LUND UNIVERSITY

PC Program i ADA

Hagberg, Ulf

1984

Document Version:
Förlagets slutgiltiga version

Link to publication

Citation for published version (APA):
Hagberg, U. (1984). *PC Program i ADA*. (Technical Reports TFRT-7276). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

CODEN:LUTFD2/(TFRT-7276)/1-23/(1984)

PC PROGRAM I ADA

ULF HAGBERG

DEPARTMENT OF AUTOMATIC CONTROL
LUND INSTITUTE OF TECHNOLOGY

OCTOBER 1984

LUND INSTITUTE OF TECHNOLOGY
DEPARTMENT OF AUTOMATIC CONTROL
Box 725
S 220 07 Lund 7 Sweden

Document name
Report

Date of issue
841001

Document number
CODEN:LUTFD2/(TFRT-7276)01-023/(1984)

Supervisor

Sponsoring organization

Author(s)
Ulf Hagberg

Title and subtitle
PC program i Ada.
(PC programs in Ada.)

Abstract

This paper discuss if it is possible to use Ada in process logic control systems. The first part discribes some today common systems on the market. A test example is defined to make it possible to compare different systems. From this point some attempts are made to solve the example in Ada. The second part contains programpackages and a systemsolution written in Ada.

Key words
Process Control

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
Swedish

Number of pages
23

Recipient's notes

Security classification

Distribution: The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

DOK. MENTDATABL .D RT 3/81

LUND INSTITUTE OF TECHNOLOGY
DEPARTMENT OF AUTOMATIC CONTROL
Box 725
S 220 07 Lund 7 Sweden

Document name
Report

Date of issue
84 09 28 → 841001

Document number
7276 ~~7000~~

Supervisor

Sponsoring organization

Author(s)
Ulf Hagberg

Title and subtitle
PC Program i Ada
(PC programs in Ada)

Abstract

This paper discuss if it is possible to use Ada in process logic control systems. The first part describes some today common systems on the market. A test example is defined to make it possible to compare different systems. From this point some attempts are made to solve the example in Ada. The second part contains programpackages and a systemsolution written in Ada.

Key words
Process Control

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
Swedish

Number of pages
22 3

Security classification

Recipient's notes

Distribution: The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

DOKUMENTDATABLAD RT 3/81

PC PROGRAM I ADA

Ulf Hagberg

Innehåll:

1. INLEDNING

2. DAGENS PROGRAMMERBARA STYRSYSTEM

3. KRAV PÅ ETT STYRSYSTEM

4. PC PROGRAM KONTRA VANLIGA PROGRAM

5. HUR KAN ADA ANVÄNDAS

6. SLUTSATSER

7. REFERENSER

1. INLEDNING

Denna rapport behandlar hur Ada (1) kan användas i styrsystem (PC). Som en introduktion beskrives först hur några i dag vanliga PC system fungerar och hur de programmeras. I avsnitt 3 diskuteras vilka krav som bör ställas på ett styrsystem. Det finns vissa egenskaper hos dagens styrsystem som gör att programstrukturen där skiljer sig från vanliga programspråk. Detta diskuteras i avsnitt 4. Ada kan redan i dagens styrsystem ersätta de språk inklusive operativsystem som utgör omgivning till PC språket. Frågan är då om Ada även kan ersätta själva PC språket? Detta behandlas i avsnitt 5.

2. DAGENS PROGRAMMERBARA STYRSYSTEM

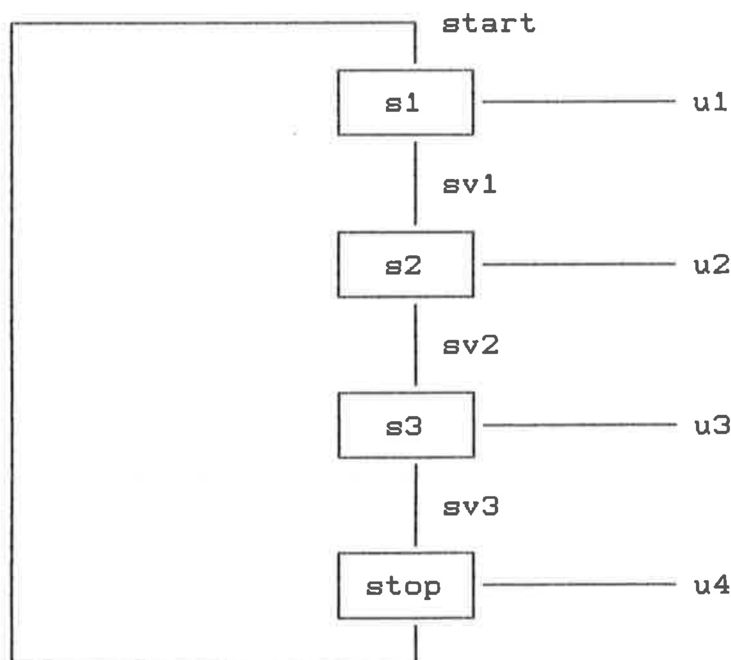
Dagens programmerbara styrsystem består i de allra flesta fallen av en enda stor programloop. Loopen består av inläsning av insignaler från den styrda processen, PC pogram exekvering samt skrivning av utsignaler till processen. Ofta genomlöpes hela pc programmet i varje loop. Detta medför att pc programmens beteende ej styrs av programflödet. Istället är det uppknutet till variabler och dess värden. Sättet att programmera systemen varierar dock. Traditionellt finns det två skolor dels den amerikanska som förespråkar grafiska relä teknik dels den europeiska som förespråkar booleska uttryck. Båda dessa metoder kan beskrivas som lågnivåspråk. Det börjar dock dyka upp produkter med något högre språknivå. Exempel på sådana system är Asea Master med sina funktionsblock, Alrite språket från Alfa Laval (2) och Grafcet från den franska samarbetsorganisationen ADEPA (3). På de följande sidorna finns beskrivet ett testexempel. Detta testexempel har sedan programmerats i dels SATTs PC språk (4) dels i Alrite från Alfa Laval.

Testexempel

Får att lättare kunna jämföra olika program exempel har följande process valts som testexempel. Processen har fyra tillstånd eller steg benämnda s1, s2, s3 och stop. Processen kan bara befinna sig i ett tillstånd. De olika stegen kan tillexempel vara uppstart, produktion, diskning och stop. Tillståndsändring sker då respektive tillstånds stegvillkor är uppfyllt. Dessa stegvillkor benämnes start, sv1, sv2 och sv3. Stegvillkoren är ofta kombinationer av olika villkor. Tillståndsändring sker enbart till nästa steg. I varje steg ska en utsignal u1, u2, u3 respektive u4 aktiveras. Då man hoppar till nästa steg deaktiveras det gamla stegets utsignal och det nya stegets utsignal aktiveras. Alarmindikatorn Alarm ska sättas om insignalen in är aktiverad. Detta ska ske oberoende av i vilket steg processen befinner sig i. De booleska ekvationerna blir då:

```
Stegsekvens.
s1   = stop and start  or s1   and not s2
s2   = s1   and sv1    or s2   and not s3
s3   = s2   and sv2    or s3   and not stop
stop = s3   and sv3    or stop and not s1

Alarm hantering.
alarm = in

Aktivering av utgångar.
u1 = s1
u2 = s2
u3 = s3
u4 = stop
```

Detta kan åskådliggöras grafiskt i en funktionsplan:



Ex. SATT

SATT har i sina system ett språk som bygger på några få och enkla instruktioner. Det finns booleska instruktioner och i de större systemen även aritmetriska instruktioner. De booleska grundinstruktionerna är:

Instruktion	Beskrivning
AD(N)nnnn	Logisk OCH(INTE)
OR(N)nnnn	Logisk ELLER(INTE)
SE(N)nnnn	Ett-(Noll)ställ kanal om villkor sant
AP	OCH vänsterparentes
OP	ELLER vänsterparentes
RP(N)	Högerparentes (INTE)

Instruktionen ADN utgör inversen (negeringen) till AD. samma sak gäller för ORN, SEN, och RPN. nnnn är adressen till en kanal eller minnescell. Instruktionen AD 0023 utför en och instruktion mellan processorns accumulator och minnescellen 0023. Resultatet lägges i accumulatorn. För att förstå funktionen måste man hela tiden tänka i termer av accumulator. Aritmetiken fungerar på liknande sätt. Testexemplet får med dessa instruktioner följande utseende:

Stegsekvensdelen:

```
  AD  0004       STOP
  AD  0014       START
  OR  0001       S1
  ADN 0002       S2
* SE  0001       S1
  AD  0001       S1
  AD  0011       SV1
  OR  0002       S2
  ADN 0003       S3
* SE  0003       S2
  AD  0002       S2
  AD  0012       SV2
  OR  0003       S3
  ADN 0004       STOP
* SE  0003       S3
  AD  0003       S3
  AD  0013       SV3
  OR  0004       STOP
  ADN 0001       S1
* SE  0004       STOP
```

Alarmhantering.

```
  AD  0500       IN
* SE  0009       ALARM
```

Aktivering av utsignaler.

```
  AD  0001       S1
* SE  0021       U1
  AD  0002       S2
* SE  0022       U2
  AD  0003       S3
* SE  0023       U3
  AD  0004       STOP
* SE  0024       U4
  END
```

Den Högra kolumnen innehåller variabelnamn som är knutna till adressen.

Ex Alfa Laval

Alfa Laval har utvecklat PC språket Alrite. Alrite har speciella språkelement för stegsekvenser och programmen delas in i subrutiner. En subrutin indelas i variabeldeklarationer, kalkyleringsdel som alltid genomlöpes samt en stegsekvensdel. Det finns även möjligheter att starta/stoppa en subrutin o.s.v. Testexemplet skrivet i Alrite utgör en subrutin och kan se ut så här:

```
VAR
BI    IN            1.1.2.1
BO    U1            1.1.1.1
BO    U2            1.1.1.2
BO    U3            1.1.1.3
BO    U4            1.1.1.4
ME    ALARM
ME    SV1
ME    SV2
ME    SV3
ME    STARTC
VAR END

PROG

CALC
      IN            ACT ALARM

STEP
 S1   STARTC        ACT U1
      SV1           ACT U2
      SV2           ACT U3
      SV3           ACT U4
      GO            JMP S1
ENDS

END
```

Efter VAR deklareras variablerna med typ namn och hårdvaruadress för in och ut signaler. ME står för minnesflagga, BI och BO för binär in- respektive utsignal. Hårdvaruadressen anger skåp, rack, kort och kanal för signalen. PROG definierar programmets början och END dess slut. Efter CALC står satser som alltid utföres. Dessa satser består av två fält. Det till höger innehåller ett villkor och det till vänster en åtgärd. I detta exemplet aktiveras alarm om in är sann annars deaktiveras alarm. Satserna mellan STEP och ENDS är var och en ett steg. Då programmet ligger i ett steg utföres stegets åtgärd, samtidigt testas villkoret på nästa rad. När detta så kallade stegvillkor blir uppfyllt deaktiveras föregående stegs åtgärd samtidigt som det nya stegets åtgärd aktiveras. Det är möjligt att utöka ett steg med flera satser. Normalt hoppar man till nästa steg men det är också möjligt att hoppa till vilket steg som helst. Det sista steget i exemplet har införts för att åstadkomma ett hopp till det första steget S1. Hoppet utföres endast om steget S1s villkor STARTC är uppfyllt. GO betyder att villkoret är sant. S1 används endast som en label. För enkelhetens skull har här ej angivits hur stegvillkoren får sina värden.

3. KRAV PÅ ETT STYRSYSTEM

Nedan följer en lista med krav på ett styrsystem. Kraven behandlar endast programmeringen och operatörskommunikationen.

1. Programmen bör vara ordnade med insignalinläsning först sen programexekvering samt sist utsignalskrivning. Detta för att förhindra att villkoren ändras under exekveringen av programmet.

2. Det ska vara möjligt för operatören att se aktuell status för alla in och utsignaler samt alla viktiga interna variabler som till exempel i vilket steg ett visst program befinner sig.

3. Operatören bör arbeta med samma variabelnamn som programskrivaren.

4. Det ska vara möjligt att modifiera programmen. Helst bör detta kunna ske under drift men man kan också tänka sig att ta delar av programmet ur drift när man gör ändringar.

5. De olika programmen ska kunna startas och stoppas av varandra samt av operatören

6. Det är en fördel kapacitetsmässigt och struktureringsmässigt om programmen kan fördelas i olika tasks eller processer med olika samplingstid.

7. En task bör kunna innehålla flera program.

8. Det ska gå att överföra variabler mellan olika program.

9. Det måste finnas något språkelement eller konstruktion som underlättar stegsekvenser.

10. Förutom stegsekvenser måste det finnas delar av programmet som alltid genomlöpes (för övervakning).

11. Det ska vara lätt att utföra logiska och aritmetiska beräkningar.

4. PC PROGRAM KONTRA VANLIGA PROGRAM

De fysiska förutsättningarna för ett PC program skiljer sig på flera punkter från vanliga program. Ett PC program måste till exempel övervaka ett stort antal signaler samtidigt medan ett vanligt program ofta bara väntar på en eller några få händelser. Det är inte ovanligt att ett PC system har 1000 - 4000 anslutna signaler. Ett PC program består ofta av 95%-100% av booleska beräkningar och villkor medan ett vanligt till stor del består av aritmetiska beräkningar och villkor. De språk som i dag används för PC programmering kontra vanlig programmering skiljer sig också. Med språk för vanlig programmering menas till exempel Ada eller Pascal. De flesta PC spåk ligger på samma nivå som assembler. För att hålla ordning på ett större PC program måste det struktureras enligt vissa regler. Program exemplet från SATT visar upp en struktur för stegsekvenser som ganska lätt kan göras om till ett språkelement. Det är just den strukturen Alrite understödjer. Våra vanliga högnivåspråk har också utvecklats från generella assembler konstruktioner men under andra förutsättningar och krav. Någon konstruktion som understödjer stegsekvenser finns inte inbyggt. Däremot finns en hel rad andra konstruktioner som loopar och if-then-else.

5. HUR KAN ADA ANVÄNDAS

När nu Ada är på väg att bli industristandard ställer sig frågan genast om språket är lämpat för PC programmering. Ada har inget språkelement som direkt understödjer stegsekvenser däremot finns alla de viktigaste booleska operatorerna med. Det är även möjligt att deklarera variabler av typen boolean. Nedan följer olika alternativa lösningar på testexemplet. Därefter finns ett förslag till systemlösning.

5.1 Stegsekvenser i Ada.

För att förenkla programmen i exemplen ges ej insignaler, utsignaler och variabler några värden.

Ada ex 1.

Som första exempel har valts en case konstruktion för att beskriva de olika tillstånden. Tillstånden finns lagrade i uppräkningstypen STEP. Detta exempel visar att case satsen ej är lämplig. Det blir för stor textmassa kring stegen vilket gör det svårt att överblicka stegsekvensen. Vidare måste alla variabler som är aktiva i ett steg återställas innan steget lämnas.

```
procedure ex1 is

type STEPT is (S1,S2,S3,STOP);
STEP : STEPT := STOP;

SV1,SV2,SV3,START : BOOLEAN;
U1,U2,U3,U4 : BOOLEAN := FALSE;
IN1 : BOOLEAN;
ALARM : BOOLEAN := FALSE;

begin
  loop

    case STEP is

      when S1   => if SV1
                   then
                     STEP := S2;
                     U1 := FALSE;
                   else
                     U1 := TRUE;
                   end if;
      when S2   => if SV2
                   then
                     STEP := S3;
                     U2 := FALSE;
                   else
                     U2 := TRUE;
                   end if;
      when S3   => if SV3
                   then
                     STEP := STOP;
                     U3 := FALSE;
                   else
                     U3 := TRUE;
                   end if;
      when STOP => if START
                   then
                     STEP := S1;
                     U4 := FALSE;
                   else
                     U4 := TRUE;
                   end if;
    end case;

  ALARM := IN1;

  end loop;
end;
```

Ada ex 2.

I detta exempel har de booleska ekvationerna från definitionen av test exemplet tagits och skrivits rätt upp och ner. Denna lösning ger betydligt mer kompakt text. Nackdelen här är att det är svårt att överblicka stegvillkoren. Samtidigt är det lätt att skriva fel samt svårt att ändra befintliga stegvillkor.

```
procedure Ex2 is

S1,S2,S3 : BOOLEAN;
STOP : BOOLEAN := TRUE;

SV1,SV2,SV3,START : BOOLEAN;
U1,U2,U3,U4 : BOOLEAN := FALSE;
IN1 : BOOLEAN;
ALARM : BOOLEAN := FALSE;

begin
  loop

  -- Stegsekvens.
  S1   := (STOP and START)  or  (S1   and not S2);
  S2   := (S1   and SV1)    or  (S2   and not S3);
  S3   := (S2   and SV2)    or  (S3   and not STOP);
  STOP := (S3   and SV3)    or  (STOP and not S1);

  -- Alarm hantering.
  ALARM := IN1;

  -- Aktivering av utgångar.
  U1 := S1;
  U2 := S2;
  U3 := S3;
  U4 := STOP;

  end loop;
end;
```

Ex Ada 3

Strukturen på exempel 2 är tilltalande frånsett stegvillkoren. För att råda bot på detta har det i exempel 3 tillförts ett generiskt paket med namnet State keeper. State keeper har stegen eller tillstånden som en generisk parameter. Detta medför att tillståndet kan skyddas inuti det generiska paketet och endast påverkas via paketets procedurer.

```
generic
    type Statelist is (<>);
package Statekeeper is
  function State return Statelist;
  -- Returns the actual state list.

  function Teststate(S : in Statelist) return BOOLEAN;
  -- Returns true if S is the actual state else false.

  procedure Changeto(S : in Statelist; Cnd : in BOOLEAN);
  -- If Cnd is true then the actual state is changed to S.
end Statekeeper;

package body Statekeeper is

  Actualstate: Statelist;

  function State return Statelist is
  begin
    return actualstate;
  end State;

  function Teststate(S: in Statelist)return BOOLEAN is
  begin
    return (actualstate = S);
  end Teststate;

  procedure Changeto(S : in Statelist; Cnd : in BOOLEAN) is
  begin
    if Cnd then actualstate := S; end if;
  end Changeto;

begin

  actualstate:=Statelist'FIRST;

end Statekeeper;
```

Med hjälp av det generiska paketet State keeper får Ex 3 följande utseende.

```
With Statekeeper;

procedure Ex3 is

    type Statelist is ( S1 , S2 , S3 , Stop );

    package Mystatekeeper is new Statekeeper(Statelist);
      use Mystatekeeper;

    Sv1,Sv2,Sv3,Start  : BOOLEAN:=true;
    Ut1,Ut2,Ut3,Ut4    : BOOLEAN:=false;
    In1                : BOOLEAN;
    Alarm              : BOOLEAN:=false;

begin
   loop

    case State() is
      when Stop => Changeto( S1   , Cnd => Start );
      when S1   => Changeto( S2   , Cnd => Sv1   );
      when S2   => Changeto( S3   , Cnd => Sv2   );
      when S3   => Changeto( Stop , Cnd => Sv3   );
    end case;

    Alarm := In1;

    Ut1 := Teststate(S1);
    Ut2 := Teststate(S2);
    Ut3 := Teststate(S3);
    Ut4 := Teststate(Stop);

  end loop;
end Ex3;
```

Ofta är stegvillkoren betydligt mera komplicerade. Det kan då vara svårt att få plats på en rad. Detta kan lösas enklt så här:

```
  when S1  => Sv1 := not Alarm and Pumpon and Ready;
              Changeto( S2, Cnd => Sv1 );
  when S2  => ...
  .
  .
```

5.2 System lösning.

Hittills har bara program innehållande en stegsekvens studerats. För att få ett fungerande system måste olika stegsekvenser kunna kommunicera med varandra samt med IO och operatören. Ett system bör också innehålla sådana funktioner som tex timers. På de följande sidorna finns ett exempel på hur ett sådant system kan byggas upp.

Timers.

Timers används i PC program för att mäta tider för olika ändamål. Typexemplet är att en timer aktiveras i ett steg och att stegvillkoret till nästa steg är när timern löpt ut. Ett paket Timer handler med typen Timer och ett antal procedurer som opererar på Timer kan se ut så här.

```
with CALENDAR; use CALENDAR;

package Timerhandler is

   type Timer is private;

   procedure Initialize(T : in out Timer; Int: Duration);

   procedure Activate(T : in out Timer; Cnd : BOOLEAN);
   -- The timer restarts if Cnd is true.
   -- If Cnd is false it Stops.

   procedure Restart(T : in out Timer);
   -- Restarts the timer.

   procedure Start(T : in out Timer);
   -- Starts the timer after a Stop.

   procedure Stop(T : in out Timer);
   -- Stops the timer and saves the rest time.

   function Timeout(T : Timer) return BOOLEAN;
   -- Checks if timeout is reached.


private

   type Timer is
     record
       Ti       : TIME;
       Interval : DURATION;
       Rest     : DURATION;
       Start    : BOOLEAN;
     end record;

end Timerhandler;
```

```
package body Timerhandler is

  procedure Initialize(T : in out Timer; Int: Duration) is
  begin
        T.ti        := Clock();
        T.Interval := int;
        T.Rest      := int;
        T.Start     := false;
  end Initialize;

  procedure Activate(T : in out Timer; Cnd : BOOLEAN) is
  begin
    if Cnd then
      if not T.Start then
        T.Ti := CLOCK() + T.Interval;
        T.Rest := T.Interval;
        T.Start := TRUE;
      end if;
    else
      T.Start := FALSE;
    end if;
  end Activate;

  procedure Restart(T : in out Timer) is
  begin
    T.Ti := CLOCK() + T.Interval;
    T.Rest := T.Interval;
    T.Start := TRUE;
  end Restart;

  procedure Start(T : in out Timer) is
  begin
    if not T.Start then
      T.Start := TRUE;
      T.Ti := CLOCK() + T.rest;
    end if;
  end Start;

  procedure Stop(T : in out Timer) is
  begin
    if T.Start then
      T.Rest := T.Ti - CLOCK();
      T.Start := FALSE;
    end if;
  end Stop;

  function Timeout(T : Timer) return BOOLEAN is
  begin
    return ( (T.Ti - CLOCK()) < 0.0 );
  end Timeout;

end Timerhandler;
```

IO hantering.

I de flesta PC system krävs det speciella rutiner för att hantera IO. För detta ändamål finns ett paket IO task. Detta paket innehåller all information om hur och var olika IO kanaler ska läsas respektive skrivas. Paketet IO tasks utseende beror av hårdvaran och tas ej upp här. För att göra det möjligt att använda egna IO namn i sekvensprogramen har det generiska pakete IO införts. De utvalda IO signalernas namn utgör de generisk parametrarna. Kopplingen mellan namn och IO kanal anges i sekvensprogrammen. Det generiska paketet IO har följande utseende.

```
with IOtask; use IOtask;

generic
  type Inputname is  (<>);
  type Outputname is (<>);
package IO is
  type Inputvalue    is array (Inputname range <>) of BOOLEAN;
  type Inputaddress  is array (Inputname range <>) of INTEGER;

  type Outputvalue   is array (Outputname range <>) of BOOLEAN;
  type Outputaddress is array (Outputname range <>) of INTEGER;

  procedure ReadIO (V : out Inputvalue; A : in Inputaddress);
  procedure WriteIO(V : in Outputvalue; A : in Outputaddress);
end IO;

package body IO is

  procedure ReadIO(V : out Inputvalue; A : in Inputaddress) is
  begin
    for I in A'FIRST..A'LAST loop
      V(I) := GetIO(A(I));
    end loop;
  end ReadIO;

  procedure WriteIO(V : in Outputvalue; A : in Outputaddress) is
  begin
    for I in A'FIRST..A'LAST loop
      SetIO(V(I),A(I));
    end loop;
  end WriteIO;

end IO;
```

Flera sekvensprogram och kommunikation mellan dem.

Processen som styrs av sekvensprogrammen kräver ständig övervakning och det är därför naturligt att genomlöpa programmen periodiskt. Samtidigt bör operatören ha möjlighet att kommunicera med programmet. För att göra detta möjligt och samtidigt inte förlora överskådligheten har sekvensprogrammen lagts i var sitt paket. För varje anrop till paketet genomlöpes sekvensprogrammet en gång. Med i anropet finns in och ut variabler till programmet.

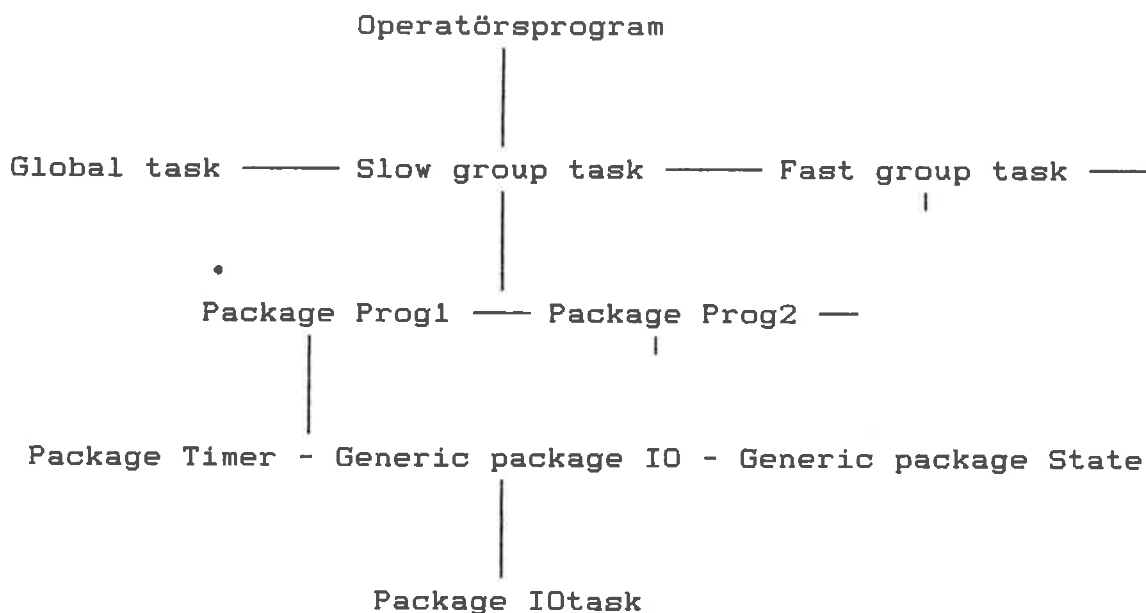
```
With IO,Statekeeper,Timerhandler;
Use Timerhandler;

package Prog1pac is

  type P1globalname is (Start, Sv1, Sv2, Alarm);
  type Prog1values is array (P1globalname) of BOOLEAN;

  procedure Prog1(Gbl : in out Prog1values);

end Prog1pac;

package body Prog1pac is

  Type Inputlist  is (In1);
  Type Outputlist is (Out1,Out2,Out3,Out4);

  package MyIO is new IO(Inputlist,Outputlist);
    use MyIO;

  Inv : Inputvalue(inputlist);
  Inaddress : constant Inputaddress   :=
    (In1 => 101);

  Outv : Outputvalue(outputlist);
  Outaddress : constant Outputaddress :=
    (Out1 => 201,
     Out2 => 202,
     Out3 => 203,
     Out4 => 204);


  type Statelist is ( S1 , S2 , S3 , Stop );

  package Mystatekeeper is new Statekeeper(Statelist);
    use Mystatekeeper;

  Timer1 : Timer;
```

```
  procedure Prog1(Gbl : in out Prog1values) is
```

```
  begin

    ReadIO(Inv,Inaddress);

    case State() is
      when Stop => Changeto( S1   , Cnd => Gbl(Start)      );
      when S1   => Changeto( S2   , Cnd => Gbl(Sv1)        );
      when S2   => Changeto( S3   , Cnd => Gbl(Sv2)        );
      when S3   => Changeto( Stop , Cnd => Timeout(Timer1) );
    end case;

    Activate( Timer1, Cnd => Teststate(S3) );
    Gbl(Alarm) := Inv(In1);

    Outv(Out1) := Teststate(S1);
    Outv(Out2) := Teststate(S2);
    Outv(Out3) := Teststate(S3);
    Outv(Out4) := Teststate(Stop);

    WriteIO( Outv , Outaddress );

  end Prog1;

begin

Initialize(Timer1, 1.0);

end Prog1pac;
```

Jämfört med exempel 3 har här Sv3 ersatts med en timer med namnet Timer1. Timer1 aktiveras då programmet befinner sig i tillståndet S3. Stegvillkoret för detta tillstånd blir uppfyllt då Timer1 löpt ut. Värdena på Start, Sv1 och Sv2 fås från andra program.

Nu återstår en övergripande programstruktur samt operatörskommunikation. Det finns då flera anledningar att gruppera sekvensprogrammen i olika tasks. Dels kan tidskraven, det vill säga samplingstiden, variera betydligt. Dels kan det rent strukturerings- och funktions-mässigt vara lämpligt med en uppdelning. Det är också möjligt att gå ett steg längre och göra varje stegsekvens till en egen task. Detta är dock med dagens hårdvara ineffektivt rent exekveringsmässigt. Likaså är det tveksammt om sekvenser med stark koppling mellen sig ska läggas i olika tasks. Dessa grupp tasks måste naturligtvis kunna kommunicera med varandra via globala variabler. För att göra detta möjligt och samtidigt skydda variablerna lägges dessa i global task. Dessa tasks har i detta exempel lagts tillsammans med operatörshanteringen som utgör huvudprogram. Nedan finns en schematisk bild över programsystemet.



Systemet kan lätt utvidgas med reglerfunktioner. Regulatorer ska samplas periodiskt och samplingstiden väljes i förhållande till den process som regleras. Med dessa krav faller det sig naturligt att införa så kallade Regler tasks på samma nivå som group tasksen. Det blir då lätt att kommunicera mellan sekvensprogram och regulatorer.

Nedan finns ett enkelt program enligt ovanstående system. Operatören har möjlighet att sätta variabeln Start i Prog1 i Slowgroup samt att inspektera om den globala variabeln Alarm är satt. Fastgroup sätter den globala variabeln Gblalarm. Slowgroup använder samma variabel som insignal till Prog1. I Slowgroup överföres Agitate från Prog1 till Start i Prog2.

```
with TEXTIO; use TEXTIO;
with CALENDAR; use CALENDAR;
with Prog1pac, Prog2pac, Prog3pac;
use  Prog1pac, Prog2pac, Prog3pac;

procedure Main is

  task Global is
    entry Operator(Oalarm : out BOOLEAN);
    entry Fastvar(F1 : in BOOLEAN);
    entry Slowvar(S1 : out BOOLEAN);
  end Global;

  task Slowgroup is
    entry Operator(Start : in BOOLEAN);
  end Slowgroup;

  task Fastgroup is
    entry Operator;
  end Fastgroup;
```

```
  task body Global is
    Gblalarm : BOOLEAN := true;
  begin
    loop
      select
        accept Operator(Oalarm : out BOOLEAN) do
        Oalarm := Gblalarm;
        end Operator;
      or
        accept Fastvar(Falarm : in BOOLEAN) do
        Gblalarm := Falarm;
        end Fastvar;
      or
        accept Slowvar(Salarm : out BOOLEAN) do
        Salarm := Gblalarm;
        end Slowvar;
      end select;
    end loop;
  end Global;
```

```
task body Slowgroup is
  Sampletime : DURATION := 1.0;
  Nexttime : TIME := CLOCK();

  P1 : Prog1values;
  P2 : Prog2values;

begin
  loop
    select
      accept Operator(Start : in BOOLEAN) do
      P1(Start1) := Start;
      end Operator;
    or
      delay Nexttime - CLOCK();
      Nexttime := Nexttime + Sampletime;

      Global.Slowvar( P1(Alarm) );


      Prog1(P1);
      P2(Start) := P1(Agitate);

      Prog2(P2);

    end select;
  end loop;
end Slowgroup;
```

```
task body Fastgroup is
  Sampletime : DURATION := 0.5;
  Nexttime : TIME := CLOCK();

  P3 : Prog3values;

begin
  loop
    select
      accept Operator;
    or
      delay Nexttime - CLOCK();
      Nexttime := Nexttime + Sampletime;

      Prog3(P3);

      Global.Fastvar( P3(Alarm) );

    end select;
  end loop;
end Fastgroup;
```

```
  Inchar : CHARACTER;
  Alarm  : BOOLEAN;

begin

  -- Code for the Operator communication......
  loop
    PUT("Enter command");
    GET(Inchar);
    case Inchar is
      when 'S' => PUT("Start Prog1");
                  Slowgroup.Operator(TRUE);
      when 'A' => Global.Operator(Alarm);
                  if Alarm
                  then
                    PUT("ALARM");
                  else
                    PUT("No alarm");
                  end if;
      when others => PUT("Illegal command");
    end case;
  end loop;

end Main;
```

6. SLUTSATSER

Det är lätt att bygga upp programpaket i Ada. Dessa Programpaket kan användas för att dölja mindre intressanta detaljer. På detta sätt fås renare programstrukturer. Adas hårda typning ställer dock till en del problem. En del av dessa problem har här lösts med hjälp av generiska paket. En annan fördel med Ada är att hela systemet med operatörskommunikation, pcprogarm och reglerloopar kan skrivas i samma språk. Ada har också tasks som språkelement vilket gör att man slipper speciella operativsystem. Ett krav som ej här har uppfyllts är modifiering av programmet under drift. Man bör kunna ändra och lägga till program. Ada som programspråk tycks ha alla förutsättningar för att kunna användas i styr och regler system. Däremot har det ännu ej dykt upp tillräckligt bra kompilatorer och hårdvara för att tidsmässigt kunna konkurera med dagens system.

REFERENSER.

(1) Reference Manual for the Ada Programming Language, United States Department of Defense Jan/83.
(2) Alrite Reference Manual, Alfa Laval Automation AB.
(3) Function Chart. Högnivåspråk för sekvensstyrning, Automation Mars/84.
(4) System PBS, Satt Control.