGIN
  DAoutScale[joint] :=  4095.0 / (2.0*MaxRefOut);
END RefScale;
```

```
(*-------------------------------------------------------------------*)
PROCEDURE RefOut ( joint : AnalogOutputs; value : LONGREAL );

VAR card, channel,
    RefVel           : INTEGER;

BEGIN (* RefVelOut *)
   RefVel := TRUNC( value * DAoutScale[joint] );
   IF    RefVel >  2047 THEN RefVel :=  2047;
   ELSIF RefVel < -2048 THEN RefVel := -2048;
   END; (* if *)
   CASE joint OF
     | 0..3  :  card    := 1;
                channel := joint;
     | 4..7  :  card    := 2;
                channel := joint-4;
   END; (* case *)
   DAout(card, channel, RefVel);
END RefOut;
(*-------------------------------------------------------------------*)
PROCEDURE ZeroSense ( joint : JointType ) : CARDINAL;
  (* ZeroSense  = [0,1]  *)
VAR
  zerosense, ready : CARDINAL;
BEGIN
  DigPort1.adr := 7 - joint;
  DigPort1.read  := Active;
  DigPort1.setbusy := Active;
  REPEAT  DigWireInput(2, 2, ready)  UNTIL ready = 1;
  DigWireInput(2, 1, zerosense);
  DigPort1.read := Inactive;
  DigPort1.setbusy := Inactive;
  DigPort1.adr      := 0; (* Axis 7 not present: no reset noise on joints *)
  RETURN zerosense;
END ZeroSense;
(*-------------------------------------------------------------------*)
END ServoIO.
```

```
IMPLEMENTATION MODULE RingBuffer;

FROM SYSTEM      IMPORT BYTE, ADR ;
FROM Semaphores  IMPORT Semaphore, InitSem, Wait, Signal;
FROM Console     IMPORT PutString, PutChar, PutLn;
FROM Conversions IMPORT CardToString;

TYPE
  RingHead = RECORD
               NotEmpty    : Semaphore;
               ReadIndex,
               WriteIndex,
               BufferSize  : SHORTCARD;
             END;
  HeadPointer = POINTER TO RingHead;
  BigArray    = ARRAY [0.. 100000] OF BYTE;
  SlotPointer = POINTER TO BigArray;

VAR
  string      : ARRAY [0..10] OF CHAR;

(*-------------------------------------------------------------*)

PROCEDURE InitRing  ( VAR r : ARRAY OF BYTE );

VAR
  head  : HeadPointer;
  slot  : SlotPointer;

BEGIN (* InitRing *)
  head := ADR(r[0]);
  head^.BufferSize  := (HIGH(r) + 1 - SIZE(RingHead));
  head^.ReadIndex   := 0;
  head^.WriteIndex  := 0;
  InitSem(head^.NotEmpty, 0, "RingBuffer.NotEmpty");
END InitRing;

(*-------------------------------------------------------------*)

PROCEDURE MoreData ( VAR r : ARRAY OF BYTE ) : BOOLEAN;

VAR
  head  : HeadPointer;

BEGIN (* MoreData *)
  head := ADR(r[0]);
  RETURN NOT( head^.ReadIndex = head^.WriteIndex );
END MoreData;

(*-------------------------------------------------------------*)

PROCEDURE MoreSpace ( VAR r    : ARRAY OF BYTE;
                      Variable : ARRAY OF BYTE ) : BOOLEAN;

VAR
  head  : HeadPointer;
  i,
  index : SHORTCARD;

BEGIN (* MoreSpace *)
  head := ADR(r[0]);
  index := head^.WriteIndex;
  i := 0;
  WHILE (i <= HIGH(Variable)) AND (index + 1 <> head^.ReadIndex) DO
    INC(index);
    IF index >= head^.BufferSize THEN index := 0; END;
    INC(i);
  END; (* while *)

  RETURN i = HIGH(Variable) + 1;
END MoreSpace;

(*-------------------------------------------------------------*)

PROCEDURE PutData (VAR r : ARRAY OF BYTE; Value : ARRAY OF BYTE );

VAR
  head  : HeadPointer;
  slot  : SlotPointer;
  index,
  i     : SHORTCARD;

BEGIN (* PutData *)
  head := ADR(r[0]);
  index := head^.WriteIndex;
  slot := ADR(r[SIZE(RingHead)]);
  i := 0;
  WHILE (i <= HIGH(Value)) AND (head^.WriteIndex + 1 <> head^.ReadIndex) DO
    slot^[head^.WriteIndex] := Value[i];
    INC(head^.WriteIndex);
    IF head^.WriteIndex >= head^.BufferSize THEN
      head^.WriteIndex := 0;
    END; (* if *)
    INC(i);
  END; (* while *)
  IF i = HIGH(Value) + 1 THEN
    Signal(head^.NotEmpty );
  ELSE
    head^.WriteIndex := index;
    PutString('OOPS 1'); PutLn;
  END (* if  *);
  (*CardToString( string, head^.WriteIndex, 5 );
  PutString(' WriteIndex: '); PutString( string ); PutLn;*)
END PutData;

(*-------------------------------------------------------------*)

PROCEDURE GetData ( VAR r : ARRAY OF BYTE; VAR Value : ARRAY OF BYTE );

VAR
  head   : HeadPointer;
  slot   : SlotPointer;
  index,
  i      : SHORTCARD;

BEGIN (* GetData *)
  head := ADR(r[0]);
  index := head^.ReadIndex;
  slot := ADR(r[SIZE(RingHead)]);
  Wait(head^.NotEmpty );
  IF index <> head^.WriteIndex THEN
    FOR i := 0 TO HIGH(Value) DO
      Value[i] := slot^[index];
      index := index + 1;
      IF index >= head^.BufferSize THEN index := 0; END; (* if *)
    END; (* for *)
    head^.ReadIndex := index;
  ELSE
    PutString('OOPS 2'); PutLn;
  END; (* if *)
```

```
  (*CardToString( string, head^.ReadIndex, 5 );
  PutString(' ReadIndex: '); PutString( string ); PutLn;*)
END GetData;

(*---------------------------------------------------------------*)

END RingBuffer.
```

```
function w = bofreq

% function w = bofreq
%
% With bofreq you can pick out interesting
% frequencies in a bodeplot.

[n,m] = size(glob_scale);
if n == 2,
  subplot(211);
  axis(glob_scale(1,:));
  loglog(10^glob_scale(1,1), 10^glob_scale(1,3), 'i');
else
  hold on
end

for i = 1:100,
  if n == 2,  subplot(211); end
  [x,y,b] = ginput(1);
  if     b == 1
    w = [w;x];
    loglog(x,y,'+');
  elseif b == 2
    disp(' button not implemented')
  elseif b == 3
    break
  end
end

if n == 2,
  subplot(212);
  axis(glob_scale(1,:));
  axis;
else
  hold off
end

if glob_hz
  w = w*2*pi;
end
```

```
function [a, b] = fr2tf( Giw )

% function [a, b] = fr2tf( Giw )
%
% This function makes a bodeplot and allows you to choose
% interesting frequences. From these frequences the
% A and B polynoms are calculated. A new bodeplot, from
% these polynoms, is done in the old bodeplot.
% Finally a pole-zero plot is done and the coefficients,
% poles and zeros written.

bopl(Giw);

disp('')
disp(' left button  : choose points ')
disp(' right button : quit ')

w = bofreq;

fr = fpick(Giw, w);

nb  = input(' deg B : ');
na  = input(' deg A : ');
tau = input(' tau   : ');

[b,a,e] = lsbac( fr, nb, na );

bosh( frc( b, a, tau, 1, 3, 200 ) )

input(' press return for pole-zero plot ');

% scales are specific for the robot projekt
clg
subplot(211)
pzpl( b, a, 2, 4, 0.01, [-250 250 -250 250] )
pzgrid
title(' POLE - ZERO PLOT ')
subplot(212)
pzpl( b, a, 2, 4, 0.01, [-10 5 -300 300] )
pzgrid
title(' POLE - ZERO PLOT (Re and Im in different scale) ')

disp(''), disp('')
disp(' --- coefficient in B ---')
b'
disp(' --- coefficient in A ---')
a'
disp('    --- zeros ---')
rb = roots(b)
disp('    --- poles ---')
ra = roots(a)
```

```
function Cuy = coher( uy, M, G, T)

% function Cuy = coher( uy, M, G, T)
% Computes the coherence function.
% The coherence and the amplitude are shown.
% uy  = input output matrix
% M   = window size (see SPA)
% G   = transfer function generated by SPA
% T   = sampling interval
% Cuy = [frequences coherence]

n = length(G);
w = G(2:n,1)';

phiu = spa( uy(:,1), M, w, -1, T) * [0;1;0];
phiy = spa( uy(:,2), M, w, -1, T) * [0;1;0];

absphiyu = G(2:n,2).* phiu(2:n);
Cuy(:,1) = G(2:n,1);
Cuy(:,2) = absphiyu./sqrt(phiu(2:n))./sqrt(phiy(2:n));

clg
subplot(211)
semilogx(w, Cuy(:,2))
title('Coherence')
subplot(212)
semilogx(w, G(2:n,2))
title('Amplitude')
xlabel('frequence')
input('<Return>');
clg
G(2:n,2) = G(2:n,2)/max(G(2:n,2));
semilogx(w, G(2:n,2),'--')
hold
semilogx(w, Cuy(:,2))
hold
title('Coherence and amplitude (solid/dashed)')
xlabel('frequence')
```