ISSN 0280–5316
ISRN LUTFD2/TFRT--5469--SE

Liten, snabb realtidskärna i C

implementation för DSP med exempel

Anders Carlsson

Institutionen för Reglerteknik
Lunds Tekniska Högskola
Maj 1993

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

Document name
MASTER THESIS

Date of issue
May 1993

Document Number
ISRN LUTFD2/TFRT--5469--SE

Author(s)
Anders Carlsson

Supervisor
Anders Blomdell

Sponsoring organisation

Title and subtitle
Liten, snabb realtidskärna i C, implementation för DSP med exempel

Abstract

A real-time kernel has been developed for a DSP system for fast digital control applications. The system consists of a NB-DSP2300 board from National instruments corp. located inside a Macintosh II series computer. This thesis describes the developement process, hardware specifics, the kernel Application programming interface (API) and communication primitives for interfacing to the LabVIEW software package.

Also the thesis describes an example application using the kernel. The application is a three-phase induction machine controller, wich includes a full-order state observer. The controller performs very well, just being a basic setup, and may well be developed further.

Key words
Real-time programming, induction-machine control

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
Swedish

Number of pages
92

Recipient's notes

Security classification

Innehåll

Inledning

Detta examensarbete har utförts huvudsakligen vid institutionen för reglerteknik, Lunds Tekniska Högskola, höstterminen 1992. Huvudsakligen, eftersom visst arbete även har utförts vid institutionen för industriell elektroteknik och automation, IEA.

Jag har varit ensam om mitt arbete, till skillnad från de flesta teknologer (Det normala, och rekommenderade, är ju att man är två). Jag har dock inte upplevt detta som en nackdel, mycket tack vare min handledare som ställt upp som diskussionspartner när jag funderat över hur jag skulle fortsätta.

Bakgrund

Upprinnelsen till detta examensarbete är ett projekt som jag och fyra andra teknologer genomförde vid institutionen för reglerteknik vid LTH, våren 1992. Detta projekt gick ut på att konstruera en snabb, adaptiv regulator med hjälp av en signalprocessor, TMS320C30, monterad på ett instickskort i en Macintosh II. Vi lyckades med att få ihop en fungerande regulator, men arbetsinsatserna för att komma dit blev så stora att vi varken kunde få den driftsäker eller åstadkomma ett vettigt användargränssnitt.

Att utvecklingsmiljön lämnade en del övrigt att önska visste man redan. Vår projekthandledare, Anders Blomdell, frågade redan i början av projektet om vi var intresserade av att skriva en realtidskärna till systemet. Jag tyckte det var en intressant idé, men ogenomförbar inom ett tvåveckorsprojekt. Eftersom jag hade letat efter ett examensarbete ett tag frågade jag Anders Blomdell om jag kunde göra realtidskärnan som examensarbete och om han var intresserad av att handleda. Anders kunde mycket väl tänka sig att handleda och vi kunde komma överens om vilken tidsperiod som det kunde bli fråga om.

Målsättning

Målet med examensarbetet har dels varit att implementera en liten och effektiv realtidskärna med gängse tillbehör såsom drivrutiner, kommunikationsverktyg och andra hjälpmedel. Dels att utveckla ett exempelprogram, skrivet för den här kärnan, som visar hur man utnyttjar funktionerna som kärnan erbjuder. Det beslutades inte på förhand VAD detta exempelprogram skulle göra.

Senare under arbetets gång kom jag i kontakt med Mats Alaküla vid institutionen för Industriell Elektroteknik och Automation, IEA. Mats fick höra vad jag höll på med, och blev genast intresserad av ett samarbete. Ur detta möte växte det fram vad som skulle bli tillämpningsexemplet.

Målen får nog sägas ha blivit uppfyllda. Den resulterande realtidskärnan verkar vara stabil. Exempelapplikationen tjänar inte bara som utvärdering av min realtidskärna utan även för hela systemet, som därmed visar vad man kan åstadkomma med kombinationen Macintosh, DSP och LabVIEW. Resultaten har varit så intres-

santa att IEA beslutat sig för att investera i ett liknande system som det jag har utvecklat realtidskärnan för, för att använda i sin forskningsverksamhet.

Tillerkännanden

Slutligen vill jag tacka personer som hjälpt mig under arbetets gång. Min handledare, Anders Blomdell, får tack för alla diskussioner om implementationsdetaljer och för hjälp och inspiration när jag hade kört fast. Mats Alaküla, utan vars strömriktarutrustning mitt programexempel hade varit komplett omöjligt att genomföra, och för styrmetoden, så att jag slapp räkna ut det själv. Karl Johan Åström, professor som stod för utrustningen, och institutionen med förresten. Gustav Olsson och Jaroslav Valis, inspirerande professorer vid IEA, utan vilka IEA inte hade varit vad den är.

Kapitel 1

Utvecklingsmiljö

Detta kapitel behandlar översiktligt den hårdvara och mjukvara som använts under utvecklingsarbetet.

1.1 Hårdvara

1.1.1 Översikt

Det system som använts för att utveckla Ärkenöt har bestått av en Macintosh IIfx, i vilken det har monterats ett instickskort från National Instruments, NB-DSP2300, med en signalprocessor, Texas Instruments TMS320c30.

Detta instickskort har varit målsystem för Ärkenöt. Förutom kortet med signalprocessorn, har det funnits ett universellt kort för I/U, NB-MIO-16-25L, samt ett kort med sex analoga utgångar, NB-AO-6. Även dessa kort kommer från National Instruments.

I de avsnitt som följer finns översiktliga beskrivningar av de olika instickskorten och deras egenskaper.

1.1.2 NB-DSP2300

NB-DSP2300 är ett avancerat kort för signalbehandling i realtid eller för bruk som hjälpprocessor till Macens huvudprocessor. Det senare fallet understöds av tillverkaren med omfattande biblioteksrutiner och utvecklingshjälpmedel för applikationer som utnyttjar kortet.

I fallet med realtidsapplikationer finns det stöd i form av en instruktionsbok som beskriver kortet någorlunda detaljerat och ger ett applikationsexempel. Dessutom levererar tillverkaren, National Instruments, en uppsättning utvecklingsverktyg, C-kompilator, avlusare samt ytterligare programexempel.

Gemensamt för dessa programexempel är att de till mycket liten del innehåller verkligt återanvändbar kod. Vill man bygga egna applikationer får man antingen börja från grunden eller modifiera något exempel.

1.1.3 NB-MIO-16

NB-MIO-16-25L är ett kort ur en hel familj universalkort för analog/digital I/U till Macintosh II som National Instruments levererar. Korten skiljer sig vad gäller känslighet och upplösning på de analoga ingångarna samt på tiden det tar att omvandla ett analogt värde till ett digitalt.

NB-MIO-16 familjen har 8 digitala in- alt. utgångar. Det finns fyra alternativ för hur man kan använda dem:

- Som 8 digitala utgångar.
- Som 4 digitala ingångar plus 4 digitala utgångar, två alternativ.
- Som 8 digitala ingångar.

Familjen har en A/D-omvandlare med 12 alternativt 16 bitars upplösning. Till denna är kopplad en analog multiplexer, samt en instrumentförstärkare med variabel förstärkning. På kortet finns byglingar som bestämmer inkopplingen av insignalerna via multiplexern till förstärkaren. Man kan välja att använda obalanserade ingångar, vilket ger tillgång till 16 st, eller balanserade ingångar, 8 st.

Förstärkarens förstärkning kan varieras programmässigt. Vilka förstärkningar som finns tillgängliga beror på vilken version av kortet man har. I drivrutinerna till detta kort har ingen hänsyn tagits till detta. De analoga ingångarna programmeras för att ha förstärkningen ett (detta alternativ är alltid tillgängligt). Insignalsområdet är i detta fall från -10 V till $+10$ V.

Korten har också två analoga utgångar via D/A-omvandlare med 12 bitars upplösning. Dessa kan konfigureras för att ha området 0 V – $+10$ V, -5 V – $+5$ V, samt -10 V – $+10$ V. Detta sista alternativ är det enda som understöds.

Avslutningsvis finns det en räknar/klockkrets på kortet. Två av de fem räknare som finns i denna krets är helt ägnade åt att kontrollera A/D omvandlaren, men de resterande tre har fått sina signaler utdragna till anslutningsdonet. Dessa tre räknare kan användas för räkning av pulser, tidmätning, tidsstyrning, pulsbreddsmodulering etc.

1.1.4 NB-AO-6

Detta kort har sex analoga utgångar. De har 12 bitars upplösning, liksom de analoga utgångarna på NB-MIO-16. Dessa utgångar har ännu fler möjliga funktionssätt, men för att förenkla programmeringen har det antagits att de är lika som på NB-MIO-16 kortet.

1.2 MPW Shell

MPW Shell är den programutvecklingsmiljö som Apple levererar till sina Macintoshdatorer. Denna miljö är uppbyggd med UNIX som förebild, men konstruktörerna har valt att undvika UNIX:s kryptiska kommandonamn. Likheten med UNIX gör det relativt lätt att flytta en kompilator från UNIX eller DOS eller VMS till MPW under MacOS.

Detta har utnyttjats av av Texas Instruments som tack vare MPW kan leverera kompilatorer för sina signalprocessorer som kan köras på en Macintosh. Tyvärr har Texas Instruments valt att inte göra några anpassningar till MPW:s utvecklingsmiljö mer än att man kompilerat dem för MPW. Mest notabelt är detta vad gäller felmeddelanden. I normala fall kan man t.ex markera och sedan exekvera ett felmeddelande från en kompilator som en kommandorad och får då upp ett fönster med rätt källkodsfil med den felaktiga raden markerad.

När man använder Texas Instruments kompilator kan man tyvärr inte göra på detta eleganta sätt eftersom felutskrifterna inte är kompatibla med MPW shell.

1.3 Avlusaren DSP Debug

Den avlusare som använts är en Macintosh-applikation vid namn DSP Debug som levererats av National Instruments Inc. DSP Debug är en trevligt hjälpmedel för avlusning av program som exekveras på NB-DSP2300. Detta gäller i första hand när

man kör program som inte utnyttjar periodiska avbrott för att reglera sin verksamhet. Program som utnyttjar periodiska avbrott (eller avbrott överhuvudtaget) för att klara sina arbetsuppgifter är däremot betydligt besvärligare att avlusa, främst beroende på konstruktionsmissar som National Instrument har gjort när de konstruerat NB-DSP2300. Eftersom Ärkenöt är i högsta grad beroende av avbrott för att fungera, så blir det följaktligen besvärligt att avlusa program som körs under Ärkenöt.

Problem med DSP Debug Orsaken till problemen för DSP Debug är att den fasta mjukvaran på NB-DSP2300 inte innehåller något stöd för användning av TRAP-instruktioner som brytpunkter. I stället är DSP Debug hänvisad till att utföra ett CALL till den ROM-rutin som hanterar brytpunkter på NB-DSP2300. Denna rutin stänger inte av avbrotten ordentligt, varför NB-DSP2300 genast börjar exekvera program igen, med det troliga resultatet att programmet och därmed NB-DSP2300 kraschar. När detta hänt är man tvungen att starta om värddatorn för att kunna använda NB-DSP2300 igen.

Ärkenöt och DSP Debug Som konstaterades ovan, kan man inte stega sig fram genom sitt program eller sätta brytpunkter för att följa programmets exekvering när man använder Ärkenöt. Vad man kan göra är att inspektera NB-DSP2300:s minne för att på så sätt kunna följa variabelvärden. Man kan inspektera de olika trådarnas stackar, och på så sätt kunna få en uppfattning om var en tråd har stannat t.ex. Tyvärr förutsätter dessa tekniker en ingående kunskap om hur Ärkenöt, NB-DSP3200, kompilator, länkare och assembler fungerar för att vara effektiva.

Min plan för att komma tillrätta med problemen är att tillföra funktioner till Ärkenöt som kan ta över avlusningen från NB-DSP2300:s ursprungliga mjukvara, samt ändra i DSP Debug så att den utnyttjar Ärkenöts avlusningsfunktioner i stället.

1.4 LabVIEW

LabVIEW är ett program som utvecklats av National Instruments. Detta program är till för att bygga upp vad som kallas virtuella instrument. Ett virtuellt instrument i LabVIEW består av en frontpanel samt ett kopplingsschema som visar hur instrumentet är uppbyggt. Frontpanelen byggs upp av olika byggelement såsom visarinstrument, grafer, sifferindikatorer, knappar, rattar, skjutskalor, indikeringslampor, etc.

I kopplingsschemat finns frontpanelens kontroller och indikatorer representerade som symboler, som anger vilken datatyp som symbolen avger eller tar emot. För att få den funktionalitet man önskar av sitt instrument bygger man upp ett kopplingsschema med hjälp av de funktioner som LabVIEW tillhandahåller.

LabVIEW erbjuder motsvarigheter till normala programspråks kontrollstrukturer, aritmetiska/logiska operationer, matematiska funktioner, omvandlingar, filhantering, strängfunktioner samt ett stort antal färdiga virtuella instrument. I kopplingsschemat fungerar ett virtuellt instrument som en funktion eller en subrutin i ett normalt programspråk.

Kodgränssnitt i LabVIEW Om man behöver utföra någon särskild operation, som inte redan finns som färdig funktion i LabVIEW, kan man skapa vad som kallas för en Code Interface Node (CIN). Den placeras ut i kopplingsschemat och förses med lämpligt antal ingångar respektive utgångar. Därefter kan man skriva sin funktion i något normalt programspråk (C, Pascal, assembler) som man kompilerar och sedan länkar med ett bibliotek som gör att man kan lägga in sin funktion i ett virtuellt instrument.

Exempel på detta är de funktioner som sköter om överföring av data mellan NB-DSP2300 och LabVIEW.

Kapitel 2

Att implementera en realtidskärna

Detta kapitel handlar om mitt arbete med att implementera Ärkenöt. Kapitlet berör implementationsdetaljer såsom datastrukturer och gränssnitt till hårdvaruberoende delar. Det diskuteras också om hur man borde eller inte borde arbeta när man utvecklar en realtidskärna.

2.1 Inledning

När man bestämt sig för att göra en realtidskärna inställer sig frågan: hur? Vad skall man tänka på? I vilken ordning skall man göra saker för att undvika de största svårigheterna?

Det finns förmodligen lika många svar på de här frågorna som det finns programmerare som gjort en egen kärna någon gång. Jag har inte för avsikt att ge det slutliga svaret på frågorna, men i det följande kommer jag att beskriva hur jag gjorde.

2.2 Avlusningsverktyg

Innan man börjar med att implementera en kärna bör man se till att ha tillgång till vettiga avlusningsverktyg. Med vettiga menas i det här sammanhanget att någorlunda entydigt kunna spåra fel trots att datorns verksamhet störs av avbrott från externa enheter. Det är önskvärt att kunna exekvera program steg för steg, sätta brytpunkter, och generera felutskrifter när något går snett. Om man inte har verktyg som klarar minst en av dessa funktioner, får man vackert sätta sig ned och försöka ordna fram sådana verktyg. I annat fall kommer man att få det besvärligt i fortsättningen.

2.2.1 DSP Debug

När jag påbörjade mitt arbete fanns det en avlusare på assemblernivå, DSP Debug. Denna avlusare klarar tyvärr inte att avlusa program i realtidsmiljö. För att kunna avlusa realtidskärnan var jag därför tvungen att skriva en Macintosh-applikation som möjliggjorde felutskrifter från signalprocessorn. Resultatet av detta delprojekt är applikationen `DSP Console`.

2.2.2 DSP Console

DSP Console är en mycket enkel applikation som läser en buffert, förlagd i delat minne på DSP2300, som text och skriver ut den i ett fönster på Macinsosh II:s bildskärm. För att signalprocessorn enkelt skall kunna skriva text i bufferten, har standardfunktionerna `printf` och `puts` implementerats.

Från början fanns det bara en, speciell buffert, som till sig hade knutet en assemblerfunktion som skötte om all skrivning av data till bufferten. DSP Console hade i sig en motsvarande funktion som hämtade data ur bufferten med jämna intervall.

Senare, med mera välutvecklade kommunikationskanaler har DSP Console ändrats till att utnyttja `shared buffers`-paketet i stället.

DSP Console är skriven skriven med hjälp av MacApp. Detta har gjort det möjligt att inkludera utskrift och dokumentering av resultatutskrifter från signalprocessorn utan att egentligen skriva någon kod själv, mer än den som kommunicerar med signalprocessorn.

2.3 Korutiner

När man nu kan avlusa program, kan man fortsätta med att implementera vad som kallas för korutiner. En korutin innehåller allting som processorn behöver för att kunna exekvera program. Den innehåller också information som gör det möjligt för processorn att byta mellan olika korutiner. Ett korutinpaket innehåller funktioner för att byta mellan olika korutiner, för att initialisera korutiner, samt för att hantera avbrott.

2.3.1 Kontextbytet

Det första man måste få att fungera är kontextbytet (eng. context switch). Kontextbytet är den kod som får processorn att byta från en korutin till en annan. Denna kod är den mest grundläggande i alla former av operativsystem, realtidskärnor, och andra programsystem, som utger sig för att kunna erbjuda fleruppdragskörning. Dessutom är kontextbytet en av de få delar av en realtidskärna som måste skrivas i assembler och som därmed är beroende av den speciella dator man skriver kärnan för.

Innan man skriver koden för kontextbytet är det lämpligt att utforma kontextens datastruktur. Det finns många olika vägar att gå, var och en av med sina fördelar. Vilken modell man väljer beror på det valda datorsystemet, den valda processorn, minnesutrymme etc. och det kan vara lämpligt att utforma gränssnittet mot de överordnade funktionerna så att en ändring av kontextbytet (t.ex om man byter processorarkitektur) inte behöver innebära att man måste ändra i de överordnade funktionerna.

Här skall påpekas vikten av att kontextbytet hanterar avstängningen av avbrott på rätt sätt. Om så inte är fallet kommer datorsystemet aldrig att vara pålitligt. Grundprincipen är att kontextbytet aldrig försöker göra något själv. Givetvis måste avbrotten stängas av medan man byter stack, men den kontext man går in i skall själv innehålla information om ifall avbrott är tillåtna eller ej.

2.3.2 Avbrottshantering

Avbrottshanteringen kan man lösa på olika sätt. Den kanske vanligaste varianten är att man ansluter en procedur (avbrottshanterare) till avbrottet. Avbrottshanteraren anropas då varje gång processorn får det avbrott som avbrottshanteraren är ansluten till.

```
typedef struct context_rec {
void  *SP;
void *stack_limit;
void *stack;
} context_rec,*context_t;

extern const size_t stack_safety_space;

void setup_context(context_t ctxt,
   void (*entry_point)(),
   void* stack,
   size_t stack_size);
/*
 * setup_context performs initialization of the context_rec
 * pointed to by ctxt and the initial stack pointed to by stack.
 * it is the users responsbility to allocate
 * the stack and the record.
 */
void switch_context(context_t target);
/*
 * Transfers control to the target context.
 */
context_t current_context(void);
/*
 * Returns currently executing context
 */
#define context_stack(_ctxt) ((_ctxt)->stack)
```

Figur 2.1: Gränssnitt till den del av Ärkenöt som utför själva kontextbytet. Implementationen av detta gränssnitt är maskinspecifik, dvs den kan inte flyttas mellan olika processorer.

```
long cpu_disable();
/*
 * cpu_disable() disables maskable interrupts to the CPU and
 * returns word containing information about the interrupt
 * enable state before the call.
 */
void cpu_reenable(long);
/*
 * cpu_reenable() reenables interrupt disabled by cpu_disable
 * the argument is the return-value from cpu_disable.
 */
void (*intvec(intr_id interrupt,void (*intr_handlr)()))();
/*
 * intvec installs intr_handlr as the interrupt service routine
 * for interrupt and then returns the old handler
 */
```

Figur 2.2: Avbrottshanteringen i Ärkenöt. Liksom för kontextbytet kan implementationen inte flyttas mellan olika maskiner.

I korutinsammanhang kan man emellertid nämna den modell som Wirth valt i Modula-2, nämligen IOTRANSFER. IOTRANSFER byter korutin och placerar den anropande korutinen i vänteläge för ett valt avbrott. När avbrottet sedan inträffar återvänder IOTRANSFER automatiskt. På detta vis har man alltså en hel korutin som avbrottshanterare i stället för en procedur.

För Ärkenöt valt procedurvarianten valts, dels för att den är effektivare, dels för att den är lite lättare att implementera.

I detta sammanhang vill jag också uppmärksamma läsaren på funktionerna `cpu_disable()` och `cpu_reenable()`. Dessa två funktioner används för att göra systemanrop till atomära operationer, d.v.s att ett systemanrop inte kan avbrytas på något vis.

Det är synnerligen viktigt att dessa två funktioner verkligen gör rätt. Om de missar det allra minsta kommer realtidskärnan att drabbas av mystiska kraschar som är praktiskt taget omöjliga att spåra upp.

I en enkel kärna som Ärkenöt är dessa funktioner de enda som behövs för att skydda kärnans interna variabler.

I fall det är fråga om ett mer komplext system, t.ex med flera processorer som symmetriskt delar på arbetsbördan, kan man inte förlita sig på en så enkel mekanism. I stället måste kärnans olika delar skyddas med hjälp av programmässiga lås, t.ex implementerade med Dekkers algoritm. Den intresserade kan titta i källkoden till Mach MK, ett operativsystem utvecklat vid CMU.

2.4 Kärnan

När man fått ihop korutinerna kan man börja implementera kärnan. Kärnan består av flera delar som samarbetar och i viss mån bygger på varandra. I grunden återfinner man ett listhanteringspaket, som står för hantering av köer och listor av trådar/processer/meddelanden/tidlistor etc. Detta fungerar som fundament för de funktioner som kärnan erbjuder, t.ex processynkronisering, kommunikation och hantering av tid.

```
typedef struct queue_entry {
struct queue_entry *prev;
struct queue_entry *next;
};

typedef struct queue_entry      *queue_t;
typedef struct queue_entry      queue_head_t;
typedef struct queue_entry      queue_chain_t;
typedef struct queue_entry      *queue_entry_t;
```

Figur 2.3: Datastrukturen för köer i Ärkenöt. Den är lånad från Mach MK.

2.4.1 Köer och listor

Den mest fundamentala delen i en kärna är en effektiv köhantering. De olika processer som delar på processorn flyttas hela tiden omkring mellan olika köer allteftersom arbetet fortskrider. Om det är många processer som delar på processorn tillbringar de dessutom mesta tiden med att stå i kö.

För just denna del har jag lånat ett listhanteringspaket från en annan kärna, Mach MK från Carnegie-Mellom University. Detta för att slippa problem med att tänka ut, implementera och sedan alldeles säkert felsöka i det också.

Eftersom utformningen av köerna har en relativt stor återverkan på utformningen av andra delar av en realtidskärna, särskilt processens datastruktur, måste man tänka in köerna i sitt sammanhang innan man väljer utformning.

Till exempel kan man välja att ha köer bestående av köelement, som i sin tur innehåller en pekare på den post som står i kön. Ett annat alternativ, som utnyttjas i Ärkenöt, är att implementera köerna som en del av de element som köar.

I Ärkenöt finns det två (egentligen tre) typer av köer. Dels finns det FIFO-köer av trådar. Dessa fungerar som en vanlig kö.

Dels köer som sorteras efter prioritet, prioritetsköer. En prioritetskö är implementerad som en hel sats FIFO-köer, en för varje prioritet. Antalet prioritetsnivåer, och därmed antalet separata köer i prioritetsköerna, är sexton.

Den tredje typen av kö finns i klockan. Denna kö innehåller `timeout`-element sorterade efter klockslag.

2.4.2 Processer/Trådar

En process är den grundläggande enheten inom vilken programexekvering sker i ett realtidssystem. När kärnan låter processorn byta arbetsuppgift så byter den helt enkelt process. För att kunna hantera flera olika processer måste kärnan ha en processpost som innehåller allt som kärnan behöver veta om processen. I processposten återfinner man till exempel kontextposten, som beskriver processorns tillstånd, uppgifter om prioritet, exekveringstid, signaler, resursutnyttjande etc.

Processpostens utseende beror alltså i hög grad på vilka funktioner man vill att realtidskärnan skall ha. I vissa fall kan det också bli fråga om att lägga till fält i processposten allteftersom man utvecklar kärnan. Som exempel kan nämnas processposten i Ärkenöt nedan.

Ärkenöt har en ganska liten processpost, se figur 2.4. Detta beror på att Ärkenöts processer snarare är vad man kallar för lättviktsprocesser eller *trådar*. Utmärkande för trådar är att de har gemensamt adressområde för globala variabler och programkod, men varsin stack. Detta är anledningen till att en process kallas för `thread` eller tråd i Ärkenöts terminologi.

```
typedef struct thread_rec {
   /* Reserved field for machine dependent code */
   context_rec   context;

   queue_chain_t thread_list_chain;
   boolean_t     terminated;

   /* Fields for thread queueing (by monitors,events etc.) */
   queue_chain_t queue_chain;
   int           base_pri;       /* Threads basic priority level   */
   int           sched_pri;      /* Threads current priority       */
   caddr_t       priority_queue; /* points at the priority queue   */
                                 /* this thread resides in (if any)*/
   boolean_t     blocked;        /* TRUE if thread is blocked      */

   /* Fields for time management */
   int           quantum;      /* remaining ticks before reschedule */
   timer_elt     timeout;
   boolean_t     timeout_set;

   /* Events */
   caddr_t       waiting;      /* event I'm waiting for */
   int           wait_result;

   /* Monitors */
   queue_head_t  monitor_list; /* List of all monitors I have occupied */
   caddr_t       blocked_by;   /* Thread is blocked by this monitor */
} *thread_t;
```

Figur 2.4: Datastrukturen för en tråd (process) i Ärkenöt. Notera fälten av typen `queue_chain_t` som representerar köer som tråden kan befinna sig i.

```
typedef struct timer_elt {
   queue_chain_t  chain;
   timevalue      exp_time;
   void           (*action)(void *);
   void           *action_data;
   boolean_t      private;        /* don't free on expiration */
} timer_elt,*timer_elt_t;

void set_timeout(timer_elt_t telt);
void remove_timeout(timer_elt_t telt);
```

Figur 2.5: Datastrukturen för Ärkenöts timeout.

2.4.3 Hantering av tid

Ingen realtidskärna är komplett utan en korrekt tidmätning. Tidmätning sköts i regel med en hårdvaruklocka, som avbryter processorn med jämna mellanrum.

Programmeringen av denna klocka är i allmänhet mycket varierande mellan olika maskiner. För att undvika maskinberoende kod i själva kärnan anropar Ärkenäts systemklocks uppstartsrutin en funktion `hardclock_init` som sköter om uppstart av hårdvaruklockan samt ansluter avbrottsrutinen till dess avbrott.

Processorn tar hand om avbrottet i en avbrottsrutin som hanterar alla uppgifter som är relaterade till tidmätning. Förutom att räkna upp en räknare, och på så sätt mäta verklig tid, kan det handla om att mäta hur mycket tid en pågående process förbrukat. Det kan vara att ombesörja exekvering av funktioner vid bestämda tidsögonblick eller att väcka processer som begärt väckning vid visst klockslag.

Samtidigt får inte denna avbrottsrutin vara alltför lång. Den exekverar med avstängda avbrott, och under den tiden kan det (beroende på hårdvara) vara omöjligt att svara på andra (viktigare) avbrott.

När jag implementerat denna del av Ärkenöt har jag strävat efter en generell mekanism för att utföra uppgifter vid givna tidpunkter. Den jag har valt att använda mig av är s.k `timeouts`, se figur 2.5

Ett timeout innehåller en tidpunkt, en funktionspekare, samt ett argument som funktionen anropas med. När tidpunkten löper ut anropas funktionen som en del av klockavbrottet. Eftersom funktionen anropas i den avbrutna trådens kontext måste den vara kort. Dessutom bör den inte reservera mer än ett fåtal lokala variabler.

Denna mekanism ger möjlighet att utvidga tidshanteringen till andra metoder för tidshantering. Ett exempel är de speciella händelser som programmeras att inträffa med jämna intervall, se även 4.4.4.

Mätning av arbetsbelastning

Ärkenöt har i sig inbyggt mätning av medelbelastningen på processorn. Denna sköts om i ett samarbete mellan drivrutinerna, systemklockan, och den tråd som är aktiv när ingenting finns att göra. Mätningen av denna medellast förutsätter att det finns en funktion `hardclock_get_value` som ger ett noggrannt värde på tiden sedan senaste klockavbrottet.

Avbrottsrutiner förutsätts samarbeta med kärnans belastningsmätning genom att anropa funktionen `set_cpu_active` som första åtgärd innan något annat görs.

2.4.4 Processynkronisering

Vilka möjligheter för synkronisering av processer som man vill att kärnan skall erbjuda är i mycket en fråga om tycke och smak. I litteraturen och i de operativsystem som redan existerar finns talrika exempel på olika metoder för processynkronisering. Det finns även programmeringsspråk som har inbyggda metoder för beskrivning och synkronisering av parallella arbetsuppgifter (Ada t.ex). Gemensamt för alla alternativ är att det i princip räcker med ett. I vissa fall vill man emellertid av bekvämlighetsskäl och effektivitetsskäl ha tillgång till flera olika.

Vid implementationen av Ärkenöt har jag valt att implementera de synkroniseringsalternativ som vi får lära oss använda vid LTH. Nämligen semaforer, monitorer, händelser och brevlådor.

Monitorerna är försedda med prioritetsärvning. Med detta menas att en tråd som reserverat en monitor får sin prioritet höjd om en annan tråd med högre prioritet väntar på att få tillgång till monitorn.

För att få generellare kod är processorkön (ofta kallad ready-kö) byggd på samma datastruktur som monitorerna. Detta medför att händelserna kan vara fristående, dvs en tråd som väntar på en fristående händelse flyttas direkt till processorkön, istället för till monitorns kö.

Brevlådorna kan överföra vilka data som hels, eftersom brevlådorna arbetar med pekare till de överförda objekten. Brevlådorna kan ha vilken storlek som helst, dock minst ett.

2.4.5 Drivrutiner

Slutligen skall jag ge den andra motivationen för att man skriver en realtidskärna. Det är möjligheten att på ett relativt effektivt sätt isolera applikationsprogrammeraren från hårdvarudetaljer genom att skriva återanvändbara drivrutiner för hårdvaran i datorsystemet. Detta innefattar allt från bitmanipulering till att skriva avbrottshanterare som gör vad som skall göras när det kommer ett avbrott från en viss enhet. Tack vare kärnans möjligheter till dynamisk omfördelning av processortid är det dessutom lätt att utnyttja väntetider vid långsamma yttre enheter till att utföra effektivt arbete i någon annan process under tiden.

Dessa funktioner som isolerar hårdvara brukar med ett gemensamt namn kallas för drivrutiner. Beroende på vilken ambitionsnivå man har, kan man välja att ha specialiserade rutiner, med funktioner som passar till hårdvaran, eller så kan man ha ett gemensamt gränssnitt till samtliga, och på det viset få en prydlig generalitet som underlättar flyttning till annan hårdvara. Så ambitiös har jag inte varit med Ärkenöt, utan jag har nöjt mig med att utveckla de drivrutiner som har varit nödvändiga för att kunna använda den hårdvara som funnits i "mitt" system.

Kapitel 3

Hårdvara och kommunikation

Detta kapitel handlar om målsystemets hårdvara. Kapitlet beskriver hårdvaran mer detaljerat än kapitel 1. Det beskriver också en del av de funktioner som hanterar hårdvaran. Sepciellt avseende fästs vid egenskaper som som endast fått summarisk behandling i den medföljande dokumentationen.

3.1 DSP2300

Detta avsnitt handlar om instickskortet NB-DSP2300. Målet är främst att beskriva programmässiga problem som berör kortets hårdvara. Hårdvarans uppbyggnad diskuteras endast i övergripande ordalag. Den intresserade läsaren rekommenderas att ha instruktionsboken tillgänglig som bredvidläsning.

3.1.1 DSP2300, allmänt

Instickskortet NBDSP2300 har en hel del hårdvara utöver signalprocessorn TMS-320C30. Denna hårdvara inkluderar en DMA-processor[1], gränssnittslogik mot huvudbussen, snabbt statiskt RAM-minne och dubbelportat minne för in- och utmatning av data. Delar av denna hårdvara kontrolleras via ett kontroll- och statusregister, medan andra delar (d.v.s minnena) inte behöver kontrolleras alls.

Till grund för flera av de följande funktionerna ligger funktioner som direkt hanterar NB-DSP2300:s kontroll/status-register. Flera av dessa är C-makron, inte funktioner. Dokumentet `dsp2300_regs.h` innehåller definitioner av makron och funktioner. Dokumentet `dsp2300_regs.c` definierar de pekare som refereras av makrofunktionerna samt innehåller implementationer av de funktioner som inte är makron.

3.1.2 DMA, på kortet

På instickskortet NB-DSP2300 finns det en DMA-processor. Den finns beskriven i manualen till NB-DSP2300, avdelning 3-20 – 3-22, men jag återger huvuddragen här.

DMA-processorn kontrolleras via bitar i kontrollregistret, samt ytterligare tre register. Registren innehåller:

[1] DMA står för Direct Memory Access. Med detta avses att en yttre enhet direkt kan överföra data till och från internminnet utan processorns hjälp. En DMA-processor är en enhet som självständigt kan flytta data mellan minnet och en extern enhet eller något annat minne.

1. En lokal adress (d.v.s som hör hemma på kortet).
2. En adress som pekar ut på huvudbussen.
3. En räknare som anger hur många ord som det är kvar att förflytta.

Bitarna i kontrollregistret styr datariktningen, till eller från kortet, samt adressberäkningen för huvudbussen, upp/ned eller fast.

Kortets DMA-processor kan alltså bara flytta data mellan NB-DSP2300:s minne och ett annat instickskort eller någon annan enhet som är åtkomlig via huvudbussen. Man bör också lägga märke till att kortets DMA-processor har en annan adressrymd än signalprocessorn.

När DMA-processorn är klar med en dataöverföring genererar den ett avbrott till signalprocessorn. Detta avbrott kvitteras genom att skriva till DMA-processorns räknarregister.

I detta sammanhang vill jag också nämna en egenhet med detta avbrott (och avbrottet från RTSI-bussen). Avbrottet är nivåtriggat vilket i detta fall innebär att avbrottet inte deaktiveras automatiskt. Avbrottet förblir aktivt tills signalprocessorn kvitterat på angivet sätt. Detta medför att avbrottet tas två gånger. Den andra gången har avbrottet förhoppningsvis kvitterats vilket kan konstateras vid en undersökning av signalprocessorns avbrottsflaggor. I detta fall behöver man inte genomlöpa avbrottsrutinen.

Denna hantering sköts om automatiskt av den programmässiga avbrottshanteringen som finns i dokumentet `interrupt_handler.asm`. Detta för att man skall slippa tänka på denna egenhet mer än en gång.

3.1.3 DMA, på chipet

Signalprocessorn har en egen DMA-processor, som kan agera helt på egen hand. Den har samma adressrymd som signalprocessorn, kan svara på avbrott istället för signalprocessorn och givetvis generera avbrott till signalprocessorn när den är klar med en transaktion. Den finns utförligt beskriven i instruktionsboken till signalprocessorn. Denna DMA-processor är helt generell med avseende på datariktning och adressberäkningar.

3.1.4 Klockan

TMS320c3x har två klockor/räknare inbyggda. Dessa kan användas för diverse ändamål, se även instruktionsboken. För att kunna hantera dem behändigt har jag skrivit funktioner för programmering av dem. Dessa funktioner finns definierade i dokumentet `tms_timers.h`. En av klockorna (noll) utnyttjas av Ärkenöt som realtidsklocka, den andra utnyttjas av övervakningsprogramvarans avlusningsfunktioner för tidtagning mellan brytpunkter.

3.1.5 RTSI-bussen

National Instruments har försett sina instickskort i NB-serien med något som de kallar för en "Real-Time System Integration Bus", förkortat RTSI. Denna buss består av ett flertal programmerbara triggsignaler, avbrottsledningar och (två?) seriekanaler. Jag skall bara beröra avbrottshanteringen eftersom jag hittills inte använt bussens övriga funktioner.

RTSI-bussens avbrottshantering erbjuder möjligheten för ett annat instickskort att generera avbrott till ett (flera) NB-DSP230X-kort. Avbrottets källa identifieras genom att läsa statusregistret. Man kan då få fram vilken kortplats som genererade

avbrottet. Rutiner som sköter om detta finns i dokumentet `rtsi_interrupt.c` med motsvarande gränssnitt i dokumentet `rtsi_interrupt.h`.

RTSI-avbrottet har samma egenhet (nivåtriggat) som beskrivits i avsnittet om Kortets DMA-processor (3.1.2). Dessutom finns det här en anledning att se upp. Det är nämligen så att NB-DSP2300 själv kan generera RTSI-avbrott på valfri kanal, genom att programmera kontrollregistret.

Ett självgenererat avbrott skiljer sig inte från andra, det kommer givetvis till signalprocessorn också. Man måste alltså hindra signalprocessorn från att ta RTSI-avbrottet om man skall generera ett sådant. Omvänt måste man givetvis programmera kontrollregistret så att kortet inte genererar något RTSI-avbrott innan man tillåter signalprocessorn att ta RTSI-avbrott. Om man glömmer något av detta kommer signalprocessorn att fastna i en evig slinga av avbrott.

Vid uppstart av NB+DSP2300 programmeras kontrollregistret så att RTSI-avbrottet genereras. Innan man tillåter signalprocessorn att ta detta avbrott måste man alltså stänga av det genom att programmera om kontrollregistret. Om detta finns det ingenting nämnt i dokumentationen, vilket kostade oss många timmars felsökning när vi höll på med projektet våren 1992.

3.1.6 Minnet

På NB-DSP2300 finns det 256kByte statiskt RAM, organiserat i 32-bitars ord. Detta minne kan samtidigt adresseras av både signalprocessorn och DMA-processorn eller huvudbussen. Detta gör att en tillämpning som körs på värddatorn kan komma åt signalprocessorns minne utan att signalprocessorn förlorar prestanda.

Dessutom finns det 8kByte RAM på signalprocessorns In/Ut-buss. Detta minne kan adresseras samtidigt av DMA-processorn och signalprocessorn. När man använder DMA för att överföra data mellan detta minne och t.ex värddatorns primärminne kan man dessutom göra en *partiell* omvandling mellan det flyttalsformat som signalprocessorn använder och standardformatet IEEE 754. Denna omvandling styrs av en bit i kontrollregistret.

Förutom de externa minnesareor som finns på kortet har också signalprocessorn 8kByte internminne, plus fickminnen för instruktioner och data. Signalprocessorns internminne är så snabbt att signalprocessorn kan utföra två minnesåtkomster per instruktionscykel i det.

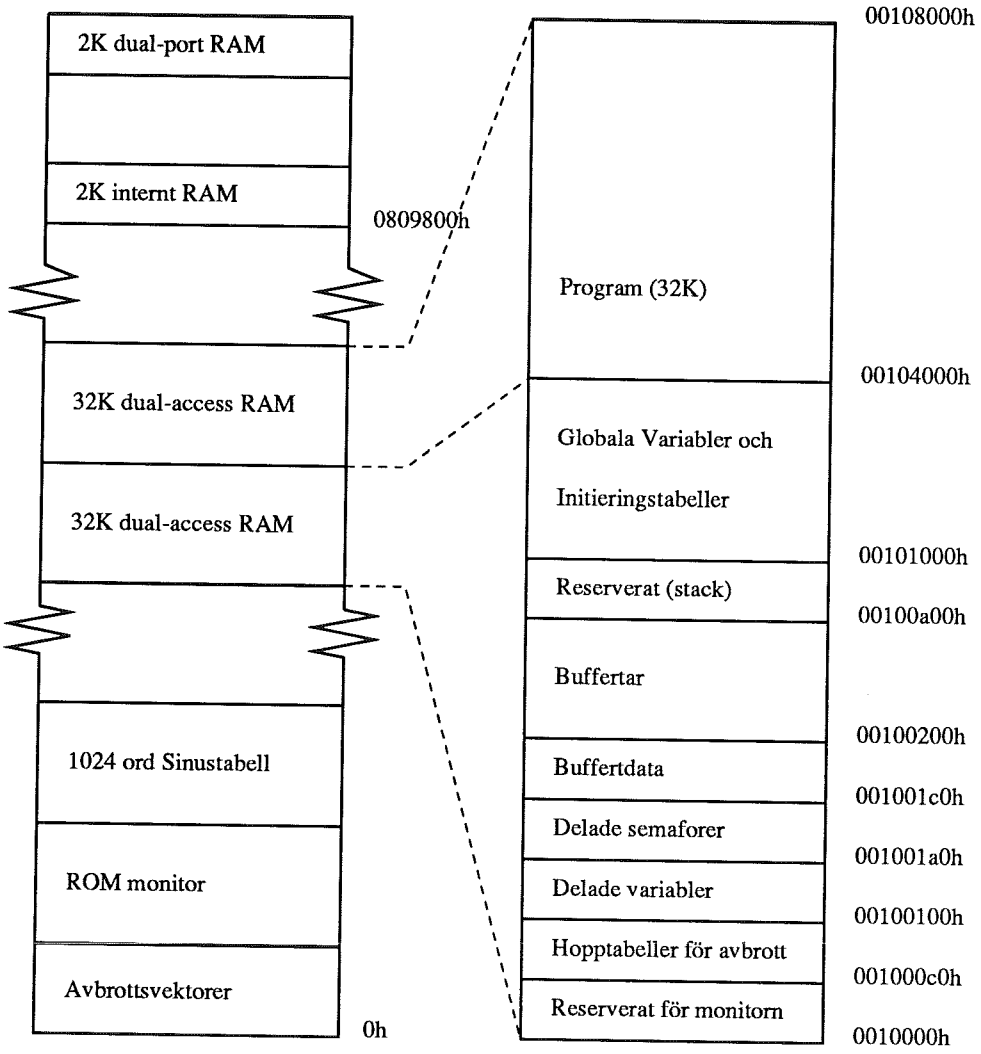
3.1.7 Gränssnitt mot huvudbussen

Gränssnittet mot värddatorns huvudbuss (NuBus) är komplett. Detta betyder att instickskortet kan agera både herre (eng. master) och slav på bussen.

Busslåsning

Signalprocessorn har en egenhet som leder till problem ur realtidssynpunkt. Problemet uppstår när signalprocessorn genomför en busstransaktion samtidigt som en annan bussherre gör en minnesåtkomst på NB-DSP2300. Eftersom signalprocessorn inte kan avbryta en påbörjad busstransaktion hamnar bussen i ett dödläge som inte upplöses förrän Busslogiken genererar ett bussfel. Detta tar avsevärd tid (ca 100ms), vilket är fullständigt oacceptabelt när man använder signalprocessorn för att reglera motorer eftersom sampelintervallet då är i storleksordningen 1ms. Dessutom tappas signalprocessorns busstransaktion bort, vilken förr eller senare, oftast förr, kan få signalprocessorn att stanna helt.

Lösningen på problemet är att låta NB-DSP2300:s busslogik reservera bussen vid varje transaktion som initieras från Kortet. Se instruktionsboken om biten "`LOCIT*`". Detta medför visserligen att varje transaktion tar lite längre tid (ca



Figur 3.1: Minneskarta för NB-DSP2300 med Ärkenöt

0,3 μs) men resultatet blir att man helt slipper de slumpvisa totalstopp som annars hade blivit resultatet.

Eftersom kortets DMA-processor inte har detta problem är det intressant att kunna aktivera bussreserveringen när signalprocessorn skall gå ut på bussen och sedan stänga av igen när det är färdigt.

Eftersom ett program kan bestå av flera parallella, oberoende, processer (trådar) som var och en går ut på bussen, vill man hålla reda på hur många gånger man aktiverat bussreserveringen och inte stänga av den förrän alla processer som startat bussreserveringen också avslutat den. Följande funktioner, definierade i dokumentet `dsp2300_regs.c`, sköter om detta.

`void activate_NuBus_locking()`
: Ett anrop till funktionen `activate_NuBus_locking` medför att alla transaktioner som utförs av någon enhet på DSP2300 spärrar alla andra aktiviteter på bussen.

`void deactivate_NuBUs_locking()`
: Funktionen `deactivate_NuBUs_locking` skall anropas när man inte behöver reservera bussen längre. Om man glömmer detta i sin drivrutin kommer värddatorn att hänga upp sig så fort man startar signalprocessorn.

Adressgenerering

Signalprocessorn TMS320c3x kan bara generera 24-bitars adresser och därför har man blivit tvungen att komplettera signalprocessorns gränssnitt mot huvudbussen med extra basregister, fyra till antalet. Dessa basregister förser signalprocessorn med 4MByte stora fönster ut i huvudbussens adressrymd. Detta är möjligt eftersom TMS320c3x bara adresserar 32-bitars ord, jämför med 680x0 och huvudbussen (NuBus) som kan adressera en byte (oktett) i taget.

Programmeringen av dessa register sköts av en drivrutin som håller reda på vilka register som utnyttjats och spärrar för överutnyttjande. Drivrutinen definierar följande funktioner i dokumentet `drivers:nb_page.h`.

`int set_nubus_page_reg(void **NuBus_address)`
: Funktionen `set_nubus_page_reg` räknar om en 32-bitars NuBus-adress, pekad på av argumentet `NuBus_address` till en 24-bitars adress som kan användas av signalprocessorn, samt reserverar ett sidregister till denna adress. Sidregistrets nummer returneras. Typisk användning är att räkna om basadressen till ett I/O-kort varefter kan man hantera I/O-kortet som en `struct` i C.

`void free_nubus_page_reg(int index)`
: Funktionen `free_nubus_page_reg` ledigmarkerar det register som anges av `index`.

Avbrottshantering

NB-DSP2300 har en funktion som ger möjlighet för andra enheter anslutna till huvudbussen att generera avbrott till signalprocessorn. Detta sker genom att skriva till en adress inom en viss del av kortets adressrymd, avbrottsminnet. Detta avbrott utnyttjas av övervakningsprogrammet till att starta och stoppa program, till brytpunktsåterstart samt för att kunna läsa och skriva i minne som normalt inte kan kommas åt utifrån.

Slutligen kan NB-DSP2300 själv generera avbrott via huvudbussen, avsedda för värddatorns drivrutiner.

3.1.8 Fasta styrprogram

På NB-DSP2300 finns det ett ROM som innehåller ett övervakningsprogram (eng. monitor). Detta program exekveras vid uppstart av kortet och sköter då om all initialisering av kortets hårdvara till ett känt tillstånd. När detta är gjort placeras signalprocessorn i en vänteslinga. Vänteslingan gör inget annat än blinkar med de tre lysdioder som finns monterade på kortet. Denna funktion används även för diagnosticering av kortet. När kortet är helt, blinkar alla tre dioderna i fas. Om kortet är trasigt blinkar de i någon annan sekvens, som anger vad som är fel.

Som nämnts i avsnittet om bussgränssnittet innehåller övervakningsprogrammet funktioner för att kunna avlusa program och för att komma åt minnesceller som inte är direkt åtkomliga från huvudbussen. Dessa funktioner utnyttjas av tillämpningen `DSP Debug`, men de kan också utnyttjas av andra tillämpningar. Tyvärr kommer dessa funktioner i konflikt med realtidskärnan.

För att undgå dessa har Ärkenöt en egen avbrottshanterare för avbrott genererade genom skrivning i avbrottsminnet. Denna avbrottshanterare gör samma sak som övervakaren i fallet med minnesåtkomster. Den kan också stoppa all processoraktivitet och återvända till övervakaren. Dessutom kan man registrera ytterligare funktioner som anropas vid en viss funktionskod. I framtiden kan man här inkludera en komplett avlusare och på så sätt helt överta övervakarens funktioner.

Ärkenöts avbrottshanterare för avbrott från huvudbussen, har följande funktion för att registrera en funktion som avbrottshanterare tillsammans med en funktionskod. Dokumentet `nubusm_interrupt.h` definierar den.

`void at_nubusm_int(long (*f)(),int opcode)`
: Funktionen `at_nubusm_int` registrerar funktionen `f` tillsammans med funktionskoden `opcode`. De funktionskoder som är upptagna anges av tabell 5.1.

3.2 Värddatorkommunikation

Detta avsnitt handlar om implementationen av funktionerna för överföring av data mellan NB-DSP2300 och värddatorn. Samtliga kommunikationskanaler utnyttjar delat minne för mellanlagring av data. Den fysiska överföringen av data sköts av huvudbussen, som alltså motsvarar det fysiska skiktet. Transportskiktet sköts av funktioner, både på signalprocessorsidan och på värddatorsidan.

3.2.1 Delade data

Delade data är en kommunikationskanal som erbjuder 64 variabler som kan läsas av värddatorn och skrivas av signalprocessorn, notera att det omvända inte gäller, samt 64 variabler som kan läsas av signalprocessorn och skrivas av värddatorn.

I gränssnittet refereras variablerna med index 0–63, oberoende av vilken riktning som data överförs i. En variabel som skrivs med index 14 av signalprocessorn läses alltså med index 14 av värddatorn och vice versa. Detta gör att man kan undvika problem som kunnat uppstå om index hade varit olika för värddatorn respektive signalprocessorn.

På signalprocessorsidan består gränssnittet av två C-makron, ett för läsning och ett för skrivning. Detta gör att man kan överföra valfria enkla datatyper utan att använda typomvandlingar (eng. cast). Som extrabonus blir det dessutom effektivare då man slipper utföra ett funktionsanrop. Makrofunktionerna kontrollerar också indexet så att man inte kan gå utanför gränsen.

På värddatorsidan har jag hittills bara implementerat gränssnitt för LabVIEW. Detta är i form av fyra "Code Interface Nodes" som sköter om läsning respektive skrivning. Dessa funktioner hanterar hela block av variabler, d.v.s. de läser/skriver

ett givet antal variabler med början vid ett givet startindex. Se även avsnitt 5.2. Där beskrivs virtuella instrument som isolerar kodgränssnittsnoderna från användaren.

3.2.2 Delade semaforer

Delade semaforer är skyddade variabler. Med detta avses att endast en åt gången av signalprocessorn och värddatorn kan komma åt dess värde. För att illustrera hur de är uppbyggda ger jag ett kodexempel, hämtat från signalprocessorsidan.

```
#define kSharedVarBase  0x0100180
#define kNrOfSharedVars 8
#define kPID            0
#define kOtherPID       1

typedef struct __shared_variable {
   long    flag[2];
   long    turn;
   long    data;
} __shared_variable;

__shared_variable *shared_variable = (__shared_variable *) kSharedVarBase;

/*
 * Write: writes value into protected variable
 */
void shared_variable_write(int index,register long value)
{
   /* avoid range errors! */
   if ((index < 0) || (index > kNrOfSharedVars)) return;

   /* Critical Section Entry */
   shared_variable[index].flag[kPID] = 1;
   shared_variable[index].turn = kOtherPID;

   while (shared_variable[index].flag[kOtherPID] &&
          (shared_variable[index].turn == kOtherPID)) {} ;

   /* Critical Section */
   shared_variable[index].data = value;

   /* Critical Section Exit */
   shared_variable[index].flag[kPID] = 0;
}
```

Notera den tomma satsen i while-slingan. De övriga funktionerna ser praktiskt taget likadana ut. Texten ovan består av klipp ur dokumenten `shared_variables.h` och `shared_variables.c`.

3.2.3 Buffertar

Som avslutning finns det också åtta buffrade överföringskanaler. Kanalerna har ingen bestämd datariktning, utan data kan både hämtas och lämnas i buffertarna av både signalprocessorn och värddatorn.

Buffertarna är förlagda i ett kontinuerligt block i minnet, se figur 3.1. All information som behövs finns i poster som ligger för sig själva, på bestämda adresser. De data som finns i posterna skyddas från samtidig modifikation på samma sätt som de delade semaforerna. Nedan ges datastrukturen för en post:

```
typedef struct __buffer_info_rec {
   long   flag[2];
   long   turn;
   long   magic_cookie;
   long   size,count;
   long   overrun_count;
   long   last_written,last_read;
   long   *buffer;
} __buffer_info_rec;
```

Fältet `magic_cookie` talar om ifall datastrukturen och bufferten den kontrollerar överhuvudtaget är giltig.

Funktionerna som implementerar buffertarna på signalprocessorsidan är oberoende av realtidskärnan. Om t.ex `shared_buffers_get` anropas på en tom buffert så hamnar signalprocessorn i en vänteslinga. Detta är inte speciellt lyckat. Det vore önskvärt om värddatorns kommunikationsmetoder orsakade ett avbrott när en buffert har rörts av värddatorn. På det viset kunde man låta signalprocessorn göra något annat medan en tråd väntar på att en buffert skall bli tillgänglig.

Nu är detta inte något stort problem. En tråd som kommunicerar med värddatorn via buffertarna har normalt så låg prioritet att den inte kan påverka resten av programmet genom att hamna i en vänteslinga.

3.3 NB-MIO-16

Drivrutinerna till instickskorten NB-MIO-16-H/L-X bygger på en gemensam datastruktur. Denna beskriver kortets uppbyggnad och innehåller allt som drivrutinerna behöver veta.

Speciellt är det definierat en datastruktur som beskriver själva kortet. I drivrutinens datastruktur finns en pekare till en struktur som beskriver kortet. Pekaren sätts upp med hjälp av drivrutinen för sidregistren i samband med initialiseringen av drivrutinen.

Detta har gjort det möjligt att sköta adressberäkningen till kortet implicit med referenser till element i kortets datastruktur.

3.3.1 Analoga ingångar

Kortets analoga ingångar kontrolleras av funktionen `mio_16_analog_in`. Huvuddragen i denna funktions arbete är följande:

1. Programmera kortets kanalväljare
2. Om mer än en kanal skall samplas, programmera räknarkretsen att sköta om sampling av rätt antal kanaler och rätt sampelintervall.
3. Starta datainsamlingen.
4. Invänta avbrott från kortet, som talar om att datainsamlingen är slutförd. Detta utföres med en semafor, som blockerar den anropande tråden. Avbrottsrutinen som svarar på avbrottet signallerar sedan med denna semafor när allting är klart. Detta gör att processorn kan syssla med annat medan A/D-omvandlingen pågår.
5. Hämta data och omvandla dem från kortets heltalsformat till flyttal. Detta steg sköts om av avbrottsrutinen, på grund av den speciella karaktär som avbrottet från kortet har.

6. Återvänd till anroparen.

Steg ett till tre ovan är rättframt, det är bara att följa instruktionsboken. En detalj som dock förtjänar att nämnas är hur programmeringen av räknarkretsen är optimerad för att slösa så lite tid som möjligt.

Det visade sig, när jag testade den här funktionen, att alla åtkomster till register i räknarkretsen tar relativt lång tid, ungefär två mikrosekunder per styck! Därför programmeras sampelräknaren om endast då antalet kanaler som skall samplas är annorlunda än vid förra anropet, samt första gången. Dessutom visar det sig att man inte behöver bry sig om det sista steget, att återställa räknarkretsen, vilket sparar nio minnesåtkomster.

Just denna del är rikligt kommenterad i programkoden, men den är också fylld av villkor för kompilering, vilket gör den en smula svåröverskådlig.

Kortet levererar ett heltal mellan $-2048 * 2^{16}$ och $2047 * 2^{16}$ som motsvarar -10 respektive 9.995 Volt. För att få ett flyttal divideras det inlästa heltalet med talet $2^{27} = 134217728.0$, vilket ger ett flyttal mellan -1 och 1.

Avbrottsrutinen innehåller också några anpassningar för att allt skall fungera, främst för att det inte skall gå för fort för den relativt långsamma elektroniken på kortet.

3.3.2 Analoga utgångar

De analoga utgångarna sköts av funktionen `mio_16_analog_out`. Funktionen tar ett flyttal, omvandlar det till ett heltal som förstås av D/A-omvandlaren, skriver heltalet till den valda D/A omvandlaren, samt returnerar det värde som matades ut *efter begränsning och kvantisering.* Begränsningen görs för att undvika att D/A-omvandlaren "slår runt".

3.4 AO-6

Drivrutinen som hanterar kortet NB-AO-6, är uppbyggd på samma sätt som drivrutinen för NB-MIO-16-H/L-X, men denna är avsevärt mycket enklare.

Det enda som är speciellt med kortet är adresseringen av de sex analoga utgångarna. Kortets adressavkodning är gjord så att adressbitarna noll till fem (som det ses från signalprocessorn) adresserar varsitt kanalregister. Om man dessutom ettställer adressbit sex uppdateras de analoga utgångarna till det värde som anges av kanalregistren. Detta har fått lysa igenom i rutinen.

De analoga utgångarna sköts av funktionen `ao_6_analog_out` som är snarlik funktionen `mio_16_analog_out`, enda skillnaden är i adresseringen av de olika kanalerna. Denna sköts av ett antal makrodefinitioner som definierar bitar för att adressera var och en av de sex D/A-omvandlarna.

Kapitel 4

Programmering under Ärkenöt

Detta kapitel har som mål att beskriva hur man utvecklar program som skall fungera under Ärkenöt på det använda målsystemet. Kapitlet inleds med ett minimalt program, som skriver ut texten `"Hej, världen"` *i ett terminalfönster. Med detta program som grund beskrivs utvecklingsgången med editering, kompilering, länkning och till sist hur man kör programmet.*

I de därpå följande avsnitten introduceras Ärkenöts hjälpfunktioner för uppbyggnad av flertrådiga program. Ärkenöt erbjuder ett flertal alternativ för synkronisering och kommunikation mellan olika trådar. De synkroniserings och kommunikationsalternativ som finns är Semaforer, Händelser, Monitorer och Brevlådor. Vart och ett av dessa objekt har fått ett eget avsnitt.

4.1 Minsta möjliga program

För att skriva enkla program som inte behöver utnyttja Ärkenöts möjligheter till parallella processer behöver man inte vidta några speciella åtgärder. Det går alldeles utmärkt att skriva sina program alldeles utan att fundera på denna möjlighet. Det enda man i detta fall är intresserad av är hur man sköter in- och utmatning av data.

Den nuvarande versionen av Ärkenöt innehåller inte någon komplett implementation av C:s standardbibliotek för in- och utmatning. Detta är tråkigt, men vem behöver t.ex ett filsystem när värddatorn har ett alldeles utmärkt sådant?

Vad som finns av standardbibliotekets funktioner är de viktigaste funktionerna för textutskrift. nämligen `printf()`, `sprintf()`, `puts` och `putchar()`. Detta medför att man kan köra C-program som innehåller utskrifter av formatterad text till stdout, men inga inmatningar. Det medför också att program som fungerar under Ärkenöt och inte utnyttjar funktioner som är speciella för Ärkenöt, även fungerar i mera kompletta C-miljöer.

För att visa vad jag menar, ger jag som exempel det program som skriver ut texten `Hej, världen` på Macintosh-skärmen:

```
#include<libc:stdio:stdio.h>

void main() {

   printf("Hej, världen\r");

   exit(0);
}
```

Detta program skiljer sig endast på en punkt från motsvarande program för t.ex. ett UNIX-system. Apple har valt att använda tecknet Vagnretur (Carriage return, CR, `\r`, ascii(13)) för att ange ny rad. Till skillnad från UNIX som använder Radframmatning (Linefeed, LF, `\n` ascii(10)) för motsvarande uppgift.

Ärkenöt startar automatiskt. Man behöver alltså inte starta kärnan genom att anropa någon funktion i stil med `init_kernel()`. Detta innebär att funktionen `main()` i programmet anropas som startpunkt för en "tråd", programmets huvudtråd.

Funktionen `exit()` resulterar i att Ärkenöt avslutar all verksamhet och låter processorn återvända till sitt väntetillstånd.

För att köra detta program på den signalprocessor som jag haft som målsystem måste man först skriva in programtexten med hjälp av MPW Shell. Hur det går till beskrivs i punktform nedan. Jag förutsätter att datorn redan är startad och att MPW Shell också är startat.

1. Skapa ett nytt dokument genom att välja `New` från menyn `File`. MPW frågar efter ett dokumentnamn, välj t.ex. `hello_world.c`. C-program har dokumentnamn som slutar på `.c`. Detta är för att kompilatorn skall känna igen dem.

2. Skriv in programtexten.

3. Spara dokumentet. Detta gör du genom att välja `Save` från menyn `File`.

Sedan måste programmet kompileras. Enklaste sättet att åstadkomma detta är att kopiera dokumentet `Makefile.default` till samma mapp som innehåller dokumentet `hello_world.c`. Dokumentet `Makefile.default` finns i mappen `Libraries` som i sin tur finns i mappen `Kernel`. Kopian döper man sedan om till `hello_world.make`. Därefter editerar man kopians första rader så att de får följande utseende:

```
#
# Makefile.default
#
# Default makefile for programs under the Ärkenöt real-time kernel.
#
Sources = hello_world.c
Objects = hello_world.o
TheTarget = hello_world

#
# Don't change anything below this line
#
    .
    .
    .
```

När detta är gjort har du fått en "Makefile" som sköter om alla steg som behöver utföras för att få fram ett exekverbart program. Välj sedan `Build` från menyn med samma namn. MPW frågar då efter programnamnet. Skriv in `"hello_world"` i dialogrutan och klicka på `"OK"`. MPW svarar med att starta processen med att "bygga" ett exekverbart program. Det är bara att luta sig tillbaka, processen sköts automatiskt. Eventuella fel rapporteras på MPW:s arbetsblad. Om det blev fel vid kompileringen måste du rätta dem och sedan välja `Build`.

När programmet är felfritt ur kompilatorns synpunkt kommer processen att avslutas normalt och man har fått ett färdigt program som kan köras på signalprocessorn.

När programmet kompilerats utan felmeddelanden kan man provköra det. Till detta används programmet **DSP Debug** som är den avlusare som National Instruments levererar till sina signalprocessorkort. Dessutom behövs programmet **DSP Console** för att kunna visa programmets utskrift på skärmen. För att köra programmet kan du följa stegen nedan.

1. Starta **DSP Debug** genom att dubbelklicka på dess symbol.
2. Välj **Download program** från menyn **File**. Leta fram det körbara programmet i dialogrutan.
3. Starta **DSP Console**.
4. Aktivera **DSP Debug** igen genom att klicka i dess fönster.
5. Starta signalprocessorn genom att välja **Launch** från menyn **Execution**.
6. Signalprocessorn stoppar automatiskt. Om allt har gått rätt så finns texten **Hej, världen** i **DSP Console**:s fönster.

Jag har nu gått igenom utvecklingsprocessen för program under Ärkenöt på det system som jag använt. Jag har använt ett mycket enkelt exempel, men gången är likadan även för större program. Den "makefile" som du skapade kan t.ex. utan vidare byggas ut till flera källkodsdokument.

Det finns ett sätt till att köra program på signalprocessorn. Man kan använda LabVIEW-instrumentet DSP Console som beskrivs närmare i kapitel 5.

När man utvecklar program som inte skall stanna efter någon bestämd tid, utan är gjorda för kontinuerlig drift, vill man kunna avbryta all verksamhet tvångsmässigt. Detta för att kunna ladda ned nya versioner. DSP Debug har tyvärr problem med att inse att signalprocessorn arbetar när man använder Ärkenöt vilket medför att stoppskylten blir verkningslös. För att stoppa signalprocessorn får man istället trycka på startknappen eller välja **Launch** en gång till.

4.2 Hantering av tid <kern:time.h>

Givetvis måste en realtidskärna kunna hantera reell tid om den skall göra skäl för namnet. Ärkenöt mäter tid med noggrannheten en mikrosekund (10^{-6}s) dvs. man kan ange tider med denna noggrannhet. Systemklockans upplösning är något annat. För närvarande är upplösningen 250 mikrosekunder. Detta betyder att kärnan förmår skilja på två tider om de ligger minst 250 mikrosekunder ifrån varandra.

Ärkenöts grundläggande funktioner för hantering av tid finns samlade i dokumentet **time.h**. Dessa skiljer sig inte nämnvärt från motsvarande funktioner i ett Unix-system. I dokumentet definieras datatypen **timevalue** enligt nedan. Dessutom definieras ett antal funktioner för operationer på typen timevalue. Dessa använder sig av pekartypen **timevalue_t**.

```
typedef struct _timeval {
   long tv_sec,
        tv_usec;
} timevalue,*timevalue_t;
```

Funktionerna som definieras av **sysclock.h** är följande:

int timercomp(timevalue_t tvp, timevalue_t tvq, op)
Värdet av makrofunktionen **timercomp** är skilt från noll om villkoret **(*tvp 'op' *tvq)** är uppfyllt. Observera dock att **timercomp** inte ger korrekt resultat om **op** är något av <= respektive >=.

`void timeradd(timevalue_t tvp, timevalue_t tvq)`
: `timeradd` adderar tidsvärdet som `tvq` pekar på till det tidsvärde `tvp` pekar på. tidsvärdet som `tvq` pekar på lämnas oförändrat.

`int timerisset(timevalue_t tvp)`
: Värdet av funktionen `timerisset` är skilt från noll om något av fälten i tidsvärdet som `tvp` pekar på är skilt från noll.

`void timerclear(timevalue_t tvp)`
: `timerclear` nollställer fälten i tidsvärdet som `tvp` pekar på.

`void settimeofday(timevalue_t tv)`
: `settimeofday` sätter kärnans tid till värdet som `tv` pekar på.

`void gettimeofday(timevalue_t tv)`
: `gettimeofday` kopierar kärnans tid till posten som `tv` pekar på.

`clock_t clock()`
: Värdet av funktionen `clock` är antalet mikrosekunder sedan systemet startade. Observera dock att ordlängden 32 bitar gör att detta värde börjar om från noll efter ca 4000 sekunder (1 tim och 11 min).

`time_t time(time_t *tm)`
: Värdet av funktionen `time` är antalet sekunder sedan systemklockan startade. Detta gäller dock bara om man inte ställer om klockan med `settimeofday`. Jämför även med motsvarande definition av `time` i [1].

Exempel

Följande exempel visar hur man kan ställa fram systemklockan med en timme:

```
#include <kern:time.h>

{
   timevalue the_time,one_hour;

   one_hour.tv_sec = 3600;   /* 3600 sekunder på en timme */
   one_hour.tv_usec = 0;

   gettimeofday(&the_time);  /* ta reda på systemklockans värde */

   timeradd(&the_time,&one_hour); /* lägg till en timme */

   settimeofday(&the_time);  /* ställ om systemklockan */
}
```

Nästa exempel är att ta reda på hur lång tid en funktion tar att utföra:

```
#include <kern:time.h>

{
   clock_t starttime,total_time;

   starttime = clock();      /* ta reda på klockan */
   foo_bar(.....);           /* exekvera funktionen */
   total_time = clock() - starttime; /* beräkna exekveringstiden */

   printf("foo_bar tog %d mikrosekunder att utföra",total_time);
}
```

4.3 Flertrådiga program <kern:thread.h>

En av anledningarna till att ha en realtidskärna är att man vill kunna köra program bestående av flera relativt fristående deluppgifter som eventuellt skall exekveras med jämna, men inte nödvändigtvis samma, tidsintervall, utan att själv behöva sköta om arbetsfördelningen mellan de olika uppgifterna. Givetvis finns dessa funktioner i Ärkenöt.

En deluppgift kallas i Ärkenöts terminologi för en tråd. Ärkenöt väljer att följa olika trådar, allteftersom de behöver uppmärksamhet. Ärkenöts trådar har fixa prioriteter, i området 0 till 15 där en tråd med prioritet 0 har högst prioritet. Den tråd som startar med funktionen main har prioritet 12. Prioriteterna 0 och 15 används för Ärkenöts interna trådar.

Vad gäller prioriteringar av olika trådar finns det många varianter, men som rekommendation kan man ange att ju oftare och ju kortare intervall som tråden skall exekveras, desto högre prioritet kan man ge tråden. Trådar som skall exekvera synkront på ett eller annat sätt bör ha lika prioritet. En tråd som aldrig väntar på något, utan kan exekvera på så fort den kan få tillgång till processorn bör ha lägre prioritet än trådar som interagerar med omgivningen.

Följande funktioner finns definierade i dokumentet "`thread.h`"

`thread_t thread_create(void (*entry_point)(),int priority)`
: `thread_create` sköter reservering av minnesutrymme för stacken och trådposten samt initialiserar stacken, kontextposten och trådposten. Funktionens värde är en referens till tråden (För närvarande en pekare till trådposten, men det kan ändras). Vid returen är tråden vilande, d.v.s den får inte tillgång till processorn förrän man låter kärnan ta upp tråden. `entry_point` är en pekare till den funktion som skall anropas när kärnan tar upp tråden. `priority` är den prioritet som tråden skall ha.

`void thread_exit()`
: `thread_exit` används för att avsluta en tråd utan att avsluta hela applikationen. Funktionen återvänder inte.

`void thread_resume(thread_t th)`
: `thread_resume` får Ärkenöt att ta upp en vilande tråd. För närvarande används den bara för att påbörja en nyss skapad tråd.

Exempel

Följande exempel skapar och startar en tråd som exekverar parallellt med ursprungsprogrammet. Den nyskapade tråden avslutar sedan sig själv.

```
#include <kern:thread.h>

void new_thread() {
   .
   .
   /* Gör någonting */
   .
   .
   thread_exit(); /* Avsluta denna tråd */
}

void main() {
   thread_t new;
   .
   /* Gör något. hittills är detta den enda tråden i systemet */
```

```
        .
        .
        /* Skapa en ny tråd med prioritet 11 */
        new = thread_create(new_thread,11);

        /* Ännu har ingenting hänt med den nya tråden mer än
* att den skapats. Vi måste se till att dess funktion
* (new_thread) ovan anropas */
        thread_resume(new);

        /* Nu kan denna tråd och den nya dela på datorkapaciteten */
        .
        .
    }
```

4.4 Synkronisering och kommunikation

Ärkenöt erbjuder ett flertal metoder för synkronisering och för kommunikation mellan processer.

4.4.1 Semaforer <kern:semaphore.h>

En semafor är ett verktyg för att åstadkomma synkronisering mellan olika trådar. Det sker med hjälp av operationerna signal och wait (även kallade P och V i litteraturen).

En semafor har ett värde och en kö av trådar. Operationen wait (P) testar semaforens värde. Om värdet är större än noll räknas värdet ned med ett och den anropande tråden tillåts fortsätta exekveringen. Om värdet däremot är noll, förs den anropande tråden in i semaforens kö istället, varpå en annan tråd får tillgång till processorn.

Operationen signal (V) kontrollerar om det finns någon tråd i semaforens kö. Om så är fallet flyttas den tråd som står först i kön till processorkön, varpå man på nytt bestämmer vilken tråd som skall få tillgång till processorn. Annars räknas semaforens värde upp med ett.

Ärkenöts semaforer beskrivs av dokumentet "`semaphore.h`" som definierar följande funktioner.

`semaphore_t create_sem(int init_value)`
: `create_sem` skapar en semafor och initierar dess värde med `init_value`. Värdet av funktionen är en pekare som refererar till semaforen.

`void sem_wait(semaphore_t sem)`
: `sem_wait` testar semaforens värde, om värdet är skilt från noll, räknas värdet ned med ett och `sem_wait` återvänder omedelbart. Annars ställs den anropande tråden i kö och kärnan tar upp en annan tråd.

`void sem_signal(semaphore_t sem)`
: `sem_signal` kontrollerar om semaforens kö är tom. Om så är fallet räknas semaforens värde upp med ett och `sem_signal` återvänder direkt. Om kön inte var tom flyttar `sem_signal` den tråd som är först i semaforens kö till processorkön. Vilken tråd som därefter får tillgång till processorn beror på trådarnas prioriteter.

Exempel

Följande programexempel visar på hur man kan synkronisera två trådar till varandra.

```
#include <semaphore.h>

semaphore_t mess_sem;
char *message;

void writer() {

   /* Vänta på signaler via semaforen sem. Efter varje
    * signal: skriv ut strängen message
    */
   while (1) {
      sem_wait(mess_sem);
      printf("%s\r",message);
   }
}

void fubar() {
   static char err_message[] = "Detta är ett felmeddelande";
   .
   .
   if (error) {
       message = err_message;
       sem_signal(mess_sem);
       /* detta får writer att skriva ut meddelandet */
   }
   .
   .
}
```

4.4.2 Monitorer <kern:monitor.h>

Monitorer används för att skydda data som skall kunna modifieras av flera trådar. Det går till så att man skriver separata funktioner som modifierar gemensamma data. I dessa anropar man funktionen `enter` som första åtgärd. Som sista åtgärd (före ett eventuellt `return`) anropar man `exit`. På det viset kan man garantera att endast en tråd åt gången kan modifiera data.

Ärkenöts monitorer beskrivs av dokumentet `monitor.h` som definierar följande operationer på monitorer.

`monitor_t create_monitor(void)`
: `create_monitor` skapar en monitor och lämmnar en referens till den skapade monitorn.

`void enter_monitor(monitor_t mon)`
: `enter_monitor` kontrollerar om den angivna monitorn är upptagen. Om monitorn är upptagen ställs den anropande tråden i kö för att få tillgång till monitorn. Om inte, markeras monitorn upptagen av den anropande tråden och `enter_monitor` återvänder genast.

`void exit_monitor(monitor_t mon)`
: `exit_monitor` kontrollerar monitorns kö. Om kön inte är tom markeras monitorn upptagen av den tråd i kön som har högst prioritet och den nya tråden får tillgång till processorn.

Exempel

Antag att man har två trådar med en gemensam datastruktur som båda trådarna vill modifiera. Det är inte svårt att inse att programmet kan ge mycket underliga resultat om inte de bägge trådarna hindras från att *"samtidigt"* modifiera data. Genom att använda en monitor kan man åstadkomma detta.

I exemplet nedan visas hur man kan definiera en datastruktur, med tillhörade funktioner som opererar på datastrukturen. Genom att endast använda funktionerna för att komma åt data i strukturen kan man garantera ömsesidig uteslutning av parallellt arbetande trådar.

```
#include <kern:monitor.h>

/* En datastruktur som skall skyddas */
struct {
   int foo,bar;
   monitor_t guard;
} fubar;

/* denna funktion modifierar data i strukturen ovan */
int mod1(int new) {
   int return_value;

   /* Reservera monitorn. om någon annan redan
    * reserverat den fördröjs vi här tills den är släppt */
   enter_monitor(fubar.guard);

   /* Gör vad som skall göras */
   fubar.foo = return_value = fubar.foo + fubar.bar;
   fubar.bar = new;

   /* Släpp monitorn */
   exit_monitor(fubar.guard);
   return return_value;
}

/* denna också */
int mod2(int zebra) {
   int return_value;

   enter_monitor(fubar.guard);

   return_value = fubar.bar;
   fubar.bar = zebra * fubar.bar;

   exit_monitor(fubar.guard);
   return return_value;
}
```

4.4.3 Händelser <kern:events.h>

En tråd kan invänta en händelse (eng. event). En händelse är flyktig, d.v.s den existerar bara i det ögonblick som den orsakas. När en händelse inträffar väcks alla de trådar som väntar på den. Flyktigheten hos händelsen får till följd att en tråd som inte väntar på händelsen, aldrig kommer att få reda på att den inträffat.

I Ärkenöt kan en händelse knytas till en monitor, eller vara fristående. Händelser som tillhör monitorer är till för att en tråd skall kunna invänta en förändring av de data som bevakas av monitorn. En tråd som reserverat monitorn och sedan

inväntar att något händer, lämmnar automatiskt ifrån sig monitorn och därmed kan en annan tråd komma in. När händelsen inträffar placeras de trådar som väcks i kö för att åter reservera monitorn.

En händelse som inte uttryckligen hör till någon monitor, hör till en pseudo-monitor som bevakar processorn. När en fristående händelse inträffar, placeras de trådar som väcks i kö för att få komma åt processorn.

Följande funktioner definieras av dokumentet `kern:events.h`:

`event_t create_event(monitor_t mon)`
: `create_event` skapar en händelse, knyter den till den monitor som anges av argumentet mon (om mon är en nollpekare är händelsen fristående) och lämnar en referens till händelsen.

`void event_wait(event_t evt, unsigned long timeout)`
: `event_wait` placerar den anropande tråden i händelsens väntelista. Om händelsen är knuten till en monitor utföres också ett implicit `exit_monitor`. Om `timeout` är större än minimitiden 50 mikrosekunder sätts också ett timeout så att tråden väcks senast `<timeout>` mikrosekunder efter anropet.

`void event_cause(event_t evt)`
: `event_cause` orsakar händelsen `evt`. Om händelsen `evt` är knuten till en monitor placeras alla de trådar som finns i händelsens väntelista i monitorns kö och den anropande tråden får exekvera vidare. Om händelsen däremot är fristående placeras de väntande trådarna i processorkön I detta fall bestämmer de väntande trådarnas prioriteter vilken tråd som fortsätter exekveringen.

4.4.4 Klockhändelser `<kern:timer_event.h>`

Ärkenöt har förutom dessa normala händelser en speciell sorts händelse som kan inträffa vid en given tidpunkt. Dessa händelser kan också inträffa kontinuerligt med jämna mellanrum. I allt annat beter de sig som normala händelser. Operationerna `event_cause` och `event_wait` fungerar som vanligt. Man kan dock fundera på om det är meningsfullt att använda `event_cause` på en sådan händelse. Denna typ av händelser definieras av dokumentet "`timer_event.h`" som lägger en funktion till funktionerna ovan.

`event_t create_timer_event(monitor_t mon, timevalue_t time, timevalue_t interval)`
: `create_timer_event` skapar en "klockhändelse" som inträffar först vid tiden `time` och därefter återkommande med intervallet `interval`. Om `time` är en tidpunkt i förfluten tid och `interval` är mindre än 50 mikrosekunder skapas händelsen inte. I detta fall är funktionens resultat en nollpekare. Om `time` är en tidpunkt i förfluten tid och `interval` är *större* än 50 mikrosekunder flyttas `time` framåt i steg om interval tills `time` är i framtiden.

Exempel

Den mest typiska användningen av ett `timer_event` är för att låte en tråd utföra periodiska uppgifter, t.ex en regulator som reglerar någonting med bestämda tidsintervall. Exemplet nedan visar en funktion, avsedd att exekveras som tråd, som har en huvudslinga som utförs 50 gånger per sekund.

```
#include <kern:timer_event.h>

void bar() {
   event_t timer;
   timevalue first,interval;
```

```
    gettimeofday(&first);
    interval.tv_sec = 0;
    interval.tv_usec = 20000;

    timer = create_timer_event(NULL,first,interval);
    do {
       event_wait(timer,0);
       /* Gör något. Detta utföres femtio ggr per sekund */
    } while (1);
}
```

4.4.5 Brevlådor <kern:msg_box.h>

En brevlåda kan överföra data mellan två trådar utan några förluster. Meddelanden som läggs i brevlådan kommer ut i samma ordning. En tråd som försöker lägga något i en full brevlåda fördröjs tills det finns plats för fler meddelanden. På motsvarande sätt fördröjs en tråd som försöker hämta ett meddelande ut en tom brevlåda tills det finns något att hämta.

Ärkenöts brevlådor har valfri storlek, dock ryms minst ett meddelande.

Brevlådan kan tyvärr inte vara så liten att den inte rymmer något meddelande alls. Detta fall kallas för ett rendez-vous, trådarna möts när/där de vill utbyta information. Vid ett rendez-vous fördröjs den tråd som försöker lämmna ett meddelande tills det kommer en annan process som vill hämta meddelandet och vice versa.

För operationer på brevlådor finns följande funktioner tillgängliga.

`msg_box_t msg_box_create(int size)`
: `msg_box_create` skapar en brevlåda som rymmer `size` meddelanden och returnerar en referens till brevlådan. `size` måste vara minst 1.

`void putmsg(msg_box_t box,void **message)`
: `putmsg` lägger att meddelande i brevlådan. message är en pekare till en pekare till meddelandet. pekaren till meddelandet nollställs, så att den anropande tråden inte kan komma åt meddelandet i efterhand. Om brevlådan är full fördröjs den anropande tråden tills brevlådan har tömts på tillräckligt många meddelanden.

`boolean_t putmsg_nowait(msg_box_t box,void **message)`
: `putmsg_nowait` gör samma sak som `putmsg` med skillnaden att den anropande tråden inte fördröjs om brevlådan var tom. Funktionen har värdet TRUE om den lyckades placera meddelandet i brevlådan, FALSE annars.

`void getmsg(msg_box_t box,void **message)`
: `getmsg` plockar det äldsta meddelandet ur brevlådan. `message` är en pekare till en pekare som skall peka på meddelandet, det tidigare innehållet i pekaren förstörs. Om brevlådan är tom fördröjs den anropande tråden tills det kommer in ett meddelande.

`boolean_t getmsg_nowait(msg_box_t box,void **message)`
: `getmsg_nowait` är som `getmsg` med skillnaden att den anropande tråden inte fördröjs om brevlådan är tom. Funktionen har värdet TRUE om brevlådan innehöll ett meddelande, FALSE annars.

Exempel

Exemplet nedan gör samma sak som i exemplet med semaforer ovan, med skillnaden att meddelanden nu skrivs ut i korrekt ordning och att inga meddelanden går förlorade. Det går givetvis att lösa med semaforer också, men detta är elegantare.

```
#include <msg_box.h>

/* global pekare till en brevlåda */
msg_box_t mailbox;

{
   .
   .
   /* Skapa en brevlåda som rymmer tre meddelanden. */
   mailbox = create_msg_box(3);
   .
   .
}


void writer() {
   char *message;
   /* Vänta på meddelanden i brevlådan mailbox. När det kommer
    * ett meddelande: skriv ut det på skärmen.
    */
   while (1) {
      getmsg(mailbox,&message);
      printf("%s\r",message);
   }
}

void fubar() {
   char *message;
   static char err_message[] = "Detta är ett felmeddelande";
   .
   .
   if (error) {
       message = err_message;

       /* lägg meddelandet i brevlådan */
       putmsg(mailbox,&message);

       /* Nu är message == NULL */
   }
   .
   .
}
```

4.5 Drivrutiner för hårdvara

I det system som jag har använt för utvecklingen har det även funnits en hel del periferienheter. Många av dessa finns placerade på samma instickskort som centralprocessorn, men det har också funnits ytterligare instickskort som det har varit intressant att kunna kontrollera. Denna avdelning handlar om dem.

4.5.1 mio16 <drivers:nb_mio_16.h>

Denna drivrutin består i själva verket av en hel mängd olika drivrutiner. Detta beror på att den hårdvara som skall kontrolleras är mycket komplex. Drivrutinsamlingen kan kontrollera ett instickskort av typen NB-MIO-16-XX.

Drivrutinerna erbjuder funktioner för in- och utmatning av både analoga och digitala signaler. Det finns även funktioner för programmering av de klockor/räknare som är tillgängliga för allmänt bruk.

Den mest komplexa funktionen är den som läser in analoga värden från kortet. Detta beror på att A/D-omvandlaren är relativt långsam, så långsam att det lönar sig att låta processorn övergå till en annan uppgift medan A/D-omvandlingen utförs. Den tråd som anropar denna funktion fördröjs därför med en semafor. Semaforen aktiveras sedan av en avbrottsrutin, som anropas när A/D-omvandlingarna är klara.

Tyvärr har det visat sig att denna drivrutin inte klarar att krama ur mesta möjliga prestanda ur signalprocessorn. Detta beror bland annat på att busstransaktioner över NuBus kostar relativt mycket tid. Av denna anledning hoppas jag kunna utveckla en ny drivrutin som utnyttjar DMA för överföring av data och på det viset sparar arbete åt processorn.

För mer information om bakgrunden till denna drivrutinsamling hänvisas till instruktionsboken för familjen NB-MIO-16.

Drivrutinen definieras av dokumentet `nb_mio_16.h` som definierar en typ `mio_16` som representerar drivrutinen, samt följande funktioner.

`mio_16 mio_16_init(int board_slot, int AD_conv_time)`
: `mio_16_init` initialiserar ett NB-MIO-16-XX-kort som förutsätts vara placerat i kortplatsen `slot`, initialiserar en post som innehåller allt som drivrutinerna behöver veta, samt returnerar en referens till den posten.

`void mio_16_reset(mio_16 board)`
: `mio_16_reset` reinitialiserar det instickskort som refereras av den post som `board` pekar på.

`void mio_16_analog_in(mio_16 board, long *channels, int nr_of_channels, float *data)`
: `mio_16_analog_in` läser in analoga signaler från `nr_of_channels` kanaler vilkas nummer beskrivs av vektorn `channels`. Resultatet är flyttalsvärden mellan minus ett och ett, som hamnar i vektorn `data`. `data` måste vara tillräckligt stor för att rymma alla värden. Kanalernas känslighet ställs in för området $[-10..+10]$ Volt.

`float mio_16_analog_out(mio_16 board, int channel, float value)`
: `mio_16_analog_out` uppdaterar den analoga utkanalen `channel` till värdet `value`. `value` är ett flyttal mellan minus ett och ett, vilket motsvarar ett utspänningsområde på $[-10..+10]$ Volt. Funktionen begränsar automatiskt värdet och returnerar det begränsade och kvantiserade värdet.

`void mio_16_setup_digital_io(mio_16 board, int config)`
: `mio_16_setup_digital_io` ställer in vilka fysiska digitala signaler som skall vara insignaler och vilka som skall vara utsignaler. config kan ha ett av följande värden:

  `DIO_IN_ONLY` Både A[0:3] och B[0:3] är ingångar. Detta är normalvärdet.

  `DIO_OUT_ONLY` Både A[0:3] och B[0:3] är utgångar

  `DIO_BIN_AOUT` Signalerna A[0:3] konfigureras som utgångar, B[0:3] som ingångar.

  `DIO_AIN_BOUT` Signalerna B[0:3] konfigureras som utgångar, A[0:3] som ingångar.

`int mio_16_get_digital_in(mio_16 board)`
: `mio_16_get_digital_in` returnerar det logiska värdet av de åtta digitala ingångarna. Resultatet ges i bit[0:7] av returvärdet. bitarna[0:3] motsvarar A[0:3] medan bit[4:7] motsvarar B[0:3].

`void mio_16_set_digital_out(mio_16 board, char value)`
: `mio_16_set_digital_out` skriver `value` till de digitala utgångarna. Bitarna[0:3] i value skrivs till utgångarna A[0:3] medan bitarna[4:7] skrivs till utgångarna B[0:3].

Exempel

Här ges ett exempel på hur man kan använda drivrutinerna för att implementera en PI-regulator. regulatorn samplar två kanaler, den ena tas som ärvärdet, den andra som börvärdet. Därefter beräknas styrsignalen, som matas ut på en av de analoga utgångarna. Integraldelen uppdateras efter begränsning av styrsignalen för att undvika integratoruppvridning.

```
#include <kern:timer_event.h>
#include <drivers:nb_mio_16.h>

#define  SAMPLE_INTERVAL  250   /* 250 µs */

float Kp,Ti;
mio_16 the_board;

/* denna funktion exekveras när programmet avslutas, det ser
 * atexit-satsen nedan till
 */
void endfunc() {
    mio_16_reset(the_board);
}

void main() {
   event_t timer;
   timevalue first,interval;

   float Tsamp = 1e-6*SAMPLE_INTERVAL;
   float Ki,idel = 0.0;
   float u,ulim;

   int channels[] = {0,2};
   inr nr_of_channels = 2;
   struct {
       float y,y_sp
   } input;

   /* Initialisera klockhändelsen som styr
    * sampelintervallet */
   gettimeofday(&first);
   interval.tv_sec = 0;
   interval.tv_usec = SAMPLE_INTERVAL;

   timer = create_timer_event(NULL,first,interval);

   /* Initialisera drivrutinerna för I/O-kortet
    * NB-MIO-16 i kortplats 4
    */
```

```
    the_board = mio_16_init(4);

    /* om programmet stoppas skall I/O-kortet återställas till
     * ursprungstillståndet */
    atexit(endfunc);

    do {
       event_wait(timer,0);
       /* läs av ingångarna */
       mio_16_analog_in(the_board,channels,
                        nr_of_channels,(float*) &input);

       /* beräkna styrsignal */
       idel += (input.y_sp - input.y)/Ti;
       u = Kp * (input.y_sp - input.y + idel);

       /* mata ut styrsignalen, funktionen begränsar automatiskt
        * styrsignalen. den begränsade styrsignalen hamnar i
        * ulim.  */
       ulim = mio_16_analog_out(the_board,0,u);

       /* Begränsa integraldelen för att undvika
        * integratoruppvridning
        */
       idel -= (u - ulim)/Kp;

    } while (1);
}
```

4.5.2 ao6 <drivers:nb_ao_6.h>

Denna drivrutin ger tillgång till de sex analoga utgångar som finns på instickskortet NB-AO-6. Drivrutinen möjliggör samtidig uppdatering av flera kanaler. Man kan också ge samma värde till flera kanaler med ett enda anrop. Nedan listas de funktioner som finns tillgängliga.

`ao_6 ao_6_init(int board_slot)`
: `ao_6 ao_6_init` initialiserar ett instickskort av typen NB-AO-6, monterat på kortplats nummer `board_slot` samt returnerar en referens till en post som innehåller allt som resten av drivrutinen behöver veta om kortet.

`float ao_6_analog_out(ao_6 board, int channels, float value)`
: `ao_6_analog_out` hanterar de analoga utgångarna på kortet NB-AO-6. vilka utgångar som påverkas och hur bestäms av `channels` som är en kombination (logiskt eller eller summa) av de följande värdena.

  `AO_6_chan0` `value` skrivs till registret för kanal 0

  `AO_6_chan1` `value` skrivs till registret för kanal 1

  `AO_6_chan2` `value` skrivs till registret för kanal 2

  `AO_6_chan3` `value` skrivs till registret för kanal 3

  `AO_6_chan4` `value` skrivs till registret för kanal 4

  `AO_6_chan5` `value` skrivs till registret för kanal 5

  `AO_6_update` samtliga utsignaler uppdateras med innehållet i respektive kanals register.

Lägg märke till att man alltså kan förbereda flera kanaler och sedan uppdatera alla på en gång, liksom man kan uppdatera alla kanaler med samma värde i en enda operation. `value` är ett flyttal mellan minus ett och ett, vilket motsvarar ett utspänningsområde på -10 - $+10$ Volt. Värdet av funktionen `ao_6_analog_out` är detsamma som `value`, efter begränsning och kvantisering till tolv bitars upplösning.

För mer information om bakgrunden till denna drivrutin hänvisas till instruktionsboken för instickskortet NB-AO-6.

4.6 Värddatorkommunikation

De här kommunikationsalternativen är i första hand tänkta för användning tillsammans med LabVIEW. I kapitel 5 presenteras motsvarande funktioner som utför värddatorsidan av kommunikationsfunktionerna nedan. Den som vill och har ork att sätta sig in i uppbyggnaden (dvs. läsa källkoden...) kan givetvis utnyttja samma funktioner för ett mer specialiserat operatörsgränssnitt.

4.6.1 Delade semaforer <peripherials:shared_variables.h>

Delade semaforer är tekniskt sett minnesceller, som har skydd mot modifikation av mer än en dator samtidigt. Skyddet består av ytterligare tre minnesceller som används för Dekkers algoritm. Dessa variabler/semaforer kan användas för synkronisering mellan ett program som exekverar på signalprocessorn och ett program som exekverar på värddatorn.

Följande funktioner opererar på de delade variablerna.

`long shared_var_read(int index)`
: `shared_var_read` returnerar värdet av variabeln.

`void shared_var_write(int index, long value)`
: `shared_var_write` skriver `value` i den skyddade variabeln.

`long shared_var_give(int index)`
: `shared_var_give` adderar ett till den delade variabeln som ges av index, Funktionens resultat är den delade variabelns nya värde.

`long shared_var_take(int index)`
: `shared_var_take` testar om den delade variabeln som ges av index har ett värde större än noll. I så fall räknas värdet ned med ett. Funktionens resultat är variabelns värde före anropet.

4.6.2 Obuffrad kommunikation <peripherials:shared_data.h>

Obuffrad kommunikation utförs med 128 minnesceller. Dessa är uppdelade i dels 64 värden som signalprocessorn endast kan läsa och värddatorn endast kan modifiera, dels 64 värden som fungerar tvärtom; signalprocessorn kan bara modifiera och värddatorn endast läsa av. Numreringen av dessa värden är densamma oavsett dataflödets riktning; i båda fallen numreras de från 0 till 63. Se även 5.2.

För signalprocessorn finns följande makrofunktioner tillgängliga som hanterar överföringen:

`void write_shared_data(int index, <Type> data, Type)`
: `write_shared_data` skriver `data` på plats nummer `index`. `Type` är en av C:s fördefinierade typer. Anledningen till att det fungerar är att signalprocessorn har samma storlek på alla datatyper.

`void read_shared_data(int index, <Type> data, Type)`
: `read_shared_data` returnerar värdet på platsen `index` i variabeln `data`. Notera att eftersom det är ett makro behöver man inte ta hänsyn till C:s normala "pass-by-value"-mekanism.

Observera åter att `index` skall vara från noll till sextiotre för båda funktionerna och att ett write till ett visst index följt av ett read från samma index inte ger samma resultat, det är fråga om separata kanaler.

4.6.3 Buffrad kommunikation <peripherials:shared_buffers.h>

Detta är buffertar med data som kan används när man vill överföra textmeddelanden, när man vill överföra signalförlopp utan att tappa något och när man har varierande kapacitet hos värddatorn för att ta emot meddelanden. Det finns åtta buffertar. Det är inte på förhand bestämt vilken "riktning" som data går i. Både värddatorn och signalprocessorn kan både skriva och läsa ur samma buffert.

Påpekas bör att buffert nummer noll används av Ärkenöt för utskrift av texter i ett terminalfönster, varför man bör vara försiktig med vad man utnyttjar denna buffert till (det går, men man måste veta vad man håller på med).

Signalprocessorn kan operera på buffertarna med följande funktioner.

`int shared_buffers_put(int index, boolean_t wait, long data)`
: `shared_buffers_put` skriver `data` i bufferten. Om bufferten är full och `wait` har värdet `TRUE` läggs den anropande tråden i vänteslinga tills värddatorn plockar ut något ur bufferten. Om `wait` har värdet `FALSE` och bufferten är fylld återvänder funktionen utan att lägga in något i bufferten. `shared_buffers_put` har i detta fall värdet `FALSE`.

`int shared_buffers_forced_put(int index, int data)`
: `shared_buffers_forced_put` skriver `data` i bufferten. Om bufferten är full skrivs det äldsta elementet i bufferten över och en räknare räknas upp för att tala om att data gått förlorade.

`int shared_buffers_get(int index, boolean_t wait, long &data)`
: `shared_buffers_get` Läser det äldsta elementet ur bufferten och placerar det i data. Om `wait` har värdet `TRUE` och bufferten är tom läggs den anropande tråden i vänteslinga tills värddatorn skriver något i bufferten. Om `wait` har värdet `FALSE` får funktionen värdet `FALSE` om bufferten var tom.

4.6.4 Formatomvandlingar <peripherials:toIEEEtoC30.h>

Signalprocessorn TMS320C30 har ett flyttalsformat som skiljer sig från det gängse standardformatet IEEE754. Eftersom Apples datorer använder sig av IEEE754 för flyttalsberäkningar så måste man omvandla flyttalsvärden innan man överför dem. På samma sätt måste flyttalsvärden som genereras av värddatorn omvandlas innan man kan använda dem i signalprocessorn. Dessa omvandlingar sköts om av följande funktioner, definierade i dokumentet "`toIEEEtoC30.h`".

`IEEEfloat C30toIEEE(float C30value)`
: Funktionen `C30toIEEE` omvandlar ett flyttal i TMS320C30:s interna format till ett flyttal i formatet IEEE754.

`IEEEfloat C30toIEEE_fast(float C30value)`
: Funktionen `C30toIEEE_fast` omvandlar alla *normala* flyttal korrekt, men klarar inte specialfallen som `C30toIEEE` klarar. I gengäld är den ca 15% snabbare.

`float IEEEtoC30(IEEEfloat IEEEvalue)`
Funktionen `IEEEtoC30` omvandlar ett flyttal på formatet IEEE754 till ett flyttal i TMS320C30:s interna format.

`float IEEEtoC30_fast(IEEEfloat IEEEvalue)`
Funktionen `IEEEtoC30_fast` omvandlar alla *normala* flyttal korrekt, men klarar inte specialfallen som `IEEEtoC30` klarar. I gengäld är den ca 15% snabbare.

Exempel

```
#include <peripherials:toIEEEtoC30.h>

{
   IEEEfloat IEEEvalue;
   float     C30value;
   .
   .
   .
   C30value = 1.7e+1;
   /* omvandla till IEEE754-format */
   IEEEvalue = C30toIEEE(C30value);
}
```

Kapitel 5

Stödfunktioner i LabVIEW

Detta kapitel handlar om de virtuella instrument som kommunicerar med signalprocessorkortet. Instrumenten är till för att stödja utvecklingen av eleganta användargränssnitt till signalprocessorprogrammet. Genom att utnyttja dessa kan man enkelt få LabVIEW att ladda ned program till samt starta och stoppa signalprocessorn. Det är också mycket enkelt att transportera data mellan LabVIEW och signalprocessorn.

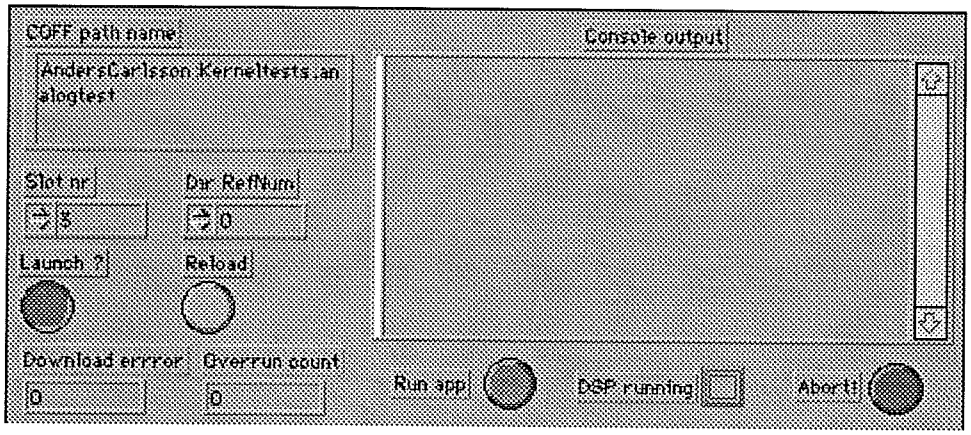
5.1 Allmänna stödfunktioner

5.1.1 DSP Console VI

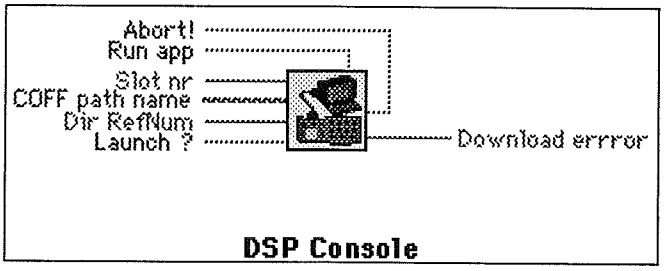
DSP Console VI är ett virtuellt instrument som kan ladda ned ett exekverbart program till signalprocessorn, starta programmet, läsa av vad signalprocessorprogrammet skriver ut (med printf, puts och putchar) samt stoppa programmet och återstarta programmet. Instrumentet är huvudsakligen tänkt för användning underordnat ett annat virtuellt instrument, men det kan även användas fristående.

Figur 5.1 avbildar det man ser på Macens bildskärm när man använder DSP Console som ett fristående instrument. De olika reglagen och indikatorerna har funktion enligt nedan.

COFF path name Komplett filspecifikation (Apple Macintosh syntax) till det kompilerade DSP-programmet.



Figur 5.1: Frontpanel till instrumentet DSP Console



Figur 5.2: Inkopplingsanvisning för det virtuella instrumentet DSP Console

Slot nr Den kortplats som DSP-kortet är monterat på

Dir RefNum Behöver ej anslutas, men skall normalt vara noll. För detaljerad information hänvisas till manualen till LabVIEW.

Launch ? Om denna knapp är intryckt när instrumentet startas, startas DSP-programmet direkt efter att programmet laddats ned.

Run app Ett tryck på denna knapp under det att DSP Console exekveras, startar DSP-programmet, om processorn står still.

Abort! Stoppar all verksamhet på DSP:n, om det pågår någon.

Reload Här kan man trycka för att ladda ned programmet igen, signalprocessorn måste vara stoppad.

DSP running Detta är en indikatorlampa som lyser om signalprocessorn exekverar program.

Console output Denna indikator visar den text som skrivs ut via printf m.fl. i DSP-programmet.

Download error Indikerar felkoden om det blev fel vid nedladdningen av programmet (Samma felkoder som för Macintosh i övrigt).

Overrun count Denna indikator visar hur många tecken som gått förlorade på grund av massiva utskrifter.

När man skall bygga upp mer komplexa gränssnitt använder man DSP Console som hjälpinstrument. I detta fall lägger man in DSP Console i det överordnade instrumentets diagram och kopplar in det enligt anvisningarna i figur 5.2

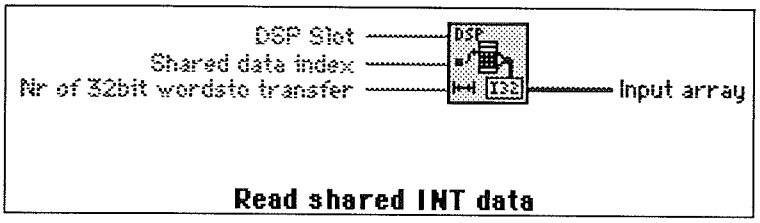
5.1.2 Interrupt DSP VI

Do DSP function VI är ett instrument som kan få signalprocessorn att utföra någon viss funktion på kommando. Argumentet OpCode bestämmer vilken funktion som signalprocessorn skall utföra. Dessutom kan man ge två argument (Arg1 och Arg2) till funktionen.

Den funktion som signalprocessorn utför vid en viss funktionskod kan bestämmas av signalprocessorprogrammet, utom för de funktionskoder som är upptagna, se tabell 5.1.

Opkod	Funktion
0	Start/Stopp av program
1	Läs minnesadress "peek"
2	Skriv på minnesadress "poke"

Tabell 5.1: Funktionskoder och dess funktion



Figur 5.3: Inkopplingsanvisning för det virtuella instrumentet Read shared INT data

5.2 Stöd för delade data

Stödet för abstraktionen delade data består av sex instrument.

Samtliga dessa instrument är inkapslingar till totalt fyra kodgränssnittsnoder, som utför det verkliga arbetet med att överföra data mellan Macens internminne och signalprocessorns delade minne. Den viktiga fördelen med detta arrangemang är att det minskar risken för fel. Detta eftersom ett virtuellt instrument utför typkontroll och eventuellt typomvandlingar på sina argument, medan en kodgränssnittsnod inte har en aning om vad funktionen den anropar egentligen vill ha för argument. Om man ger felaktiga och/eller för få argument till en kodgränssnittsnod kan man få datorn att krascha grundligt. En annan viktig fördel är att man bara behöver ändra i det kapslande instrumentet när man ändrar den funktion som anropas av kodgränssnittet. Detta eftersom virtuella instrument länkas dynamiskt av LabVIEW.

Samtliga instrument förväntar sig ett argument DSP Slot, som talar om vilken kortplats som DSP-kortet är monterat på. För samtliga instrument gäller också att argumentet Index motsvarar det index som man anger på signalprocessorsidan.

Signalprocessorns sida av detta kommunikationssätt beskrivs i avsnitt 4.6.2

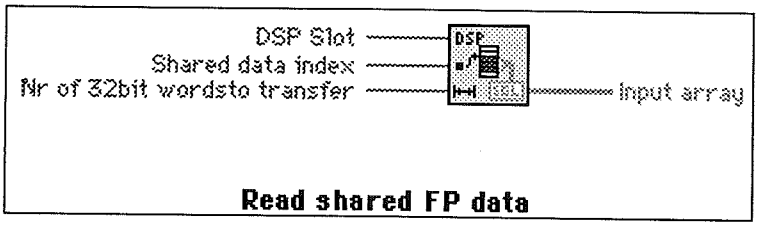
5.2.1 Read shared INT data VI

Instrumentet Read shared INT data läser data som signalprocessorn skrivit med funktionen `write_shared_data`, tolkat som heltal. Argumentet Index motsvarar indexet på signalprocessorsidan. Argumentet Antal talar om hur många element som skall läsas. Den vektor som returneras är en vektor av heltal med ¡Antal¿ element.

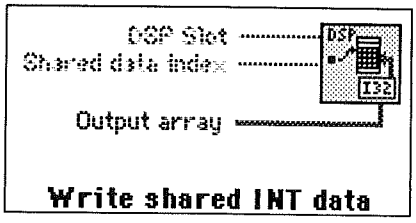
5.2.2 Read shared FP data VI

Instrumentet Read shared FP data gör samma sak som Read shared INT data med skillnaden att det returnerar en vektor av flyttal i Enkel precision.

Index motsvarar indexet på signalprocessorsidan. Argumentet Antal talar om hur många variabler som skall skrivas.



Figur 5.4: Inkopplingsanvisning för det virtuella instrumentet Read shared FP data



Figur 5.5: Inkopplingsanvisning för det virtuella instrumentet Write shared INT data

5.2.3 Write shared INT data VI

Instrumentet Write shared INT data skriver en vektor av heltal med början vid indexet som anges av argumentet index. De skrivna data kan sedan läsas av signalprocessorprogrammet med funktionen `read_shared_data`.

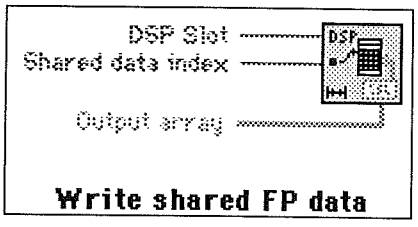
5.2.4 Write shared FP data VI

Instrumentet Write shared FP data gör samma sak som Write shared INT data med skillnaden att det förväntar sig en vektor av flyttal i Enkel precision som argument.

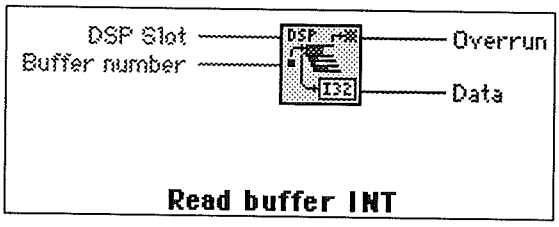
5.3 Stöd för delade buffertar

Delade buffertar stöds av fyra virtuella instrument, två som hanterar flyttal respektive två som hanterar heltal. Precis som instrumenten för delade data är de inkapslingar av Kodgränssnittsnoder (CIN).

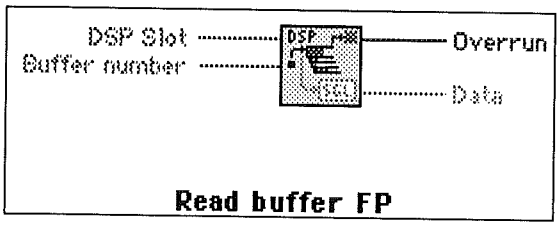
Det finns totalt åtta buffertar för överföring av data. Buffert nummer noll



Figur 5.6: Inkopplingsanvisning för det virtuella instrumentet Write shared FP data



Figur 5.7: Inkopplingsanvisning för det virtuella instrumentet Read buffer INT



Figur 5.8: Inkopplingsanvisning för det virtuella instrumentet Read buffer FP

används av DSP Console för överföring av textutskrifter, men man kan använda även denna buffert för dataöverföring, bara man vet vad man håller på med.

Motsvarande funktioner för signalprocessorn beskrivs i avsnitt 4.6.3

5.3.1 Read buffer INT VI

Instrumentet Read buffer INT läser ett 32bit ord, tolkat som heltal, från den buffert som anges av Argumentet Buffer number. Det lästa talet returneras i Data. Returvärdet overrun anger om och hur många ord som gått förlorade sedan data senast lästes.

5.3.2 Read buffer FP VI

Instrumentet Read buffer FP är identiskt med instrumentet Read buffer INT med skillnaden att det tolkar det lästa ordet som ett flyttal i enkel precision.

5.3.3 Write buffer INT VI

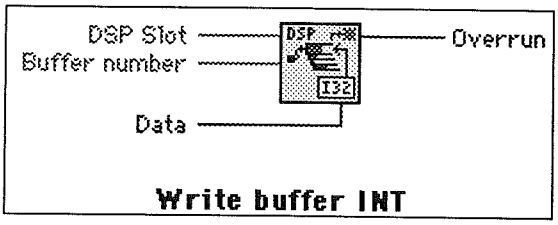
Instrumentet Write buffer INT Skriver ett heltal, Data, i den buffert som anges av Buffer number. Overrun indikerar om gamla data skrevs över av instrumentet.

5.3.4 Write buffer FP VI

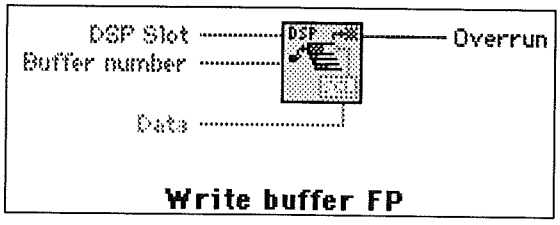
Instrumentet Write buffer FP är identiskt med instrumentet Write buffer INT med skillnaden att det skriver ett flyttal i enkel precision.

5.4 Stöd för semaforer

Det finns åtta semaforer som kan användas för synkronisering mellan LabVIEW och signalprocessorn. Dessa semaforer stöds av fyra virtuella instrument. Precis



Figur 5.9: Inkopplingsanvisning för det virtuella instrumentet Write buffer INT



Figur 5.10: Inkopplingsanvisning för det virtuella instrumentet Write buffer FP

som instrumenten för delade data är de inkapslingar av kodgränssnittsnoder som utför det egentliga arbetet.

Samtliga instrument tar ett argument, DSP Slot, som anger den kortplats som signalprocessorkortet är monterat i. De tar också ett argument, Semaphore number, som anger vilken av de åtta semaforerna som skall användas. Semaforerna är numrerade från noll till sju.

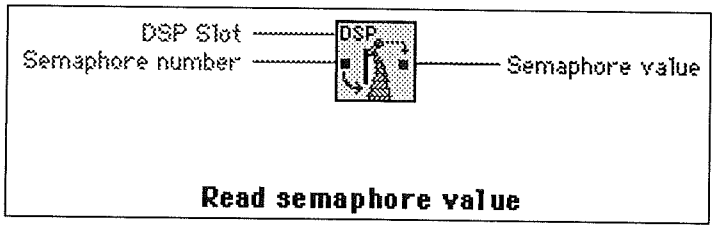
Motsvarande funktioner för signalprocessorn beskrivs i avsnitt 4.6.1. Tyvärr är inte namngivningen av dessa funktioner lika på båda sidor vilket kan orsaka viss förvirring.

5.4.1 Read semaphore value VI

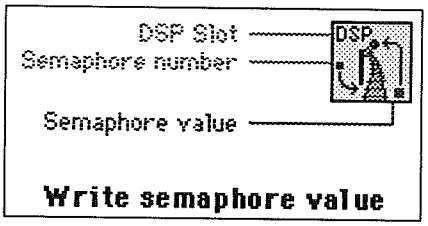
Instrumentet Read semaphore value läser av värdet av den semafor som anges av argumentet Semaphore number. Semaforens värde tolkas som heltal.

5.4.2 Write semaphore value VI

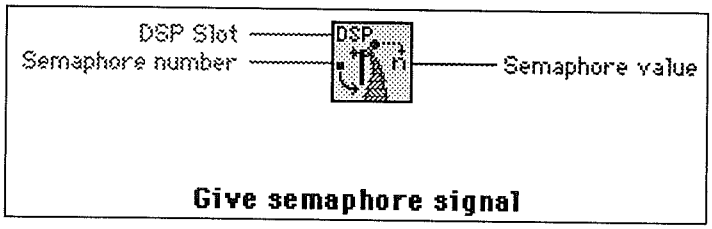
Instrumentet Write semaphore value tilldelar värdet som anges av argumentet Semaphore value till den semafor som anges av argumentet Semaphore number.



Figur 5.11: Inkopplingsanvisning för det virtuella instrumentet Read semaphore value



Figur 5.12: Inkopplingsanvisning för det virtuella instrumentet Write semaphore value



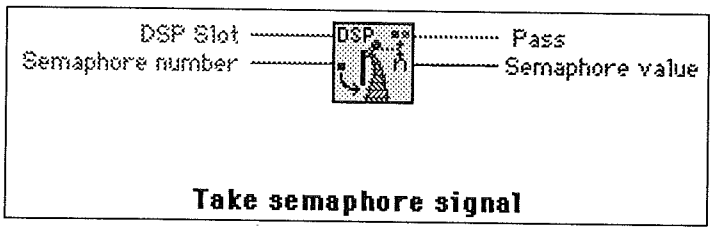
Figur 5.13: Inkopplingsanvisning för det virtuella instrumentet Give semaphore signal

5.4.3 Give semaphore signal VI

Instrumentet Give semaphore signal lägger till ett till värdet, tolkat som heltal, av den semafor som anges av argumentet Semaphore number. Semaforens nya värde returneras i Semaphore value.

5.4.4 Take semaphore signal VI

Instrumentet Take semaphore signal testar semaforens värde, tolkat som heltal. Om semaforens värde är större än noll minskas semaforens värde med ett, annars lämnas värdet oförändrat. Det gamla värdet returneras i Semaphore value. Det booleska returvärdet Pass är sant om semaforens gamla värde var minst ett.



Figur 5.14: Inkopplingsanvisning för det virtuella instrumentet Take semaphore signal

Kapitel 6

Exempel: en asynkronmaskinsregulator

6.1 Inledning

Denna del av rapporten handlar om den andra delen av mitt examensarbete, nämligen att implementera en regulator för någon process. Villkoren för denna uppgift var att den valda processen helst skulle vara så snabb och/eller så komplex, att man inte hade kunnat genomföra reglering av den utan tillgång till signalprocessorns stora beräkningskapacitet. Syftet med denna deluppgift var att demonstrera hur man kan bygga upp en regulator med hjälp av de funktioner som realtidskärnan (med kringliggande hjälpfunktioner) erbjuder.

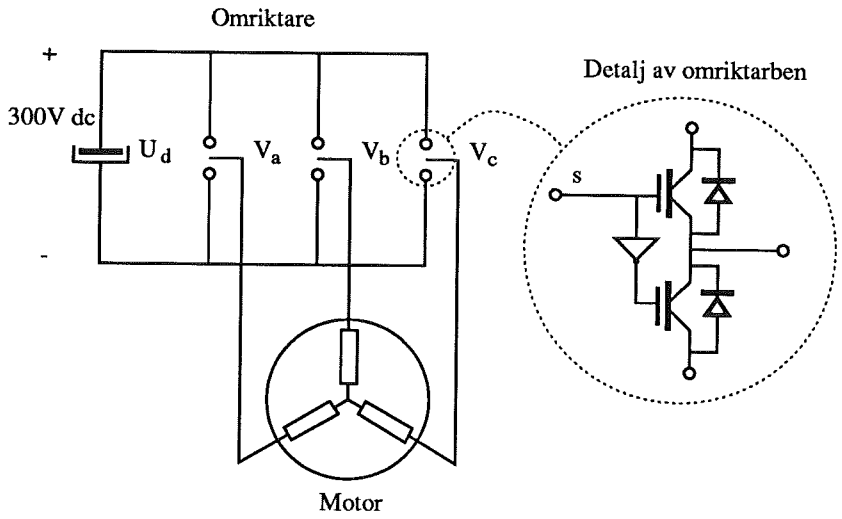
Den "process" jag valt för denna uppgift är Asynkronmaskinen. Asynkronmaskinen är en synnerligen populär och vanlig motor som används i de mest skiftande tillämpningar. Den har dock nackdelen att vara svår att använda i tillämpningar som kräver nogrann varvtalskontroll eller goda servoegenskaper. Dessa nackdelar kan man komma förbi med modern reglerteknik. Väl genomfört får den reglerade motorn så goda egenskaper att den överträffar en motsvarande reglerad likströmsmotor.

Så långt som att utveckla ett servo har vi inte gått, men vi har implementerat en varvtals/moment regulator med goda prestanda.

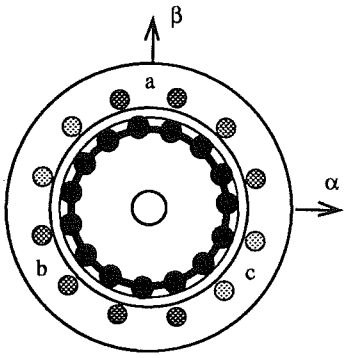
Motordrifter kräver generellt korta sampeltider, speciellt om man även skall utföra momentregleringen i mjukvara. Denna är inget undantag, speciellt som asynkronmaskinen styrs med växelstorheter, som kan ha frekvenser uppemot 100-150 Hz. Den läsare som har insikter i digital reglering förstår givetvis att här krävs sampelintervall som inte är längre än en millisekund. Om man därtill, vilket visas i de efterföljande avdelningarna, lägger den mängd beräkningar som skall utföras mellan varje sampel, har jag förhoppningsvis troliggjort att den valda processen väl fyller de uppställda villkoren.

Denna del av mitt examensarbete har genomförts i samarbete med Mats Alaküla vid Institutionen för Industriell Elektroteknik och Automation (IEA). Anledningen till det är att IEA har utrustningen som behövs för att laborera med denna typ av reglerproblem.

Regulatorn bygger på en befintlig utrustning som används inom kursverksamheten vid institutionen för Industriell Elektroteknik och Automation. Denna utrustning implementerar en asynkronmaskinsregulator i analog elektronik. Den digitala regulator som vi byggt upp utnyttjar en del av den elektronik som finns i denna utrustning. Utrustningen mäter motorspänningar och beräknar ett "tomgångsflöde" som vi sedan mäter. Det finns även mätdon för strömmar samt en enkel tachometer.



Figur 6.1: shematisk bild av en trefas växelriktare, inkopplad till en motor.



Figur 6.2: Det finns vanligtvis tre lindningshärvor i en asynkronmaskin, de relateras till statorkoordinater (α, β) som på bilden.

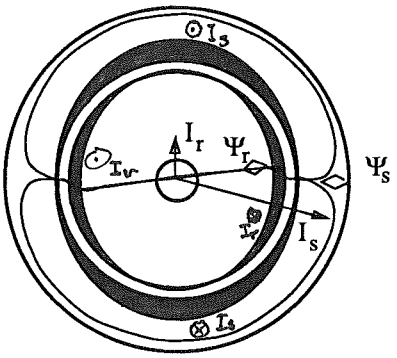
De mätvärden som utrustningen producerar är lagom stora för att mäta med den A/D-omvandlare som funnits som periferienhet till signalprocessorn.

I lab-utrustningen ingår en strömriktare med tillhörande styrelektronik, se figur 6.1, ansluten till motorn. Styrelektroniken matas med ledvärden för hur stor pulskvot som det skall vara i varje "ben" och sköter sedan om pulsbreddsmoduleringen helt automatiskt.

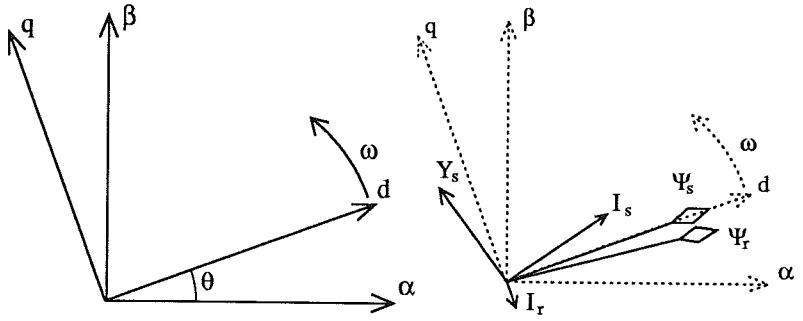
6.2 Asynkronmaskiner

Den vanligaste typen av asynkronmaskin består av en rotor med en kortsluten burlindning, placerad i en stator med tre lindningshärvor. Statorns lindningshärvor är förlagda så att de bildar tre spolar, fysiskt förskjutna 120° från varandra runt motoraxeln.

Det normala arbetssättet innebär att man ansluter sinusspänningar till de tre spolarna. Sinusspänningarna har en fasförskjutning om 120° från varandra. Man kan visa att denna matning i stationaritet alstrar ett magnetiskt flöde genom maskinen. Flödet roterar runt motoraxeln med samma vinkelfrekvens som den matande växelspänningen. Det roterande flödet inducerar strömmar i rotorns burlindning,



Figur 6.3: Sinusformig strömbeläggning i gränsytorna, samt ström- och flödesvektorer i maskinen.



Figur 6.4: Koordinatsystem och Vektorer som används för att beskriva funktionen av en asynkronmotor

som då drar med sig rotorn i flödets rotation.

När motorn avger ett vridmoment kommer rotorn att rotera något långsammare än fältbilden i motorn. Detta eftersom momentet bildas av flödet i samverkan med inducerade strömmar i rotorn, som upprätthålls just av rotationen relativt flödesfältbilden. Rotorn roterar alltså asynkront med det roterande flödet, därav namnet på maskintypen.

Man kan visa att det dynamiska system som asynkronmotorn utgör kan beskrivas av två komplexa vektordifferentialekvationer. En för statorlindningarna, indexerad med s, samt en för rotorns burlindning, indexerad med r.

Dessa differentialekvationer kan också uttryckas i olika koordinatsystem, beroende på vilken aspekt av motorn som man vill beskriva. De två koordinatsystem som är intressanta i en regulator av det slag som vi byggt upp är dels ett som är fast i "rummet", kallat α-β-koordinater eller statorkoordinater, dels ett som är fast relativt *flödets fältbild* i maskinen, kallat d-q-koordinater. Se även figur 6.4.

Om man delar upp vektordifferentialekvationerna komponentvis får man följande system:

$$\begin{aligned}
\frac{d}{dt}\psi_{s\alpha} &= u_{s\alpha} - R_s \cdot i_{s\alpha} \\
\frac{d}{dt}\psi_{s\beta} &= u_{s\beta} - R_s \cdot i_{s\beta} \\
\frac{d}{dt}\psi_{r\alpha} &= -\omega_r \cdot \psi_{r\beta} - R_r \cdot i_{r\alpha} \\
\frac{d}{dt}\psi_{r\beta} &= \omega_r \cdot \psi_{r\alpha} - R_r \cdot i_{r\beta}
\end{aligned}$$

Där

$$\psi_{s\alpha} = (L_m + L_{s\lambda}) \cdot i_{s\alpha} + L_m \cdot i_{r\alpha}$$
$$\psi_{s\beta} = (L_m + L_{s\lambda}) \cdot i_{s\beta} + L_m \cdot i_{r\beta}$$
$$\psi_{r\alpha} = (L_m + L_{r\lambda}) \cdot i_{r\alpha} + L_m \cdot i_{s\alpha}$$
$$\psi_{r\beta} = (L_m + L_{r\lambda}) \cdot i_{r\beta} + L_m \cdot i_{s\beta}$$

L_m är maskinens huvudinduktans. Denna induktans alstrar maskinens huvudflöde. $L_{s\lambda}$ är statorns läckinduktans. Alstrar ett delflöde som går genom statorn, men som inte berör rotorn. $L_{r\lambda}$ är rotorns läckinduktans. R_s är statorns resistans. R_r är rotorns resistans.

Man kan addera huvudinduktanser och läckinduktansen för respektive stator och rotor varvid man får

$$L_m + L_{s\lambda} = L_s \text{ som kallas statorinduktans och}$$
$$L_m + L_{r\lambda} = L_r \text{ som kallas rotorinduktans.}$$

Ekvationerna beskriver ett linjärt system med fyra tillståndsvariabler och en tidsvariabel parameter, rotorvinkelhastigheten. Detta är fullt rimligt eftersom ω_r (motorns varvtal) varierar långsamt relativt systemets tillståndsvariabler. Vill man ha det olinjära systemet i stället får man utöka systemet med ett tillstånd.

Man kan givetvis skriva systemet på tillståndsform. I α–β -koordinater ser det ut som följer:

$$\dot{\vec{x}} = A \cdot \vec{x} + B \cdot \vec{u}$$
$$\vec{y} = C \cdot \vec{x}$$

Där

$$\vec{x} = \begin{bmatrix} \psi_{s\alpha} \\ \psi_{s\beta} \\ \psi_{r\alpha} \\ \psi_{r\beta} \end{bmatrix}$$

$$\vec{u} = \begin{bmatrix} u_{s\alpha} \\ u_{s\beta} \end{bmatrix}$$

$$L = \begin{bmatrix} L_s & 0 & L_m & 0 \\ 0 & L_s & 0 & L_m \\ L_m & 0 & L_r & 0 \\ 0 & L_m & 0 & L_r \end{bmatrix} \quad R = \begin{bmatrix} R_s & 0 & 0 & 0 \\ 0 & R_s & 0 & 0 \\ 0 & 0 & R_r & 0 \\ 0 & 0 & 0 & R_r \end{bmatrix}$$

$$\Omega = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\omega_r \\ 0 & 0 & \omega_r & 0 \end{bmatrix} \qquad A = -R \cdot L^{-1} + \Omega$$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad C = L^{-1}_{1..4:1..2}$$

Gemensamt för de bägge beskrivningarna ovan är att de uttrycker asynkronmaskinens ekvationer i kontinuerlig tid. För att kunna reglera maskinen med en dator måste vi ta fram en diskret modell.

I den regulator som vi har byggt upp använder vi en diskret modell för att beräkna systemets tillstånd i varje sampeltidpunkt. Eftersom den kontinuerliga modellen är tidsvariabel har vi infört en förenkling av diskretiseringen. Normalt diskretiserar man ett system enligt följande:

$$\vec{x}_{k+1} = \Phi \cdot \vec{x}_k + \Gamma \cdot \vec{u}_k$$

$$\vec{y}_k = C \cdot \vec{x}_k$$

Där

$$\Phi = e^{A \cdot T_s}, \Gamma = \int_0^{T_s} B e^{A\tau} d\tau$$

För att slippa beräkna Φ respektive Γ för varje gång i reglerloopen, förenklar vi dem till:

$$\Phi = e^{(-R \cdot L^{-1} + \Omega)T_s} \simeq e^{\Omega T_s/2} \Phi_0 e^{\Omega T_s/2}$$

Där

$$\Phi_0 = e^{-R \cdot L^{-1} T_s}$$

Samt

$$\Gamma = e^{\Omega T_s/2} \Gamma_0 e^{\Omega T_s/2}$$

Där

$$\Gamma_0 = \int_0^{T_s} B e^{-R \cdot L^{-1} \tau} d\tau$$

Att detta innebär en förenkling inses lätt när man ser strukturen på matrisen

$$e^{\Omega T_s/2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos(\omega_r T_s/2) & -\sin(\omega_r T_s/2) \\ 0 & 0 & \sin(\omega_r T_s/2) & \cos(\omega_r T_s/2) \end{bmatrix}$$

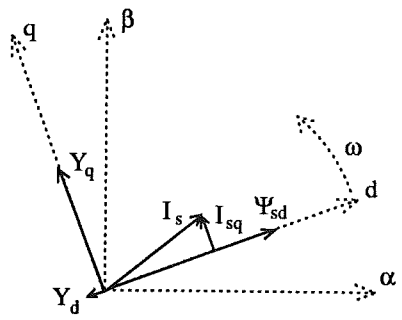
På detta vis reduceras beräkningen av matrisexponentialfunktionen till beräkning av sinus och cosinus för en skalär samt några enkla matrismultiplikationer.

6.3 Vektorstyrning

Den regulator som vi byggt upp är en så kallad vektorstyrning. Grunden till namnet är (givetvis) att man i en sådan styr/reglerar motorns storheter på vektorform. Denna form av styrning används i praktiskt taget alla moderna styrutrustningar för olika former av växelströmsmotorer.

Principen är att man bildar en vektorrepresentation av de strömmar, spänningar och flöden man är intresserad av. Detta utförs med en tre- till tvåfastransformation. När man har storheterna på vektorform kan man genom en vektortransformation välja ett lämpligt koordinatsystem, där man kan applicera traditionell reglerteori.

Den transformation som vi använt för omvandling mellan trefasstorheter och och vektorstorheter i den uppbyggda regulatorn är den effektinvarianta $2 \leftrightarrow 3$ -fas transformationen. Den ser ut som följer:



Figur 6.5: Vektorerna i maskinen, såsom de ses från ström- och flödesregulatorerna.

$$s_\alpha = s_a\sqrt{\tfrac{3}{2}} \qquad s_a = s_\alpha\sqrt{\tfrac{2}{3}}$$

$$s_\beta = (2s_b - s_a)\sqrt{\tfrac{3}{2}} \qquad s_b = s_\beta\sqrt{\tfrac{3}{2}} - s_\alpha\tfrac{1}{\sqrt{6}}$$

$$s_c = -s_\beta\sqrt{\tfrac{3}{2}} - s_\alpha\tfrac{1}{\sqrt{6}}$$

Förutom dessa transformationer har vi användning för koordinatvridningar. Dessa används för att transformera vektorstorheter mellan det statorkoordinater och flödeskoordinater. Vinkeln mellan dessa två koordinatsystem betecknas θ. Koordinattransformationerna blir då som följer:

$$s_d = s_\alpha\cos\theta + s_\beta\sin\theta \qquad s_\alpha = s_d\cos\theta - s_q\sin\theta$$
$$s_d = s_\beta\cos\theta - s_\alpha\sin\theta \qquad s_\beta = s_d\sin\theta + s_q\cos\theta$$

6.4 Regulatorn

Asynkronmaskinsregulatorn arbetar med en skattning av tillstånden i motorn. Regulatorn får in mätvärden på statorström, statorspänning och rotorvinkelhastighet. Dessa mätvärden används för att uppdatera en fullständig maskinmodell. Maskinmodellens skattade tillstånd används sedan för själva regleringen av maskinen.

6.4.1 Flödesreglering

Reglerstrategin bygger på att man först beräknar flödeskoordinatsystemet (d-q) (egentligen transformationen mellan statorkoordinater och flödeskoordinater) och transformerar statorflödesvektorn respektive statorströmvektorn till detta koordinatsystem.

$$\hat{\psi}_{sd(k)} = \sqrt{\hat{\psi}^2_{s\alpha(k)} + \hat{\psi}^2_{s\beta(k)}}$$

$$\psi_{sq(k)} \equiv 0$$

$$\theta = \arccos\left(\frac{\hat{\psi}_{s\alpha(k)}}{\hat{\psi}_{sd(k)}}\right)$$

I detta koordinatsystem återfinns statorflödesvektorn längs ena koordinataxeln. När man beräknat detta kan man helt enkelt applicera en vanlig PI-regulator på statorflödets storlek.

$$Y^*_{d(k+1)} = K_{p\psi}\left[\left(\psi^*_{sd(k)} - \hat{\psi}_{sd(k)}\right) + \sum_{l=0}^{k}\frac{T_{i\psi}}{T_s}\left(\psi^*_{sd(l)} - \hat{\psi}_{sd(l)}\right)\right]$$

Regulatorn får in börvärde och ärvärde i storheten voltsekunder och skall sedan styra en strömriktares spännings-tidyta under ett sampelintervall, alltså i voltsekunder. Regulatorn styr i samma koordinatsystem längs samma koordinataxel som statorflödesvektorn.

6.4.2 Momentreglering

Denna del av regulatorn styr hur stor momentbildande ström som går genom motorn. Momentet bildas av den strömkomponent som är vinkelrät mot statorflödet. Eftersom vi redan beräknat det koordinatsystem som har statorflödesvektorn längs ena axeln, behöver vi bara transformera statorströmmen till samma koordinatsystem.

$$i_{sq} = i_{s\beta}\cos\theta - i_{s\alpha}\sin\theta$$

$$Y^*_{q(k+1)} = K_{pI}\left[\left(i^*_{sq(k)} - i_{sq(k)}\right) + \sum_{l=0}^{k}\frac{T_{iI}}{T_s}\left(i^*_{sq(l)} - i_{sq(l)}\right)\right]$$

Strömregulatorn styr sedan ut motorspänningen vinkelrätt mot flödesvektorn. Det kan vara värt att notera att regulatorns P-del inte är dimensionslös. K_{pI} har dimensionen Vs/A vilket är detsamma som enheten Henry för induktans. Man kan visa att man kan parametrisera förstärkningen K_{pI} i summan av rotorn och statorns läckinduktanser samt en dimensionslös del, som då får det ungefärliga värdet ett.

I den givna formuleringen av strömregulatorn ovan har integraldelen en mycket viktig uppgift, nämligen att kompensera för motorns rotation. Rotationen medför att man måste använda en hel del spänning i tvärled bara för att hålla farten uppe. Om rotorvinkelhastigheten är känd, kan man införa en framkoppling till denna regulator, på följande sätt.

$$Y^*_{q(k+1)} = K_{pI}\left[\left(i^*_{sq(k)} - i_{sq(k)}\right) + \sum_{l=0}^{k}\frac{T_{iI}}{T_s}\left(i^*_{sq(l)} - i_{sq(l)}\right)\right] + \omega_r\psi_d T_s$$

Med denna framkoppling behöver integraldelen inte arbeta så hårt med att kompensera för rotationshastigheten utan kan optimeras för ett bra stegsvar.

6.4.3 Sammansättning

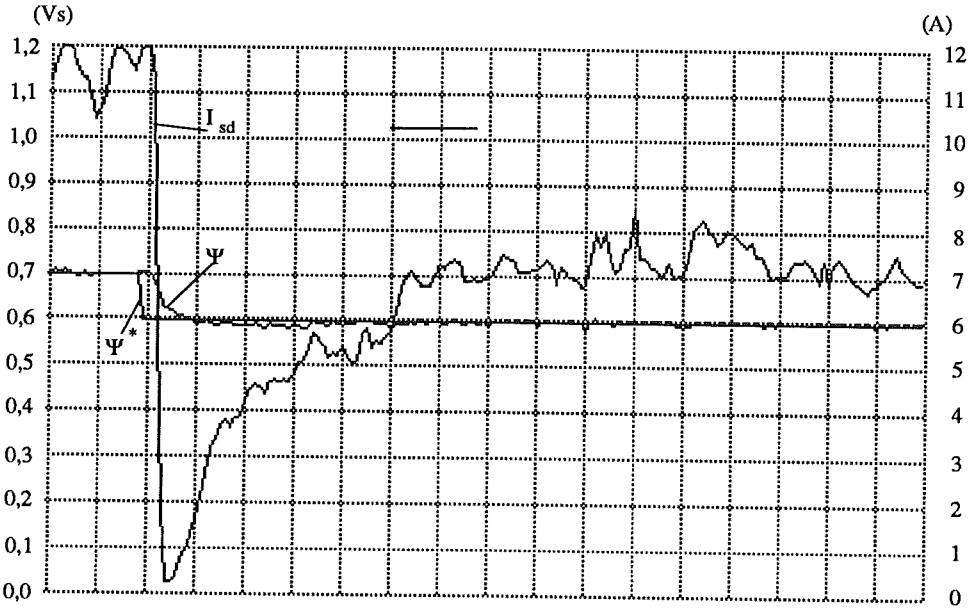
När man sedan skall styra ut sin strömriktare börjar man med att transformera de bägge regulatorernas styrvärden (flödesregulatorns längs flödet, strömregulatorns vinkelrätt mot flödesvektorn) till statorkoordinater.

$$Y^*_{\alpha(k+1)} = Y^*_{d(k+1)}\cos\theta - Y^*_{q(k+1)}\sin\theta$$

$$Y^*_{\beta(k+1)} = Y^*_{d(k+1)}\sin\theta + Y^*_{q(k+1)}\cos\theta$$

Sedan vrids styrvektorn fram ytterligare en bit, för att anpassa styrvektorn till var flödesvektorerna befinner sig då vi lägger ut de beräknade styrvärdena.

Därefter transformerar vi styrvärdena till trefasvärden och lägger ut dem som börvärden till strömriktarens styrelektronik.



Figur 6.6: Flödesbörvärde, statorflöde och statorström i d-led vid ett steg i flödesbörvärdet.

6.5 Prestanda

Slutligen vill jag presentera några mätresultat som visar regulatorns prestanda. Samtliga figurer har en skala på 10 millisekunder per ruta.

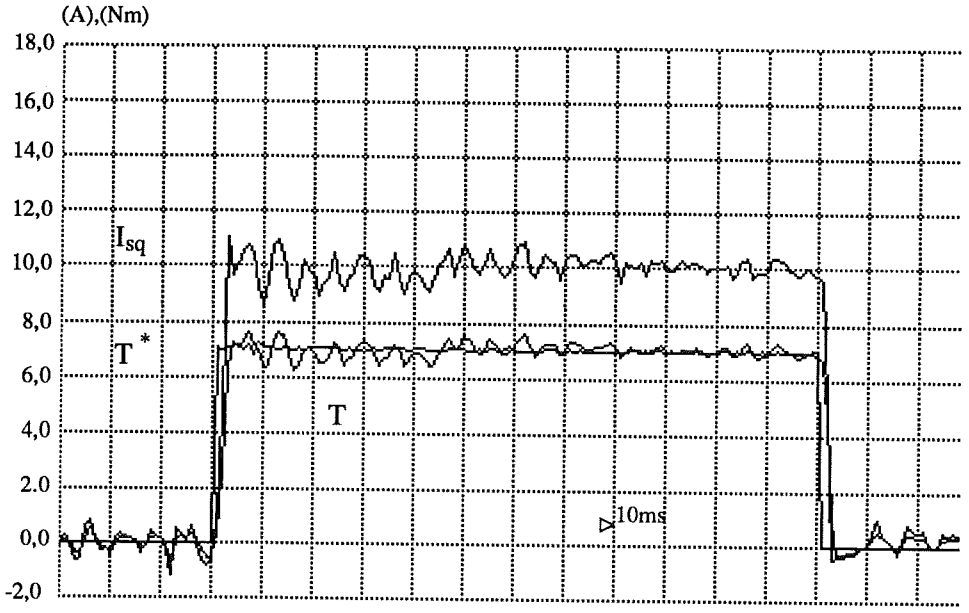
Figur 6.6 visar vad som händer vid en stegvis förändring av flödesbörvärdet (magnetisering). I figuren kan man se att det tar ungefär fem millisekunder för flödet att nå sitt nya värde. Detta är dock inte det intressanta. I figuren syns också statorströmmens komponent i d-led. Dess förlopp är typiskt för den form av reglering som används.

Enligt Mats reglerar många styrutrustningar strömkomponenten i stället för att direkt reglera flödet, och de får på det viset en långsammare förändring av maskinens magnetisering.

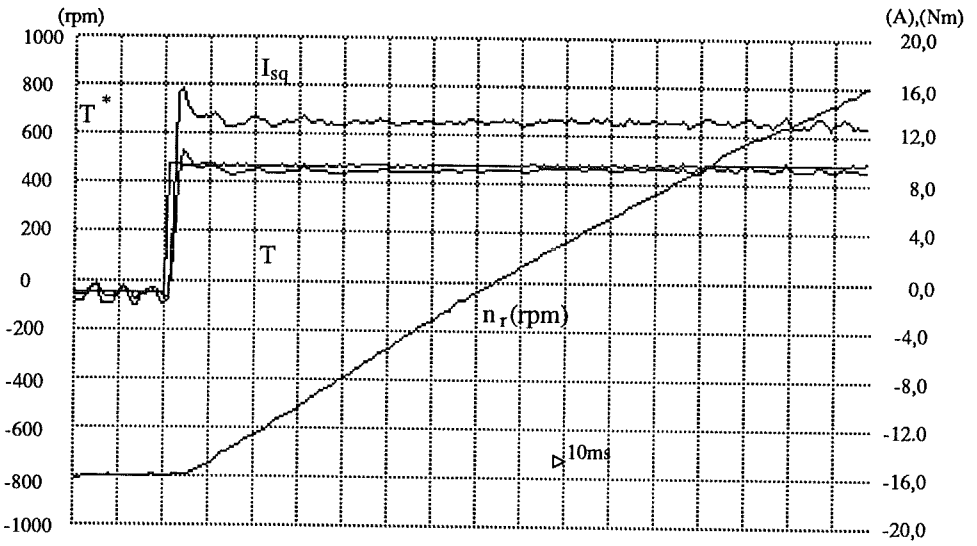
Figur 6.7 visar ett steg i momentbildande ström, motsvarande halvt märkmoment för motorn, vid konstant varvtal. Här ser man att det tar 2 millisekunder från börvärdesförändringen tills momentet ställt in sig. Detta med en millisekunds sampelintervall.

Figur 6.8 Visar ett momentsteg från något under noll till nära fullt moment, utan bromsning av varvtalet. I idealfallet skall detta resultera i en helt linjär varvtalsramp, något som regulatorn nästan lyckas med. Framför allt saknas den utplaning kring nollvarvtalet som en frekvensstyrning, eller en regulator med en enklare flödesskattning, hade gett.

Tyvärr har den varvtalsmätning som använts varit ganska bristfällig. Detta har medfört att varvtalsökningen avtar något efter att ha passerat nollpunkten. Detta har mera att göra med varvtalsmätningen i sig än regulatorns beroende av varvtalsmätningen.



Figur 6.7: Vridmoment, statorström i q-led (momentbildande ström), och "momentbörvärde" vid ett steg i börvärdet för statorström i q-led och konstant varvtal.



Figur 6.8: Varvtal, vridmoment, momentbildande ström, och momentbörvärde vid ett steg i börvärdet på momentbildande ström och frilöpande rotor.

Litteraturförteckning

[1] Brian W. Kernighan, Dennis M. Richie. *The C Programming Language, 2:nd edition* Prentice-Hall, Englewood Cliffs, N.J, 1988

[2] James L. peterson, Abraham Silberschatz *Operating Systems Concepts, second edition* Addison-Wesley, 1985

[3] Boris Magnusson,Eva Magnusson m.fl. *Realtidsprogrammering*, Institutionen för Datalogi och Numerisk analys, 1988

[4] Karl-Johan Åström, Björn Wittenmark. *Computer Controlled Systems* Prentice-Hall International edition, 1990

[5] Gustaf Olsson m.fl *Elmaskinsystem* Institutionen för Industriell Elektroteknik och Automation, 1992

[6] Mats Alaküla, Gustaf Olsson, Tore Svensson *Styrning Av Elektriska Drivsystem* Institutionen för Industriell Elektroteknik och Automation, 1991

Bilaga A

Implementation av asynkronmaskinregulatorn

A.1 Signalprocessorprogram

Programmet för signalprocessorn består av tre trådar. Dels huvudprogrammet, representerat av funktionen `main`, dels asynkronmaskinregulatorn, samt dessutom en varvtalsregulator. Dessa tre trådar kommunicerar med varandra huvudsakligen genom globala variabler. Detta är inte speciellt "vackert" ur realtidssynpunkt, men det fungerar i detta fall eftersom det inte är fråga om variabler som flera processer modifierar; informationsflödena är enkelriktade.

A.1.1 Huvudprogram – uppstart och kommunikation

Huvudprogrammet börjar med att signalera till LabVIEW när funktionen main anropas. Därefter lägger sig programmet och väntar på en signal från LabWIEW, som talar om att programmet kan starta. När LabVIEW har signalerat till signalprocessorn, startas initieringen av regulatorerna. När regulatorerna är startade skapas en klockhändelse som inträffar 100 ggr/sekund, och programmet går in i den huvudloop som överför data mellan signalprocessorn och LabVIEW. Jämför med LabVIEW-programmet.

```
/*
 * main.c
 *
 * Denna fil innehåller huvudprogrammet, som sköter om uppstartsförlopp,
 * samt överföring av data mellan LabVIEW och DSP:n
 */

#include <peripherials:toIEEEtoC30.h>
#include <peripherials:shared_data.h>
#include <peripherials:shared_variables.h>
#include <kern:time.h>
#include <kern:sys_clock.h>
#include <kern:timer_event.h>
#include <libc:stdio:stdio.h>
#include <math.h>
#include "AMcontroller.h"
#include "SpeedController.h"

void main() {
   event_t      mainTimer;
```

```
timevalue   firsttime,interval;
IEEEfloat   fpInput;
long       intInput;
float       Kp,Ti,x,fpOutput;

/*
 * Signalera till LabVIEW att kärnan har bootat färdigt.
 */
shared_variable_give(0);

/*
 * Vänta tills LabVIEW har överfört alla omvandlingsfaktorer
 */
while (shared_variable_read(1) == 0);

/*
 * Hämta Omvandlingsfaktorerna
 */
read_shared_data(20,fpInput,IEEEfloat);
Ky = IEEEtoC30(fpInput);
read_shared_data(21,fpInput,IEEEfloat);
Kpsi = IEEEtoC30(fpInput);
read_shared_data(22,fpInput,IEEEfloat);
Ki = IEEEtoC30(fpInput);
read_shared_data(23,fpInput,IEEEfloat);
Ku = IEEEtoC30(fpInput);
read_shared_data(24,fpInput,IEEEfloat);
Komega = IEEEtoC30(fpInput);

/*
 * Starta regulatorerna
 */
AMControllerInit();

SpeedControllerInit();

gettimeofday(&firsttime);

interval.tv_sec = 0;
interval.tv_usec = 100000; /* 100 ms */

timeradd(&firsttime,&interval);
mainTimer = create_timer_event(NULL,&firsttime,&interval);

do {
   event_wait(mainTimer,0);


   /*
    * Lös in börvärden från LabVIEW
    */

   read_shared_data(0,fpInput,IEEEfloat);
   Man_Isp = IEEEtoC30(fpInput);

   read_shared_data(1,fpInput,IEEEfloat);
   SetFluxSetpoint(IEEEtoC30(fpInput));
```

```
read_shared_data(2,fpInput,IEEEfloat);
OmegaSP = IEEEtoC30(fpInput);

read_shared_data(3,intInput,long);
automatic_mode = (intInput != 0) ? FALSE : TRUE;

read_shared_data(4,intInput,long);
use_observer = (intInput != 0) ? FALSE : TRUE;


/*
 * Ls in regulatorparametrar
 */
read_shared_data(10,fpInput,IEEEfloat);
Kp = IEEEtoC30(fpInput);
read_shared_data(11,fpInput,IEEEfloat);
Ti = IEEEtoC30(fpInput);
SetCurrentControllerParams(Kp,Ti);

read_shared_data(12,fpInput,IEEEfloat);
Kp = IEEEtoC30(fpInput);
read_shared_data(13,fpInput,IEEEfloat);
Ti = IEEEtoC30(fpInput);
SetFluxControllerParams(Kp,Ti);

read_shared_data(14,fpInput,IEEEfloat);
Kp = IEEEtoC30(fpInput);
read_shared_data(15,fpInput,IEEEfloat);
Ti = IEEEtoC30(fpInput);
SetSpeedControllerParams(Kp,Ti);

/*
 * Hämta Omvandlingsfaktorerna
 */
read_shared_data(20,fpInput,IEEEfloat);
Ky = IEEEtoC30(fpInput);
read_shared_data(21,fpInput,IEEEfloat);
Kpsi = IEEEtoC30(fpInput);
read_shared_data(22,fpInput,IEEEfloat);
Ki = IEEEtoC30(fpInput);
read_shared_data(23,fpInput,IEEEfloat);
Ku = IEEEtoC30(fpInput);
read_shared_data(24,fpInput,IEEEfloat);
Komega = IEEEtoC30(fpInput);

/*
 * Skriv ut plotdata, först alla i alfa-beta koordinater
 */
write_shared_data(0,C30toIEEE(PsiS_beta),IEEEfloat);
write_shared_data(1,C30toIEEE(PsiS_alfa),IEEEfloat);
write_shared_data(2,C30toIEEE(PsiS_beta_hat),IEEEfloat);
write_shared_data(3,C30toIEEE(PsiS_alfa_hat),IEEEfloat);
write_shared_data(4,C30toIEEE(PsiR_beta_hat),IEEEfloat);
write_shared_data(5,C30toIEEE(PsiR_alfa_hat),IEEEfloat);
write_shared_data(6,C30toIEEE(IS_beta),IEEEfloat);
write_shared_data(7,C30toIEEE(IS_alfa),IEEEfloat);
write_shared_data(8,C30toIEEE(IS_beta_hat),IEEEfloat);
write_shared_data(9,C30toIEEE(IS_alfa_hat),IEEEfloat);
```

```
        write_shared_data(10,C30toIEEE(IR_beta_hat),IEEEfloat);
        write_shared_data(11,C30toIEEE(IR_alfa_hat),IEEEfloat);
        write_shared_data(12,C30toIEEE(Y_beta),IEEEfloat);
        write_shared_data(13,C30toIEEE(Y_alfa),IEEEfloat);

        /*
         * Sedan alla i D-Q koordinater (d.v.s koordinater relativt huvudflödet)
         */
        write_shared_data(20,C30toIEEE(PsiS_q_qsc),IEEEfloat);
        write_shared_data(21,C30toIEEE(PsiS_d_qsc),IEEEfloat);
        write_shared_data(22,C30toIEEE(PsiS_q_hat),IEEEfloat);
        write_shared_data(23,C30toIEEE(PsiS_d_hat),IEEEfloat);
        write_shared_data(24,C30toIEEE(PsiR_q_hat),IEEEfloat);
        write_shared_data(25,C30toIEEE(PsiR_d_hat),IEEEfloat);
        write_shared_data(26,C30toIEEE(IS_q_filt),IEEEfloat);
        write_shared_data(27,C30toIEEE(IS_d_filt),IEEEfloat);
        write_shared_data(28,C30toIEEE(IS_q_hat),IEEEfloat);
        write_shared_data(29,C30toIEEE(IS_d_hat),IEEEfloat);
        write_shared_data(30,C30toIEEE(IR_q_hat),IEEEfloat);
        write_shared_data(31,C30toIEEE(IR_d_hat),IEEEfloat);
        write_shared_data(32,C30toIEEE(Yq),IEEEfloat);
        write_shared_data(33,C30toIEEE(Yd),IEEEfloat);

        /*
         * RMS-värden på spänning ström och flöde, samt varvtalet
         */
        write_shared_data(40,C30toIEEE(sqrt(Yq*Yq + Yd*Yd)/Tsamp),IEEEfloat);
        write_shared_data(41,C30toIEEE(sqrt(IS_d_filt*IS_d_filt +
                                        IS_q_filt*IS_q_filt)),IEEEfloat);
        write_shared_data(42,C30toIEEE(PsiS_d),IEEEfloat);
        write_shared_data(43,C30toIEEE(OmegaR),IEEEfloat);

        /*
         * CPU load average kan också vara intressant att kunna visa
         */
        write_shared_data(44,C30toIEEE(100.0*cpu_load_average()),IEEEfloat);

    } while (1);
}
```

A.1.2 Reglerloop – Flöde och moment

asynkronmaskinsregulatorn implementeras av den följande tråden, med vidhängande globala variabler etc.

Regleringen utförs i en slinga av avsevärd längd, uppdelad i flera deluppgifter. Mätvärden tas in en gång per millisekund, därefter beräknas styrsignaler, som uppdateras två gånger per millisekund.

Beräkningen av observatören har delats upp i två delar eftersom större delen av beräkningsarbetet kan utföras i förväg. Detta sparar en del tid i den kritiska delen mellan mätning och den första uppdateringen.

```
/*
 * AMcontroller.c
 *
 * Denna fil innehåller den tråd som sköter insamling och enhetsomvandling
 * av mätdata, beräkning av ledvärden utifrån estimerade flöden
 * samt enhetsomvandling och utmatning av nya ledvärden
 */

#include <kern:thread.h>
#include <kern:time.h>
#include <kern:timer_event.h>
#include <drivers:nb_mio_16.h>
#include <drivers:nb_ao_6.h>
#include <math.h>
#include "config.h"
#include "machine_constants.h"
#include "AMcontroller.h"

/*
 * Några konstanter som vi har användning för
 */

#define Sqrt_three_half   1.224744871
#define Sqrt_one_half     0.7071067812
#define Sqrt_two_thirds   0.8164965809
#define Sqrt_three        1.732050808

#ifndef PI
#define PI   3.1415926535
#endif

/*
 * Omvandlingsfaktorer när vi hämtar dem från LabVIEW.
 */
#undef Ky
#undef Kpsi
#undef Ki
#undef Ku
#undef Komega

float   Ky,Kpsi,Ki,Ku,Komega;

/*
 * Uppmätta spänningar/flöden
 */
float   PsiS_sp = 0.2;
float   PsiS_alfa = 0.0;
```

```
float   PsiS_beta = 0.0;
float   PsiS_d_qsc = 0.0;
float   PsiS_q_qsc = 0.0;

float   PsiS_d = 0.0;

/*
 * U-t ytor
 */
float   Y_alfa = 0.0,Y_beta = 0.0;
float   Yd,Yq;

/*
 * Parametrar till flödesregulatorn
 */
float   Kp_psi = 0.2;
float   Ti_psi,Tr_psi,Ki_psi = 0.0,Kr_psi = 0.0;

/*
 * Statorström i alfa-beta och d-q -led
 */
float   IS_alfa,IS_beta;
float   IS_d,IS_q,IS_q_sp = 0.0;
float   IS_d_filt = 0.0,IS_q_filt = 0.0;
/*
 * Parametrar till tvärströmsregulatorn
 */
float   Kp_I = 0.0;
float   Ti_I,Tr_I,Ki_I = 0.0,Kr_I = 0.0;

/*
 * Skattade Stator och rotorflöden, även dessa både i alfa-beta och d-q -led
 */
float   PsiS_alfa_hat = 0.1,PsiS_beta_hat = 0.1;
float   PsiS_d_hat = 0.1,PsiS_q_hat = 0.1;
float   PsiR_alfa_hat = 0.1,PsiR_beta_hat = 0.1;
float   PsiR_d_hat = 0.1,PsiR_q_hat = 0.1;

/*
 * Skattade stator och rotorströmmar
 * både i alfa-beta och d-q led
 */
float   IS_alfa_hat,IS_beta_hat,IR_alfa_hat,IR_beta_hat;
float   IS_d_hat,IS_q_hat,IR_d_hat,IR_q_hat;

/*
 * Övrigt
 */
float   Sin_x = 0.0,Cos_x = 0.0;
float   Ud;
float   OmegaR = 0.0;
float   Tsamp;   /* samplingsintervall i sekunder */
float   faktor1,faktor2;

boolean_t   use_observer = TRUE;

mio_16   inboard;
ao_6   outboard;
```

```
event_t   controlTimer;

void UpdateOutputs(float Y_alfa,float Y_beta) {
   register float Ya,Yb,Yc,Ymax,Ymin,Yz;

   /*
    * 2 till 3-fas omvandling, samt skalning med
    * omvandlingsfaktorer
    */
   Ya = Y_alfa * Sqrt_two_thirds/Ky;
   Yb = (Sqrt_three*Y_beta - Y_alfa)*Sqrt_two_thirds/2.0/Ky;
   Yc = (0.0 - Sqrt_three*Y_beta - Y_alfa)*Sqrt_two_thirds/2.0/Ky;

   /*
    * Maximera utnyttjandet av strömriktaren - gör
    * "rumpsinus" av ledvärdena.
    */
   Ymax = (Ya > Yb) ? Ya : Yb;
   Ymax = (Ymax < Yc) ? Yc : Ymax;
   Ymin = (Ya < Yb) ? Ya : Yb;
   Ymin = (Ymin > Yc) ? Yc : Ymin;
   Yz = (Ymax + Ymin) / 2.0 ;

   /*
    * Förbered uppdatering
    */
   ao_6_analog_out(outboard,AO_6_chan0,Ya-Yz);
   ao_6_analog_out(outboard,AO_6_chan1,Yb-Yz);

   /*
    * Vänta på uppdateringstidpunkten
    */
   event_wait(controlTimer,0);
   /*
    * Uppdatera utgångarna till nya ledvärden.
    */
   ao_6_analog_out(outboard,AO_6_chan2|AO_6_update,Yc-Yz);
}

void Controller_thread() {

   /*
    * Förberedelser
    */

   float   PsiS_int = 0.0; /* PsiS_int Måste vara noll vid start */
   float   PsiS_d_inv;

   float   Cos_theta,Sin_theta;
   float   Sin_2x,Cos_2x;

   float   IS_qint = 0.0;   /* Integraldel till strömregulatorn */

   float   Yd_sp = 0.0,Yq_sp = 0.0,Y_max = 0.0,Yd_max = 0.0;
   float   Y_alfa1,Y_beta1,Y_alfa2,Y_beta2;
   float   Yq_filt = 0.0;
```

```
float   OmegaR_old = 0.0;
int     OmegaR_cut_count = 0;

float   Sin_omegaR,Cos_omegaR,Sin_2omegaR,Cos_2omegaR;

long   channels1[] = {0,1,2,3,4};
struct {
   float PsiS_alfa0,
         PsiS_beta0,
         Ia,
         Ib;
   float OmegaR;
} samples1;

long channels2[] = {8,5};
struct {
   float   OmegaR;
   float   Ud;
} samples2;

float   US_alfa_hat,US_beta_hat,UR_alfa_hat,UR_beta_hat;
float   PsiS_alfa_hat_old = 0.0;
float   PsiS_beta_hat_old = 0.0;
float   PsiR_alfa_hat_old = 0.0;
float   PsiR_beta_hat_old = 0.0;
float   IS_alfa_old = 0.0;
float   IS_beta_old = 0.0;
float   PsiS_alfa_old = 0.0;
float   PsiS_beta_old = 0.0;

do {
   /*
    * Läs in Analoga värden
    */
    mio_16_analog_in(inboard,channels1,5,(float *) &samples1);

   /*
    * Normalisera till samma sampeltidpunkt, samt räkna om
    * till reella enheter (Vs,A,rad/s)
    * Strömmen har även en Tre- till Tvåfasomvandling.
    */
   PsiS_alfa = Kpsi * samples1.PsiS_alfa0; /*+ 0.50*Y_alfa2;*/
   PsiS_beta = Kpsi * samples1.PsiS_beta0; /*+ 0.50*Y_beta2;*/
   IS_alfa = samples1.Ia * Ki * Sqrt_three_half;
   IS_beta = Sqrt_one_half * Ki * (samples1.Ia + 2*samples1.Ib);

   /*
    * Uppdatera flödesobservatören, men gör bara det nödvändigaste
    * eftersom vi troligen har ont om tid.
    */
   US_alfa_hat = PsiS_alfa*faktor2 - PsiS_alfa_old*faktor1;
   US_beta_hat = PsiS_beta*faktor2 - PsiS_beta_old*faktor1;
   PsiS_alfa_hat += Gobs_11*US_alfa_hat + Komp_11*IS_alfa;
   PsiS_beta_hat += Gobs_22*US_beta_hat + Komp_22*IS_beta;

   /* Beräkna d-q koordinatsystemet */
   if (use_observer) {
      PsiS_d = sqrt(PsiS_alfa_hat*PsiS_alfa_hat +
```

```
                PsiS_beta_hat*PsiS_beta_hat);
   PsiS_d_inv = 1/PsiS_d;
   Cos_theta = PsiS_d_inv * PsiS_alfa_hat;
   Sin_theta = PsiS_d_inv * PsiS_beta_hat;
} else {
   PsiS_d = sqrt(PsiS_alfa*PsiS_alfa + PsiS_beta*PsiS_beta);
   PsiS_d_inv = 1/PsiS_d;
   Cos_theta = PsiS_d_inv * PsiS_alfa;
   Sin_theta = PsiS_d_inv * PsiS_beta;
}

/* Transformera strömmen till d-q koordinater */
IS_d = IS_alfa * Cos_theta + IS_beta * Sin_theta;
IS_q = 1.5*(IS_beta * Cos_theta - IS_alfa * Sin_theta) - 0.5*IS_q;

/*
 * Filtrera strömvärdena i d-q led för
 * att undgå den värsta grisigheten
 */
IS_q_filt += 0.3*(IS_q-IS_q_filt);
IS_d_filt += 0.3*(IS_d-IS_d_filt);

/*
 * Flödes- och strömregulator:
 */

/* Beräkna önskade ledvärden */
Yd_sp = Kp_psi*((PsiS_sp - PsiS_d) + PsiS_int);
Yq_sp = Kp_I*( (( (IS_q_sp > (IS_q_max)) ? (IS_q_max) :
                 ( (IS_q_sp < -(IS_q_max)) ? -(IS_q_max):IS_q_sp) ) -
               IS_q_filt) + IS_qint);

/* Begränsa ledvärdena till tillgänglig spännings-tid-
 * yta. först i tvärled.
 */
if (Yq_sp >= 0.0)
   Yq = (Yq_sp > 0.90*Y_max) ? 0.90*Y_max : Yq_sp;
else
   Yq = (Yq_sp < Y_max*(-0.90)) ? Y_max*(-0.90) : Yq_sp;

/*
 * Räkna ut hur mycket som återstår sedan strömregulatorn
 * fått sitt och begränsa sedan ledvärdet i längsled om det behövs.
 */
Yd_max = sqrt(Y_max*Y_max - Yq*Yq);
Yd = (Yd_sp < Yd_max) ? Yd_sp : Yd_max;
Yd = (Yd > (0.0-Yd_max)) ? Yd : (0.0 - Yd_max);

/*
 * Transformera till alfa-beta koordinater
 */
Y_alfa = Yd * Cos_theta - Yq * Sin_theta;
Y_beta = Yd * Sin_theta + Yq * Cos_theta;

/*
 * Skatta hur mycket flödesvektorerna kommer att vrida sig
 * till nästa gång vi samplar. egentligen: beräkna sinus & cosinus för
 * fjärdedelen av denna vinkel samt även för halva samma vinkel.
```

```
        */
       Yq_filt += 0.08*(Yq-Yq_filt);
       Sin_x = (PsiS_d > 0.3) ? 0.125*Yq_filt/PsiS_d : 0.0;
       Cos_x = sqrt(1.0-Sin_x*Sin_x);

       /*
        * Vrid fram styrvektorn med denna fjärdedels vinkel (för att hammna
        * mitt i intervallen)
        */
       Y_alfa1 = Y_alfa * Cos_x - Y_beta * Sin_x;
       Y_beta1 = Y_alfa * Sin_x + Y_beta * Cos_x;

       /*
        * Initiera uppdatering och invänta uppdateringstidpunkten
        * (Fram hit får det högst ta Tsamp/2, annars missar vi
        * uppdateringstillfället).
        */
       UpdateOutputs(Y_alfa1,Y_beta1);

        /*
         * Filtrera mätvärdet på vinkelhastigheten OmegaR
         * Detta "filter" har som huvuduppgift att klippa bort
         * störspikar som annars skulle ställa till stora besvär
         * för regulatorer och tillståndsskattningen.
         */
#define OmegaR_eps 6.0
      {
         register float OmegaR_diff,OmegaR_hat;

         OmegaR_hat = OmegaR + 0.5*(OmegaR - OmegaR_old);
         OmegaR_old = OmegaR;

         OmegaR_diff = OmegaR_hat - Komega * samples1.OmegaR;

         if ( ((OmegaR_diff > 0) ?
               OmegaR_diff : 0.0-OmegaR_diff) > OmegaR_eps) {
            if (OmegaR_cut_count < 1) {
               /*
                * Det uppmätta värdet visar en alltför stor avvikelse
                * från skattningen. Troligtvis är det en störspik som vi
                * inte vill ha in i maskinmodellen. Vi ersätter
                * därför med skattningen i stället
                */
               OmegaR = OmegaR_hat;
            } else {
               /*
                * Hoppsan... Nu har vi redan limiterat minst en gång
                * det är nog bäst att tro *lite* på "transienten" ändå
                * Lägg alltså till 70% av maxdifferensen till skattningen
                */
               OmegaR = OmegaR_hat + (( OmegaR_diff < 0 ) ?
                                      0.7*OmegaR_eps : 0.0 - 0.7*OmegaR_eps);
            }
            OmegaR_cut_count++;

         } else {
            /*
             * Normalfallet: Det uppmätta värdet är sannolikt korrekt --
```

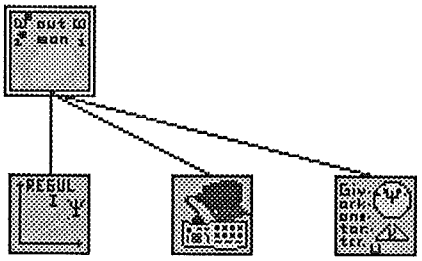
```
         * bara att använda utan vidare.
         */
        OmegaR = Komega * samples1.OmegaR;
        OmegaR_cut_count = 0;
    }
}

Ud = 300.0; /* Ku * samples2.Ud; */

/* Beräkna maximalt tillgänglig spännings-tidyta |Y|max: */
Y_max = Tsamp * Ud * Sqrt_three_half;

/*
 * Uppdatera de bägge regulatorernas integraldelar,
 * notera att vi nu kan använda skillnaden mellan önskat
 * och verkligt ledvärde till att begränsa integraldelens storlek
 */
PsiS_int += Ki_psi*(PsiS_sp - PsiS_d) + Kr_psi*(Yd - Yd_sp);
IS_qint +=  Ki_I*(IS_q_sp - IS_q) + Kr_I*(Yq -Yq_sp);


/*
 * Gör resten av uppdateringen av flödesobservatören -
 * nu bör vi nog hinna med...
 */
PsiS_alfa_old = PsiS_alfa;
PsiS_beta_old = PsiS_beta;

UR_alfa_hat = 0.0 - OmegaR*NrPoles/2*PsiR_beta_hat_old;
UR_beta_hat = OmegaR*NrPoles/2*PsiR_alfa_hat_old;

Sin_omegaR = sin(OmegaR*NrPoles/4.0*Tsamp);
Cos_omegaR = cos(OmegaR*NrPoles/4.0*Tsamp);
Sin_2omegaR = 2.0*Sin_omegaR*Cos_omegaR;
Cos_2omegaR = 1.0-2.0*Sin_omegaR*Sin_omegaR;

PsiR_alfa_hat =
  Fobs_31*Cos_omegaR*PsiS_alfa_hat_old -
    Fobs_31*Sin_omegaR*PsiS_beta_hat_old +
      Fobs_33*Cos_2omegaR*PsiR_alfa_hat_old -
        Fobs_33*Sin_2omegaR*PsiR_beta_hat_old +
          Gobs_31*US_alfa_hat + Komp_31*IS_alfa;

PsiR_beta_hat =
  Fobs_42*Sin_omegaR*PsiS_alfa_hat_old +
    Fobs_42*Cos_omegaR*PsiS_beta_hat_old +
      Fobs_44*Sin_2omegaR*PsiR_alfa_hat_old +
        Fobs_44*Cos_2omegaR*PsiR_beta_hat_old +
          Gobs_42*US_beta_hat + Komp_42*IS_beta;

/*
 * Spara undan gamla värden
 */
PsiS_alfa_hat_old = PsiS_alfa_hat;
PsiS_beta_hat_old = PsiS_beta_hat;
PsiR_alfa_hat_old = PsiR_alfa_hat;
PsiR_beta_hat_old = PsiR_beta_hat;
```

```
        /*
         * Beräkna skattningar av stator och rotorströmmar
         */
        IS_alfa_hat = C_11*PsiS_alfa_hat + C_13*PsiR_alfa_hat;
        IS_beta_hat = C_22*PsiS_beta_hat + C_24*PsiR_beta_hat;
        IR_alfa_hat = C_31*PsiS_alfa_hat + C_33*PsiR_alfa_hat;
        IR_beta_hat = C_42*PsiS_beta_hat + C_44*PsiR_beta_hat;
        /*
         * Transformera observatörens alfa-beta värden till d-q
         * koordinater.
         */
        IS_d_hat = IS_alfa_hat * Cos_theta + IS_beta_hat * Sin_theta;
        IS_q_hat = IS_beta_hat * Cos_theta - IS_alfa_hat * Sin_theta;
        IR_d_hat = IR_alfa_hat * Cos_theta + IR_beta_hat * Sin_theta;
        IR_q_hat = IR_beta_hat * Cos_theta - IR_alfa_hat * Sin_theta;
        PsiS_d_qsc = PsiS_alfa * Cos_theta + PsiS_beta * Sin_theta;
        PsiS_q_qsc = PsiS_beta * Cos_theta - PsiS_alfa * Sin_theta;
        PsiS_d_hat = PsiS_alfa_hat * Cos_theta + PsiS_beta_hat * Sin_theta;
        PsiS_q_hat = PsiS_beta_hat * Cos_theta - PsiS_alfa_hat * Sin_theta;
        PsiR_d_hat = PsiR_alfa_hat * Cos_theta + PsiR_beta_hat * Sin_theta;
        PsiR_q_hat = PsiR_beta_hat * Cos_theta - PsiR_alfa_hat * Sin_theta;

        /*
         * Förbered uppdateringen av statorflödesskattningarna
         */
        PsiS_alfa_hat = Fobs_11*PsiS_alfa_hat_old +
                        Fobs_13*Cos_omegaR*PsiR_alfa_hat_old -
                         Fobs_13*Sin_omegaR*PsiR_beta_hat_old;

        PsiS_beta_hat = Fobs_22*PsiS_beta_hat_old +
                        Fobs_24*Sin_omegaR*PsiR_alfa_hat_old +
                         Fobs_24*Cos_omegaR*PsiR_beta_hat_old;

        /*
         * Beräkna styrvärden till nästa intervall
         */

        Sin_2x = 2.0*Sin_x*Cos_x;
        Cos_2x = 1.0-2.0*Sin_x*Sin_x;

        Y_alfa2 = Y_alfa1 * Cos_2x - Y_beta1 * Sin_2x;
        Y_beta2 = Y_alfa1 * Sin_2x + Y_beta1 * Cos_2x;

        /*
         * Initiera uppdatering och invänta nästa uppdateringstidpunkt
         */
        UpdateOutputs(Y_alfa2,Y_beta2);

    } while (1); /* Så tar vi allt från början igen */
}


void to_do_atexit() {
    mio_16_reset(inboard);
    ao_6_analog_out(outboard,AO_6_chan0|AO_6_chan1|AO_6_chan2|
                    AO_6_chan3|AO_6_chan4|AO_6_chan5|AO_6_update,0.0);
}
```

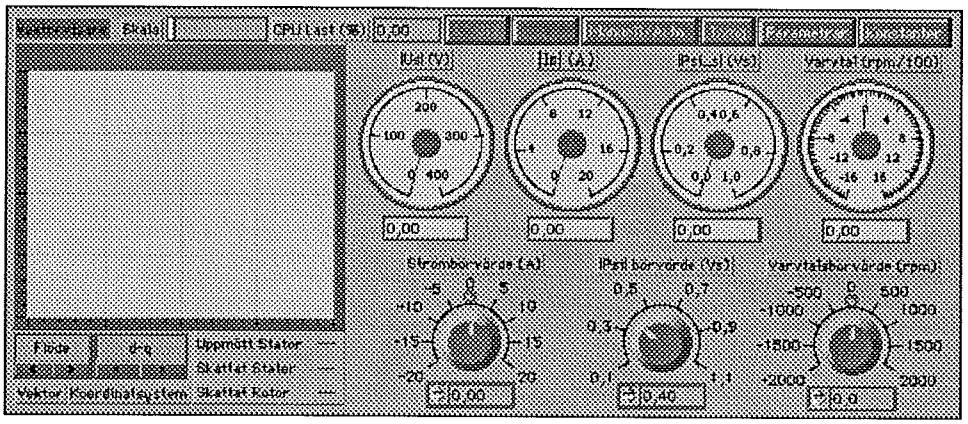
```
void AMControllerInit() {
   thread_t    th;
   timevalue   firsttime,interval;

   inboard = mio_16_init(INBOARD_SLOT);
   outboard = ao_6_init(OUTBOARD_SLOT);
   atexit(to_do_atexit);

   Tsamp = (float) SAMPLE_INTERVAL/1.0e+6;
   faktor1 = 1.0/Tsamp - 1.0/2.0/TauS;
   faktor2 = 1.0/Tsamp + 1.0/2.0/TauS;

   gettimeofday(&firsttime);

   interval.tv_sec = 0;
   interval.tv_usec = SAMPLE_INTERVAL/2;

   timeradd(&firsttime,&interval);
   controlTimer = create_timer_event(NULL,&firsttime,&interval);

   th = thread_create(Controller_thread,7);
   thread_resume(th);
}

void SetFluxControllerParams(float Kp,float Ti) {
   Kp_psi = Kp;
   Ki_psi = Tsamp/Ti;
   Kr_psi = Tsamp/Ti/2.0/Kp;
}

void SetCurrentControllerParams(float Kp,float Ti) {
   Kp_I = Kp;
   Ki_I = Tsamp/Ti;
   Kr_I = Tsamp/Ti/2.0/Kp;
}
```

A.1.3 Reglerloop – varvtal

```
/*
 * SpeedController.c
 *

#include <kern:thread.h>
#include <kern:time.h>
#include <kern:timer_event.h>
#include <math.h>
#include "SpeedController.h"
#include "AMcontroller.h"

float   OmegaSP = 0.0; /* Börvärde på rotorvinkelhastighet */
float   Kp_omega = 0.0;
float   Ki_omega = 0.0;
float   Kr_omega = 0.0;

float   Man_Isp;

boolean_t   automatic_mode;
event_t     speedCtrlTmr;

void SpeedController() {
   float Isp = 0.0;
   float Omega_int = 0.0;

   do {
      Omega_int += Kr_omega*(IS_q_filt - Isp);
      Isp = Kp_omega *((OmegaSP-OmegaR) + Omega_int);
      Omega_int += Ki_omega * (OmegaSP-OmegaR);
      IS_q_sp = (automatic_mode) ? Isp : Man_Isp;
      event_wait(speedCtrlTmr,0);
   } while (1);
}

void SpeedControllerInit() {
   thread_t    th;
   timevalue   firsttime,interval;

   gettimeofday(&firsttime);

   interval.tv_sec = 0;
   interval.tv_usec = 1e+6*SPEED_CONTROL_INTERVAL;

   timeradd(&firsttime,&interval);
   speedCtrlTmr = create_timer_event(NULL,&firsttime,&interval);

   th = thread_create(SpeedController,8);
   thread_resume(th);
}

void SetSpeedControllerParams(float Kp,float Ti) {
   Kp_omega = Kp;
   Ki_omega = SPEED_CONTROL_INTERVAL/Ti;
   Kr_omega = SPEED_CONTROL_INTERVAL/Ti/2.0/Kp;
}
```



Figur A.1: Hiearki för LabVIEW programmet som kontrollerar asynkronmaskins-regulatorn



Figur A.2: Frontpanelen på det virtuella instrument som använts för att kontrollera signalprocessorprogrammet

A.2 LabVIEWprogram

A.2.1 AMregulator

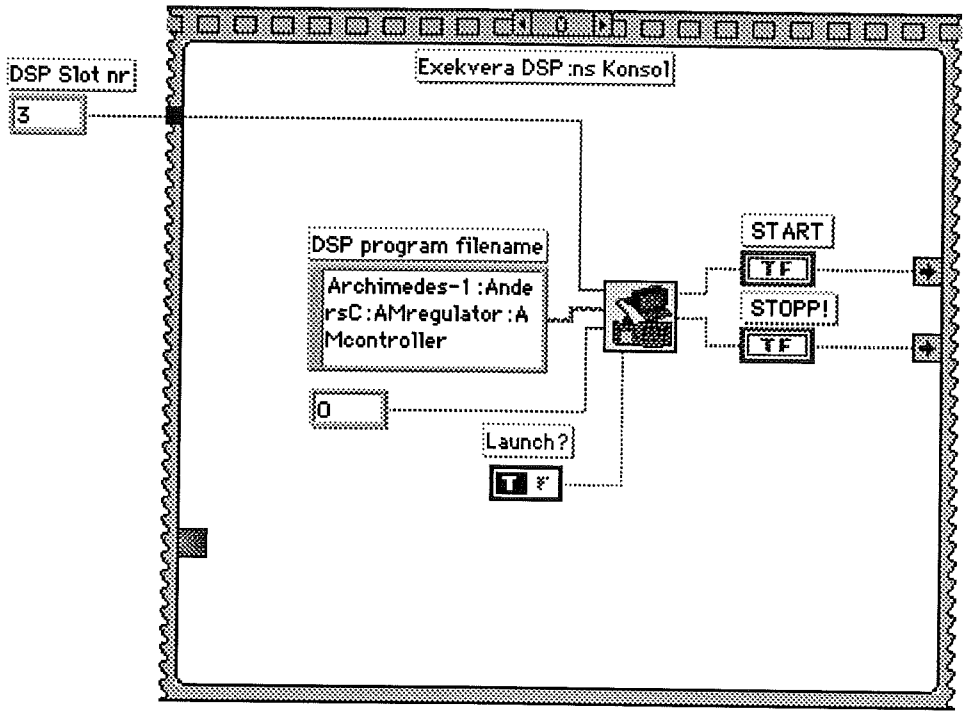
Signalprocessorn kontrolleras av ett virtuellt LabVIEW-instrument, AMregulator. Detta instrument är i sin tur en kombination av flera virtuella instrument med specifika funktioner som inte kräver ständig uppmärksamhet. Figur A.1 ger en hiearkisk bild av instrumentets uppbyggnad.

Frontpanel

Huvudinstrumentet AMregulators frontpanel innehåller en hel mängd reglage och indikatorer, se figur A.2.

Längst till vänster finns en graf som kan visa de vektorstorheter som används för regleringen av maskinen. Under denna graf finns två väljarreglage för att välja vektorgrupp och koordinatsystem. De vektorgrupper som finns är flöden, strömmar och spänningar, vektorerna kan därtill visas både i stator- (alfa-beta) och flödesko-ordinater (d-q). Denna graf är mycket illustrativ för att demonstrera hur vektor-bilden beskriver maskinens arbete, likaså för att illustrera skillnaden mellan olika reglerstrategier.

Till höger om grafen finns fyra visarinstrument som visar skalära storheter; statorspänning i Volt, statorström i Ampére, statorflöde i Voltsekunder samt axel-varvtal i rpm. De elektriska storheterna är längden av motsvarande vektorstorheter,



Figur A.3: Steg 1 i AMregulators huvudsekvens. Här anropas DSP Console.

som kan översättas till trefasstorheter med effektinvariant transformation.

Under visarinstrumenten finns det tre rattar för inställning av börvärden på momentbildande ström, statorflöde och axelvarvtal. Rattarna för momentbildande ström respektive axelvarvtal är bara aktiva en i taget, beroende på om varvtalsregulatorn är inkopplad eller ej.

Ovanför visarinstrumenten finns en rad knappar som kontrollerar operationssätt (Manuell/Auto samt Tomg), startar och stoppar signalprocessorn (START och STOPP!), samt knappar för att kalla upp instrumenten som ställer in skalfaktorer och regulatorparametrar.

Diagram

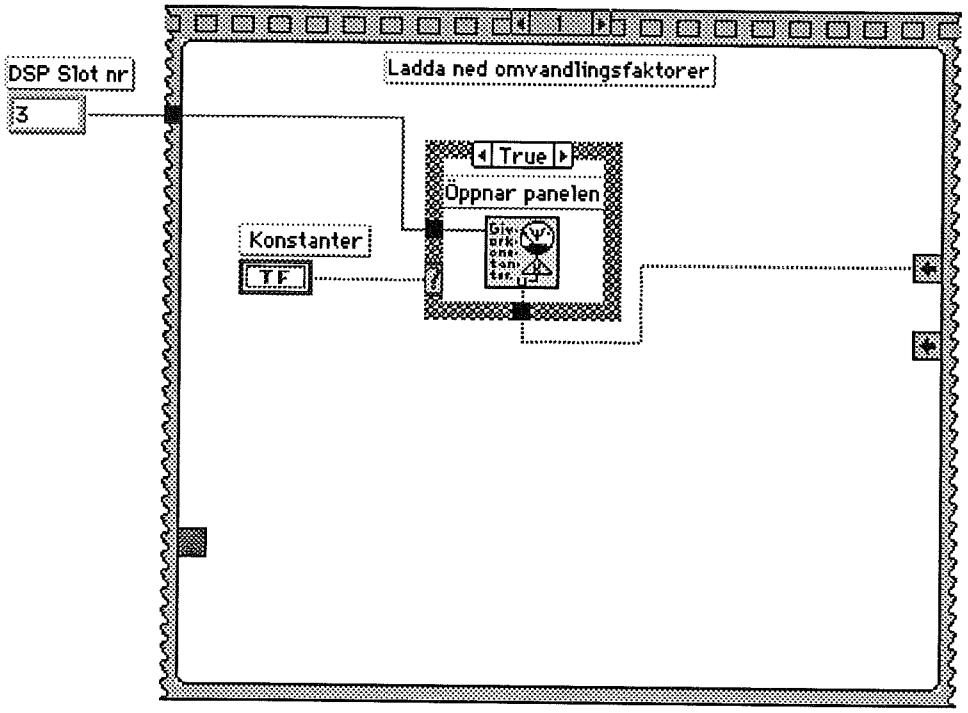
Diagrammet till instrumentet AMregulator består av en sekvensstruktur, som har sex delar, samt en numerisk konstant som anger kortplatsen för signalprocessorn.

Figur A.3 visar det första steget i sekvensen. I denna ruta finns en textkonstant som anger sökvägen till det körbara signalprocessorprogrammet och ett par andra konstanter. Dessa är argument till instrumentet "DSP Console" tillsammans med knapparna "START" och "STOPP!" som avläses här, och vars värden skickas vidare i två lokala variabler (ute till höger).

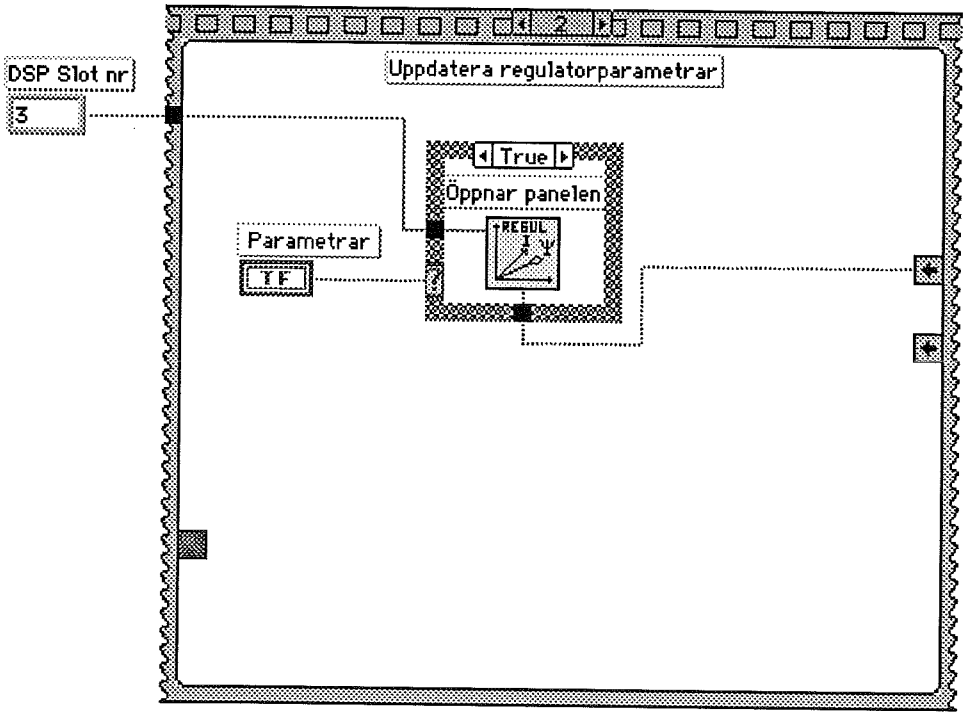
Steg två och steg tre i sekvensen är uppdatering av skalfaktorer respektive regulatorparametrar. Här ses alltså anropen av dessa instrument (figur A.4 och figur A.5).

I figur A.6 läses alla frontpanelens rattar samt resterande knappar av. Knapparna ger ifrån sig ett booleskt värde (TRUE när knappen är intryckt) som omvandlas i en villkorsstruktur till ett numeriskt värde, som kan förstås av signalprocessorprogrammet. Värdet från ratten som anger varvtalsbörvärde, omvandlas från rpm till radianer per sekund, eftersom signalprocessorprogrammet arbetar i den storheten.

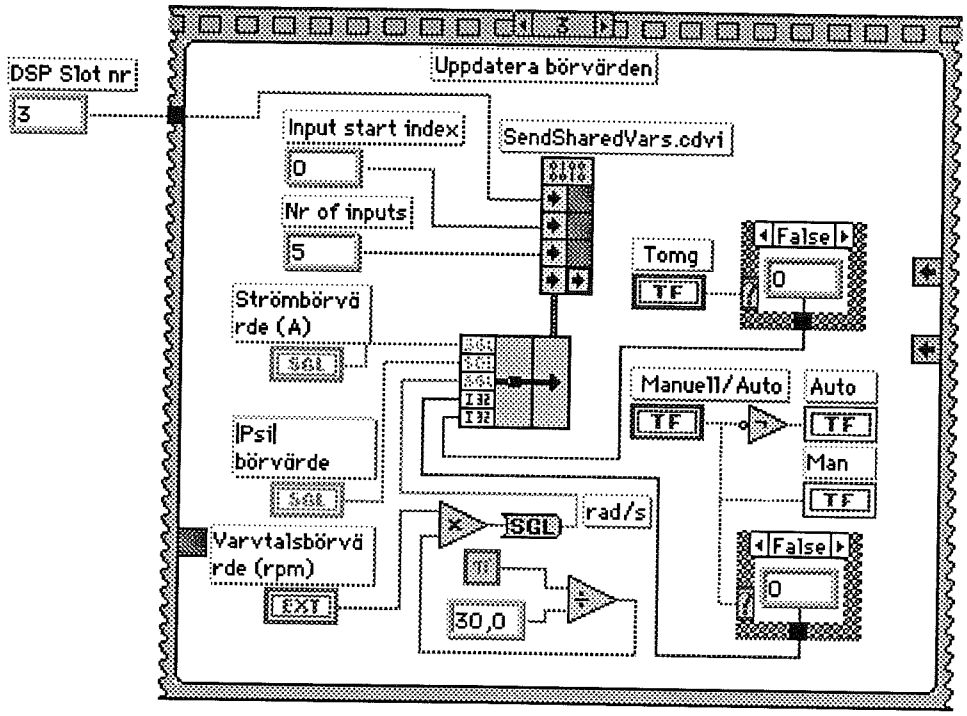
Alla värden sätts samman till ett komplex som är argument till en kodgräns-



Figur A.4: Steg 2 i AMregulators huvudsekvens. Här uppdateras skalfaktorerna.



Figur A.5: Steg 3 i AMregulators huvudsekvens. Här uppdateras regulatorparametrarna.



Figur A.6: Steg 4 i AMregulators huvudsekvens. I den här rutan sker all uppdatering av börvärden och arbetssätt.

snittsnod som skriver värdena på rätt platser i signalprocessorns minne så att signalprocessorprogrammet kan läsa av dem (Se även huvudprogrammet till signalprocessorn). Användningen av kodgränssnittsnoden beror på att de instrument för hantering av delade data som beskrivs i kapitel 5 inte existerade vid uppkomsten av detta instrument. Hade inte så varit fallet hade den ersatts med instrumentet "Write shared cluster data" som gör exakt samma sak men är säkrare att använda.

I figur A.7 hämtar en kodgränssnittsnod upp vektordata från signalprocessorn. Argumenten till den är bland annat ett index som bestämmer det koordinatsystem som vektorerna beskrivs i. indexet bestäms av en villkorskonstruktion vars väljare tas från reglaget "koordinatsystem" på instrumentets frontpanel.

För att det skall finnas någonstans att placera de data som läses måste noden förses med ett nollställt komplex. Det resulterande komplexet förs sedan vidare till nästa ruta, som konstruerar grafer av de vektorer som hämtats.

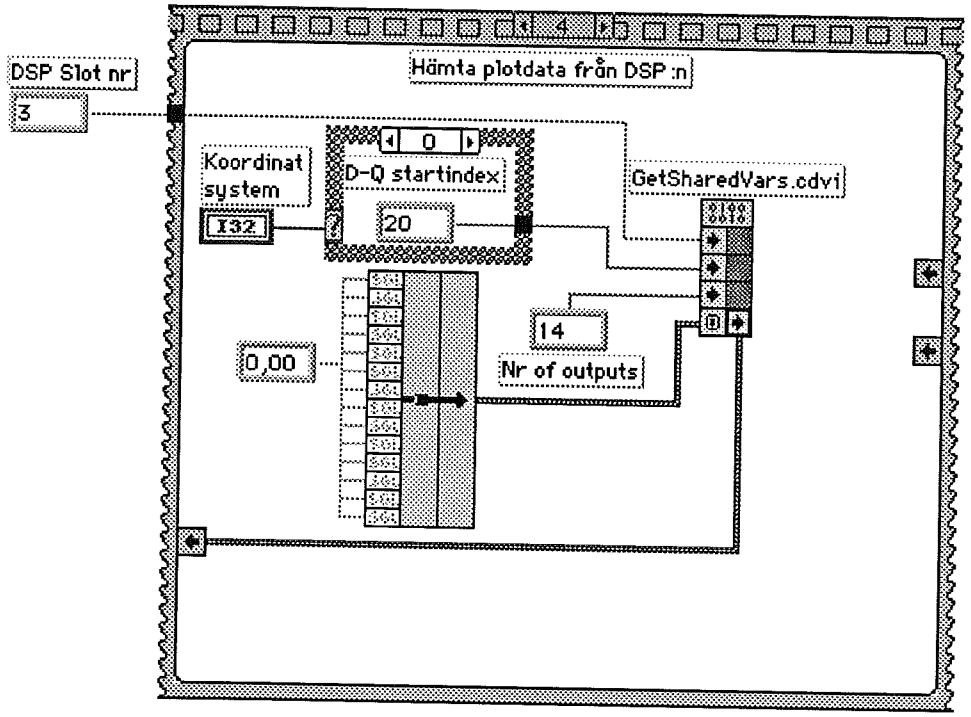
I figur A.8 ses hur grafen av statorspänningsvektorn byggs upp. Spänningsvektorn är de två sista delarna av det komplex som hämtats från signalprocessorn. Dessa två delar får sedan utgöra det andra elementet i två vektorer, vars första element är noll. På detta vis får man grafvisaren att dra en linje från $\{0,0\}$ till $\{U_{sx}, U_{sy}\}$.

Vektorerna sätts samman till en graf. grafen görs om till ett komplex av grafer, som skalas upp 2,5 ggr och matas vidare till grafvisaren.

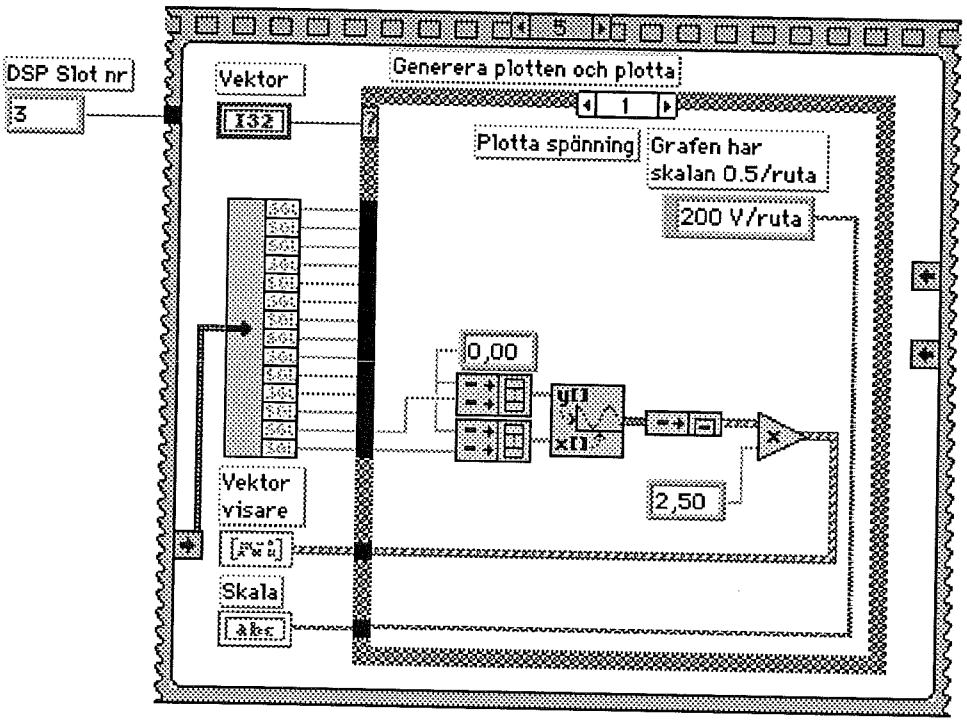
I villkorsstrukturen finns också en textsträng som visas i en indikator ovanför grafen på frontpanelen. Textsträngen talar om vad det är för skala på de vektorer som visas.

Förutom genereringen av statorspänningesvektorn finns det två fall till, som genererar flödesvektorer samt strömvektorer. dessa visas i figur A.9

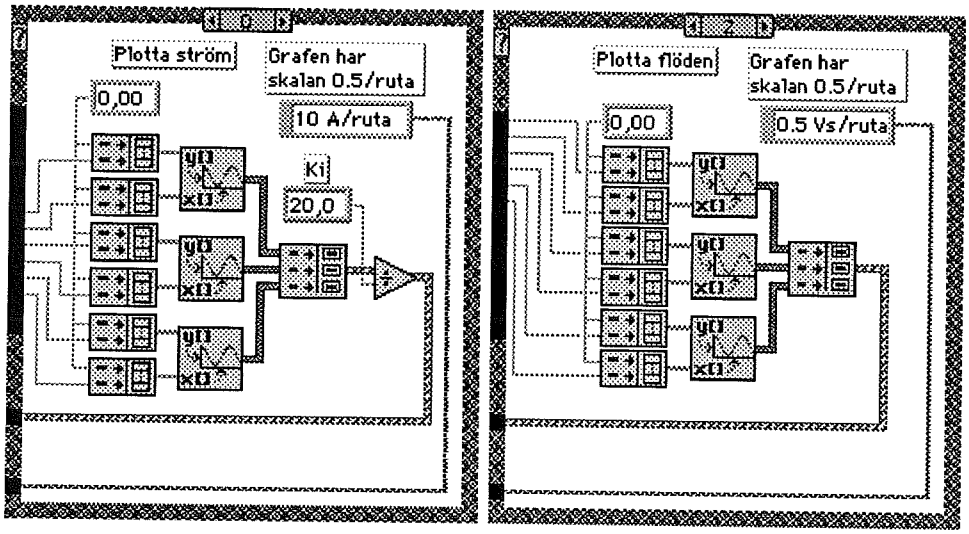
Det sista steget i sekvensen är att läsa av skalära värden från signalprocessorn. I figur A.10 visas denna ruta. Värdena hämtas på samma sätt som i figur A.7



Figur A.7: Steg 5 i AMregulators huvudsekvens. I den här rutan hämtas ärvärden på alla vektorstorheter.



Figur A.8: Steg 6 i AMregulators huvudsekvens. Den här rutan tar de vektorstorheter som hämtades i steg 5 och uppdaterar oscilloskopet.



Figur A.9: Steg 6 har två möjligheter till att generera vektorer som visas på oscilloskopet.

med skillnaden att komplexet är mindre samt hämtas från andra index. De inlästa värdena används sedan för att uppdatera instrumenten på frontpanelen.

A.2.2 Inställning av regulatorer

Instrumentet "Regulatorparametar" används för att ställa in regulatorparametrarna för de tre regulatorer som finns i reglersystemet. Alla tre regulatorerna är PI-regulatorer.

Panel

Panelen har en skjutskala för förstärkning och en skjutskala för integrationstid för var och en av de tre regulatorerna, se figur A.11.

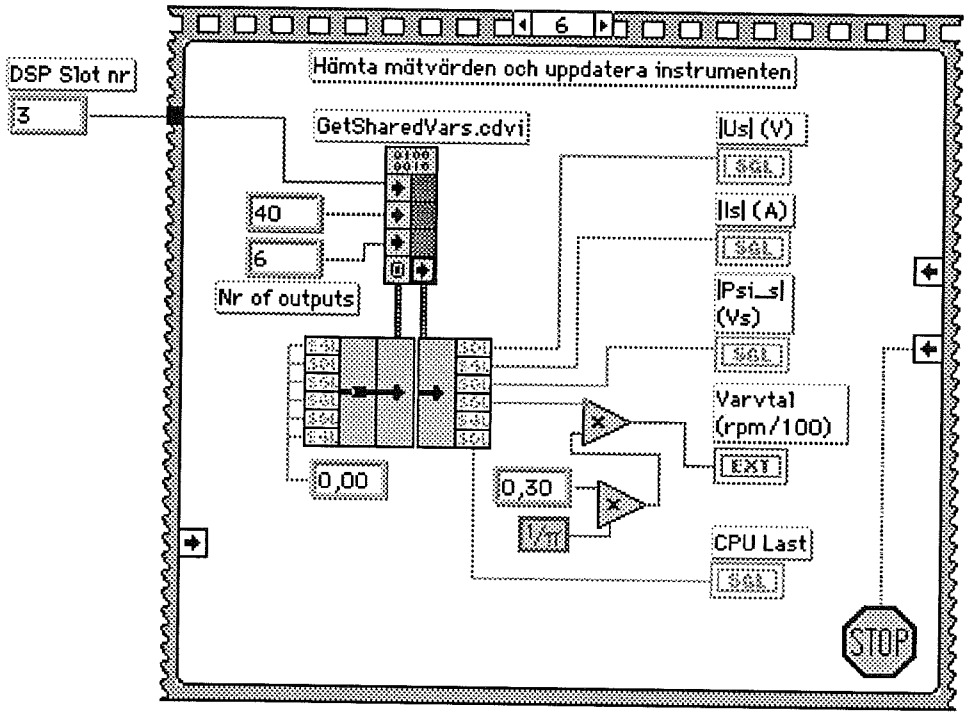
Förutom skjutskalorna finns det två knappar som kontrollerar uppdateringen. Den ena (Initiera) är knuten till START-knappen, vilket medför att regulatorerna får sina parametrar automatiskt vid uppstart. Den andra används för att uppdatera parametrar efter ändringar.

Diagram

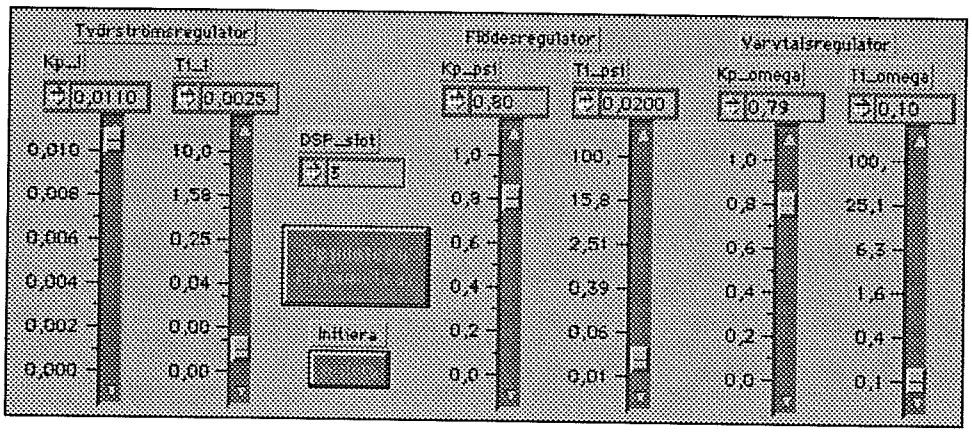
I figur A.12 visas diagrammet till instrumentet "Regulatorparametar". Instrumentet läser av inställningen av skjutskalorna och, om det är första gången som instrumentet anropas eller om någon av knapparna "Uppdatera parametrar" eller "initiera" är intryckta, transporterar ned parametrarna till signalprocessorn, där de tas om hand av huvudprogrammet.

A.2.3 Inställning av skalfaktorer

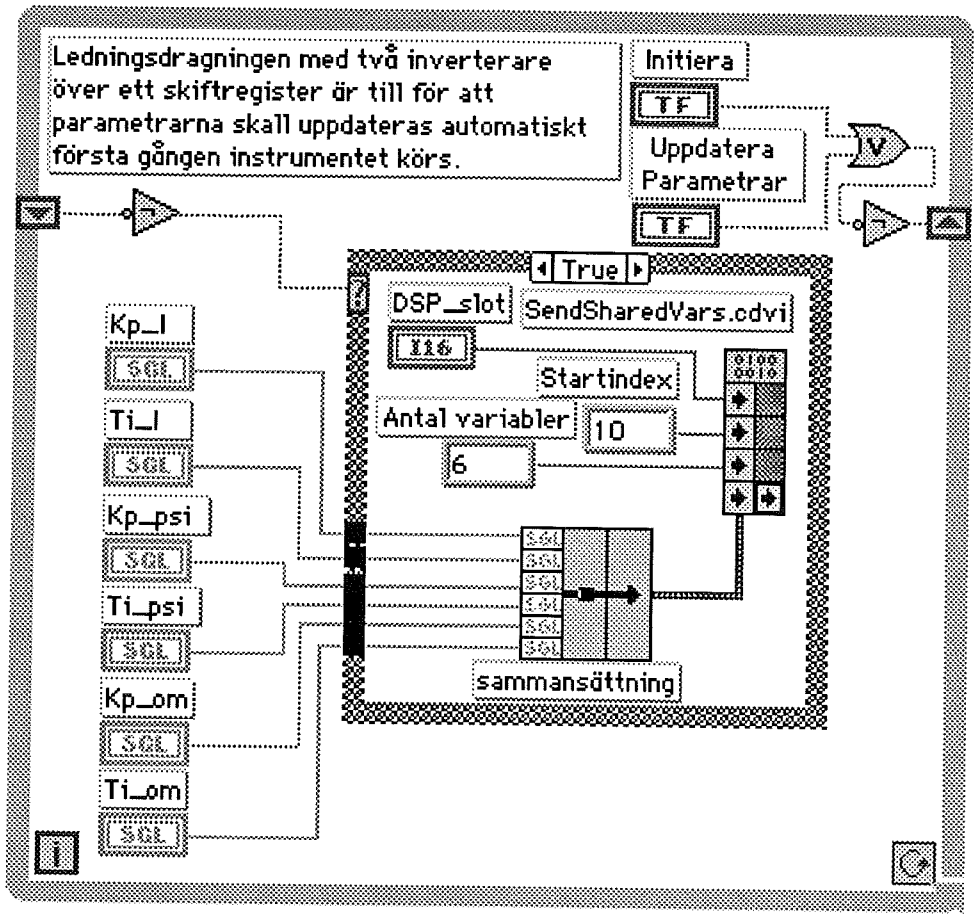
Instrumentet "Givarkonstanter" används för inställning av skalfaktorer i kontakten med den fysiska verkligheten.



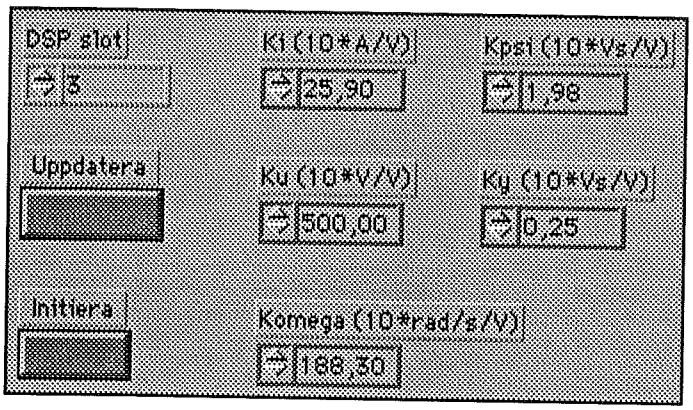
Figur A.10: Steg 7 i AMregulators huvudsekvens. I den här rutan hämtas ärvärden på alla skalära storheter.



Figur A.11: Frontpanelen till instrumentet "Regulatorparametar"



Figur A.12: Diagrammet till instrumentet "Regulatorparametar"



Figur A.13: Frontpanelen till instrumentet "Givarkonstanter".

Panel

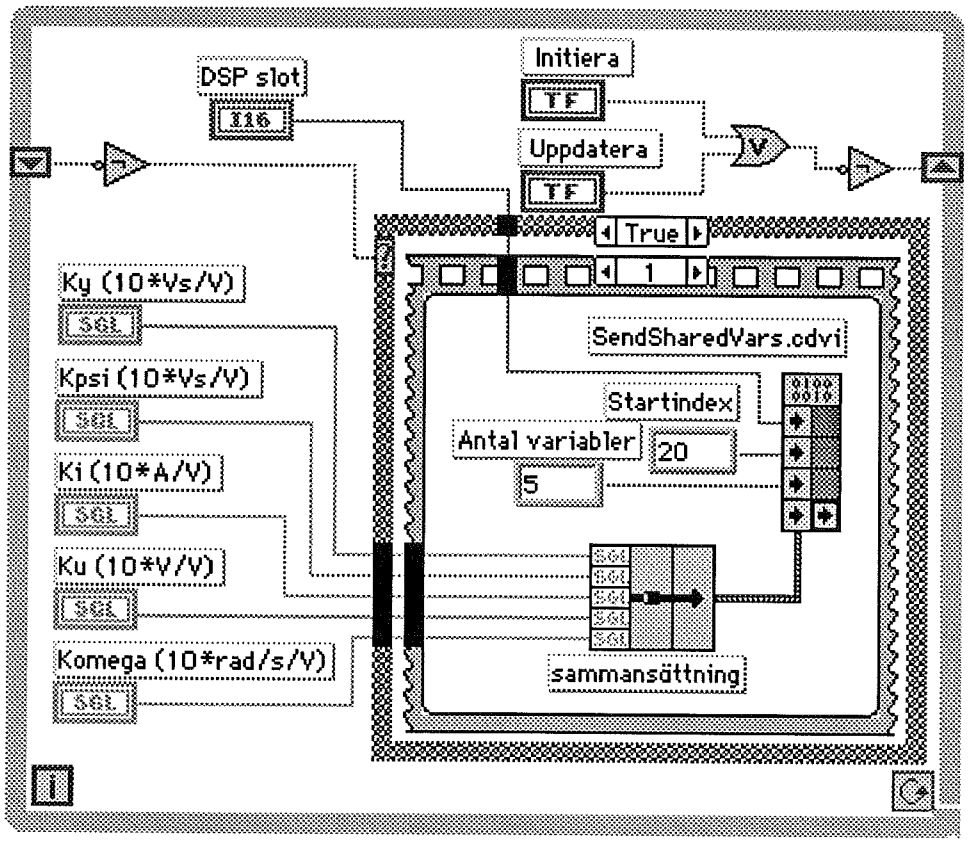
Panelen ,se figur A.13, har siffervisare för att ställa in samtliga skalfaktorer, samt två knappar med samma funktion som hos instrumentet "Regpar".

Diagram

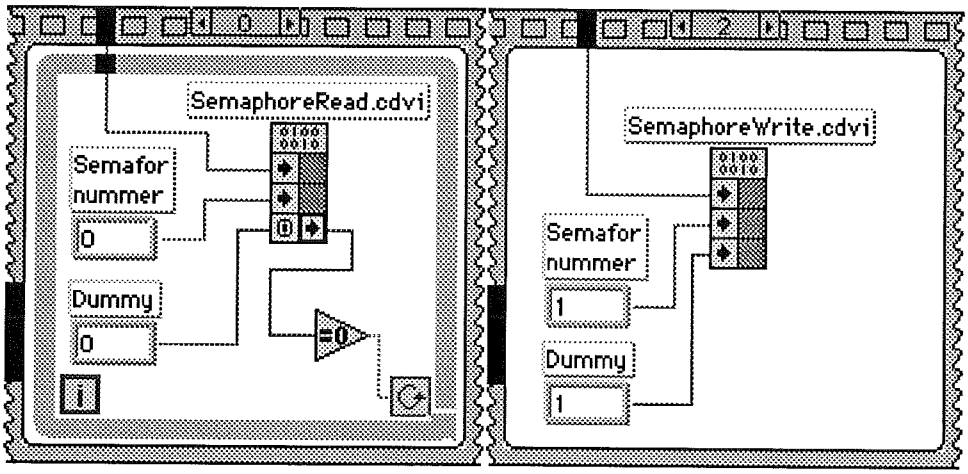
I figur A.14 visas diagrammet till instrumentet "Givarkonstanter". Instrumentet läser av inställningen av sifferindikatorerna på panelen och, om det är första gången som instrumentet anropas eller om någon av knapparna "Uppdatera parametrar" eller "initiera" är intryckta, transporterar ned parametrarna till signalprocessorn, där de tas om hand av huvudprogrammet.

Förutom att transportera ned parametrarna används detta instrument till att känna av om signalprocessorn är startklar eller ej. detta visas i figur A.15. I figuren finns de två återstående rutorna i sekvensen som ses i figur A.14.

I den första rutan känner LabVIEW av värdet av den delade variabeln med index noll. Så länge denna variabel är lika med noll fördröjs LabVIEW i denna ruta. När signalprocessorn har ettställt denna variabel, fortsätter instrumentet med att uppdatera skalfaktorerna, och därefter signallera till signalprocessorn att den kan starta regulatorerna.



Figur A.14: Diagrammet till instrumentet "Givarkonstanter".



Figur A.15: Hanteringen av semaforer i instrumentet "Givarkonstanter".