CODEN: LUTFD2/(TFRT-5434)/1-61/(1991)

Digital reglering med signal-processorer och C++

Ulf Mattsson

Institutionen for Reglerteknik
Lunds Tekniska Högskola
Februari 1991

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

Document name
MASTER THESIS

Date of issue
Febuary 1991

Document Number
CODEN: LUTFD2/(TFRT-5434)/1-061/(1991)

Author(s)
Ulf Mattsson

Supervisor
Klas Nilsson

Sponsoring organisation

Title and subtitle
Digital reglering med signalprocessorer och C++
(Digital Control using Digital Signal Processors and C++)

Abstract

For research in robot programming and sensor-based motion control, an experimental platform containing a VME computer system is used. The system contains computer boards with M680x0 processors for general computing and control, and a board from AT&T with six DSP32C signal processors for more demanding control algorithms. The general programming language to be used in the system is C++. The purpose of this work is to explore the feasibility of using C++ for efficient programming of the DSPs.

A summary is given of the DSP32C and its pipelining, interrupts, and C-language interface to the assembler. The AT&T C++ compiler *Cfront* is used to generate ANSI-C code for the DSP32C C compiler. An interface written in assembly language makes it possible to write interrupt routines in C or C++. The use of inheritance when writing control algorithm is investigated, and a base class Regulator and a derived class PID have been developed as an example.

Critical parts of control algorithms or signal processing algorithms sometimes have to be implemented in assembly language to fully utilize the computing power of the DSP. The C++ compiler has therefore been interfaced to allow easy and structured use of inline assembly code, which is supported by the DSP32C C compiler. This feature is also useful when implementing new real-time primitives. As an example of this, an interrupt handling class has been developed.

A small servo control example has been implemented on the VME DSP32C board. By execution in the target system, simulator tests of the assembly code inlining and classes therefore was verified. The conclusion is that programming of algorithms in C++, test in simulator, and downloading programs to the target system is a convenient way of implementing control systems.

Key words
Real-time programming, Process control, Signal Processing, C++, DSP32C

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
Swedish

Number of pages
61

Recipient's notes

Security classification

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

Innehållsförteckning

Förord 2

1. Inledning 3

2. Signalprocessorn DSP32C 5
2.1 Arkitektur 5
2.2 Utvecklingshjälpmedel 7
2.3 Programmering 8

3. C++ för DSP32C 10
3.1 Anpassningen av kompilatorn 10
3.2 Snittet mot assemblern 10
3.3 Inkapsling av asm-makro i en klass 12
3.4 Några hårdvarunära klasser 13

4. Realtidsprimitiver 14
4.1 Förgrund/Bakgrund Scheduler 14
4.2 Avbrotten hos DSP32C 14
4.3 Avbrottshanterare i C++ 15
4.4 Exempel på användnig av avbrott 18

5. Objekt Orienterad Programmering av Regleralgorimer . . 19
5.1 Inledning 19
5.2 Klasstruktur 19
5.3 PID-reglering 20

6. Systemintegration och Igångkörning 22
6.1 AT&T's SURFboard 22
6.2 Simulering 24
6.3 Exekvering i målsystem 24

7. Referenser 25

Appendix 26
A. Inkapsling av klassen io 26
B. Inkapsling av klassen dsp2ieee 31
C. Inkapsling av klassen intr 37
D. Startfilen vmeinit.s assemblerkod 45
E. PID-regulatorns C++ kod 53
F. Simuleringsprogrammets C++ kod 55
G. Målsystemets testprogram 58

Förord

Denna rapport avser ett examensarbete inom Institutionen för Reglerteknik vid Lunds Tekniska Högskola. Arbetet har till stor del inneburit test- och igångkörning av sådan programvara och maskinvara som inte ingår i civilingenjörsutbildningen, D. Detta har varit tidskrävande, men jag har genom utformningen av denna rapport försökt underlätta efterföljandes arbete, speciellt i inledningsskedet. Ett flertal avsnitt är utdrag av viktiga delar i de manualer som finns. Dessa avsnitt har översatts och i viss mån skrivits om för att ge en allmän introduktion och underlätta vidare läsning i manualerna.

Jag vill här passa på att framföra ett tack till Klas Nilsson för hans aktiva medverkan i arbetet, och till Dag Brück som kom med goda idéer för anpassningen av C++ kompilatorn.

1. Inledning

Digitala SignalProcessorer (DSP) är optimerade för signalbehandling av t ex sensorsignaler i ett robotsystem. Regleralgoritmer har stora likheter med filteralgoritmer och speciellt för snabba processer som industrirobotar, ger signalprocessorer möjlighet till implementering av nya algoritmer. Problemet är dock att en signalprocessors speciella arkitektur ställer speciella krav på programmeringen, som därför hittills mest utförts i assembler och i viss mån i C, och utan kraftfulla realtidsprimitiver. Inom detta examensarbete har arbete med följande inriktning utförts:

- Anpassning av AT&T's C++ kompilator för Sun, till AT&T's signalprocessor DSP32C och speciella C-kompilator.
- Objektorienterad design av regulatormoduler.
- Studium och utveckling av realtidsprimitiver lämpade för signalprocessorer.
- Igångkörning av hårdvara och utvecklingsprogramvara.
- Sammankoppling med konventionella mikroprocessorer i ett VME kortdatorsystem, med Sun arbetsstationer som värdmiljö.

AT&T's DSP32C Digital Signal Processor

En DSP är främst optimerad för beräkningar av typen: $a = b + c * d$, vanligt förekommande i t ex regleralgoritmer. DSP32C räknar med 32-bitars flyttal. Accumulering av produkter sker dock internt med 40-bitars noggrannhet.

För att erhålla en hög beräkningskapacitet används en 4-stegs "pipeline", där två olika exekveringsenheter, CAU (control arithmetic unit) och DAU (data arithmetic unit) samarbetar. Flyttalsberäkningarna sker i DAU, där en multiplicerare och adderare arbetar parallellt, medan CAU sköter kontrollfunktionerna, administrerar förflyttningen av data samt utför heltalsberäkningarna och logiska operationer.

Den C-kompilator som finns till DSP32C uppfyller ANSI-C, men har dessutom utökats med ett speciellt snitt mot assemblern. Normalt kan dock programmeringen ske helt i C.

C++ för DSP32C

C++ tillhör de objektorienterade språken, vilka karakteriseras av följande tre huvudegenskaper:

- Inkapsling: Med en typ som kallas klass, vilken innefattar både data (attribut) och operationer (metoder) på dessa data, kan man skapa ett eller flera objekt av klassen.
- Ärvning: Utgående från en grundläggande basklass, kan man successivt skapa mer specifika underklasser. De nya underklasserna ärver basklassens data och operationer.
- Polyformism: Detta gör det möjligt att att knyta olika metoder med samma namn till de skilda underklasserna på ett sådant sätt att att rätt metod automatiskt tillämpas på varje enskilt objekt.

Dessa egenskaper gynnar en modulär programuppbyggnad och underlättar konstruktion av återanvändbara och flexibla programkomponenter. De objektorienterade tankegångarna försöker efterlikna verklighetens mångfald av objekt med skiftande attribut och egenskaper.

Det objektorienterade synsättet passar väl in på regleralgoritmer där blocken i ett blockschema naturligt representeras av olika objekt. Inom institutionen för reglerteknik finns kompetens inom C++, och arbete med en ny realtidsmiljö för konventionella mikroprocessorer pågår. Motiv för objektorienterad programmering tillsammans med önskemål om likartad programmering av mikroprocessorer och signalprocessorer utgör motiv för att anpassa C++ för DSP32C.

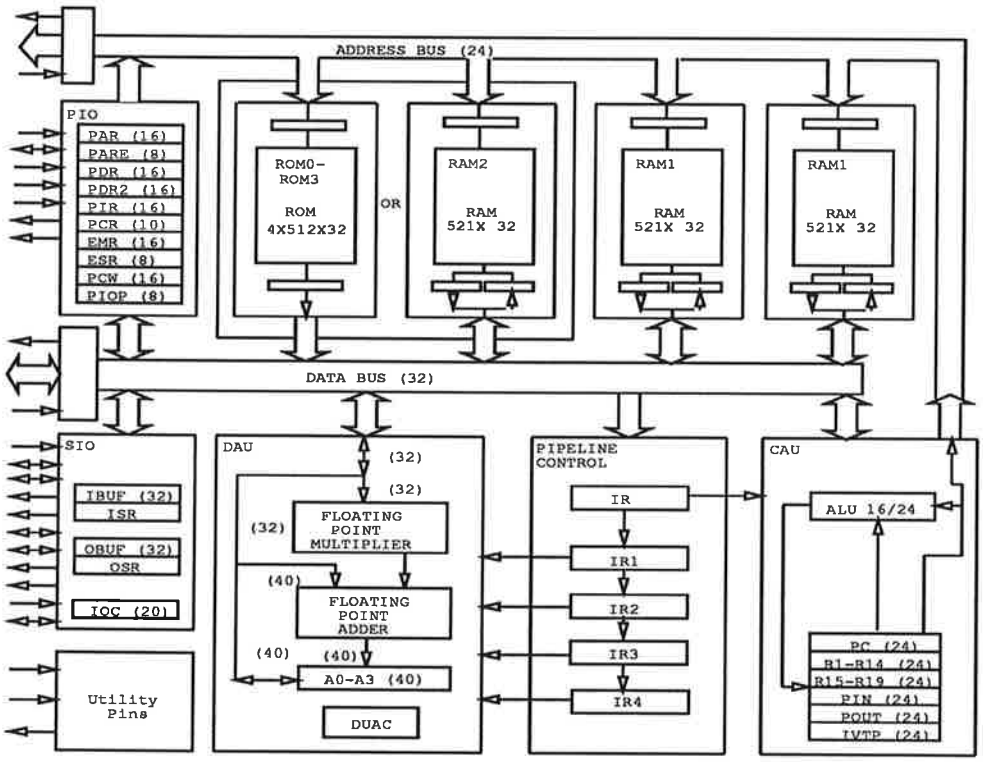
Denna rapport har följande uppläggning: I kapitel 2 beskrivs först DSP32C och dess utvecklingshjälpmedel. Kapitel 3 beskriver snittet mot assemblern, samt hur detta bevaras i den anpassning av C++ kompilatorn som utförts. En beskrivning av hur man kapslar egna assemblerfunktioner i C++ klasser ges också. Kapitel 4 beskriver avbrotten hos DSP32C, samt hur programmering av avbrottshanterare i C++ möjliggjorts. Kapitel 5 tar upp objektorienterad design av regulatormoduler. Kapitel 6 slutligen beskriver ett VME-kort med sex stycken DSP32C, samt hur detta integrerats i ett realtidssystem för robotstyrning.

Verklig exekvering i målsystemet visar dels att de utvecklade programmen fungerar bra, samt att programmering i C++ av regleralgoritmer, test i simulatormiljö, och nerladdning av färdigt program är ett lovande arbetssätt.

2. Signalprocessorn DSP32C

DSP32C är en signalprocessor som optimerats för att klara repetetiva flyttalsopereationer av typen $a = b + c * d$. I detta kapitel beskrivs hårdvarans arkitektur, utvecklingshjälpmedel, samt programmering av processorn.

2.1 Arkitektur



Figur 2.1 Blockschema för DSP32C

Två block, CAU (control arithmetic unit) och DAU (data arithmetic unit) utför exekveringen. Med hjälp av en PIPELINE CONTROL som administrerar en fystegs pipeline kan CAU och DAU arbeta parallellt, vilket medför att beräkningskapaciteten ökar. De båda exekveringsenheterna har var för sig en egen uppsättning av instruktioner. CAU utför 16- eller 24-bitars heltalsaritmetik och logiska operationer samt administrerar dataförflyttningen mellan register och minnesareorna. I DAU sker alla flyttalsberäkningar och dessutom även konvertering mellan olika datatyper.

Intern datatransport sker via en 32-bitars databuss. Genom den kan processorns pipeline-kontroll under en instruktionscykel, som är indelad i fyra maskintillstånd, göra fyra minnesaccesser: hämtning av instruktion, hämta två operander från minnet eller en I/O-port samt skriva till minnet eller en I/O-port.

DSP32C är försedd med ett internt minne, ett gränssnitt till extern minnesexpantion och minnesmappade yttre enheter. Med en adressbuss på 24 bitar

kan man adressera upp till 16 Mbyte. Hela minnesarean kan utnyttjas godtyckligt, gränssnittet mot det externa minnet hanterar bussen och eventuella "wait-states".

Den parallella I/O (PIO)-porten ger ett parallellt gränssnitt för kommunikation mellan DSP32C och externa enheter. Överföringen kan ske med 8 eller 16 bitar. Seriell kommunikation och synkronisering med enheter utanför DSP32C kan ske med den seriella I/O (SIO)-porten. Genom tre DMA-enheter kan man få direkt access till minnet via de seriella I/O-portarna och den parallella I/O-porten. En en-nivås interruptenhet finns också inbyggd på chipet.

DAU (data arithmetic unit)

Den primära exekveringsenheten för signalprocessaloritmer är DAU, som tillsammans med PIPELINE CONTROL utför flyttalsberäkningarna. Huvuddelarna i DAU är en 32 bitars flyttalsmultiplicerare, en 40 bitars flyttalsadderare, fyra 40 bitars accumulatorer och ett kontrollregister (DUAC). DAU's arbetssätt förklaras enklast med ett exempel. Om instruktionen

```
Z = aN = aM + Y * X
```

skall exekveras, sker det i fyra steg, hämta X och Y, multiplicera X och Y, addera resultatet av multiplikationen med innehållet i register aM och lägg resultatet i register aN, samt skriv ut resultatet i minnet eller till en I/O-port. Alltså:

steg 1:	hämta X och Y
steg 2:	multiplicera X och Y
steg 3:	accumulera produkten med aM
steg 4:	gör en skrivning till minne eller I/O-port

Om nu många instruktioner av detta slag exekveras i en följd, t ex vid matrisberäkningar, kommer DSP32C att automatiskt genom pipeline lämna ett resultat per instruktionscykel. Följande exempel visar hur DSP'n utför detta.

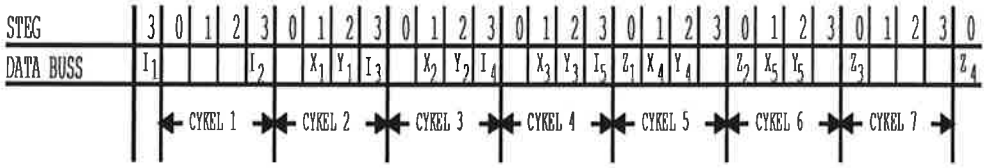
Antag att följande instruktioner skall utföras:

1) $Z = aN = aM + X_1 * Y_1$
2) $Z = aN = aM + X_2 * Y_2$
3) $Z = aN = aM + X_3 * Y_3$
4) $Z = aN = aM + X_4 * Y_4$
5) $Z = aN = aM + X_5 * Y_5$

då kommer DSP'n att exekvera följande sekvens av instruktioner:

1)				hämta_1 XY
2)			multiplicera_1	hämta_2 XY
3)		accumulera_1	multiplicera_2	hämta_3 XY
4)	skriv_1	accumulera_2	multiplicera_3	hämta_4 XY
5)	skriv_2	accumulera_3	multiplicera_4	hämta_5 XY
6)	skriv_3	accumulera_4	multiplicera_5	
7)	skriv_4	accumulera_5		
8)	skriv_5			

I följande figur visas hur pipeliningen utnyttjar data-bussen under exekveringen av föregående exempel. Med I_x menas att instruktionen hämtas, X_x resp Y_x visar när data hämtas från minnet medan Z_x visar när data sparas till minnet.



Figur 2.2 Pipeline-diagram för DSP32C's databuss

CAU (control arithmetic unit)

CAU är ansvarig för adressberäkningar, kontroll av flaggor, 16- eller 24-bitars heltalsberäkning och logiska operationer (CA-instruktioner). Den består av tre huvuddelar, en ALU, en 24-bitars programräknare samt 22 stycken 24-bitars register. CAU utför två olika saker, för det första görs CA-instuktioner, men dessutom genererar CAU adresser till DAU's operander (DA-instruktioner). CA- och DA-instruktioner avänder registerna i CAU på olika sätt. För DA-instruktioner används r1-r14 (rP) som pekare till minnet, r15-r19 som inkrementregister (rI) till rP, r20 (PIN) och r21 (POUT) som in- resp ut-pekare när DMA används vid dataöverföring över seriell I/O, och till sist r22 (IVTP) som skall innehålla adressen avbrottsvektortabellen. Allt detta måste man ta hänsyn till när man använder dessa register till CA-instruktioner, där man kan använda alla registerna generellt. CAU kan utföra instruktioner medan DAU håller på med pipline-operationer. Detta gör att DAU och CAU kan jobba parallellt.

2.2 Utvecklingshjälpmedel

rtpi

Rtpi är en debugger som arbetar med både C++ koden och assemblerkoden. För att få in källkoden i debuggern måste programmen kompileras med valet (option) -g, samt filnamnet skall avslutas med `.c`. Då den inte fanns tillgänglig i början av examensarbetet, har jag inte använt mycket den vid simuleringen av programmen.

dcsim (d3sim)

dcsim är en simulator som simulerar DSP32C som en virtuell enhet. Med hjälp av ett kraftfullt kommandospråk med vars hjälp man kan bl a:

- Ladda in exekverbara program och lista dess assemblerkod.
- Sätta brytpunkter i sitt program.
- Inspektera och ändra register, accumulatorer, minne samt information om processorns interna tillstånd.
- Definiera egna variabler som kan användas till räknare, temporär datalagring, etc.
- Definiera egna kommadosekvenser.

- Skapa egna filer med kommandosekvenser.
- Simulera I/O med de seriella portarna.
- Simulera I/O via de parallella portarna. Alternativt kan dessa använda förformatterade hjälputskrifter med hjälp av standardfunktionen `printf()`.

Avlusning av programmen sker således simulerat i värddatormiljön. Det program som laddas ner i målsystemet förutsättes sedan vara felfritt. För kombinerad felsökning av program och hårdvara finns ytterligare hjälpmedel att köpa, men detta skall ej behövas för det VME-kort från AT&T som beskrivs i kapitel 6.

2.3 Programmering

Programmering bör ske i högnivåspråk, men vissa funktioner, som t ex kommunikation med I/O-enheter, avbrottsrutiner eller optimerade algoritmer måste skrivas i assembler. För att på effektivaste sätt kunna anropa assemblerfunktioner, kan dessa ges typen `asm` vid deklarationen. Mera om detta i kapitel 3.2.

Startfil

Det finns inget operativsystem under vilket programmet kan exekvera. Istället initieras den nödvändiga omgivningen till C-programmet av en startfil. Denna läggs automatiskt in av DSP32C's C-kompilator *d3cc*. Filens namn är `../dsp/lib/crt0_32c.s` och innehåller förutom kod som behövs för att signalprocessorn skall starta exekveringingen på rätt sätt, även kod för debuggern *rtpi*. Startfilen kan uteslutas med valet *-s0* utom då valet *-g* används vi kompileringen. Då kommer av någon anledning alltid `../dsp/lib/crt0_32c.s` att läggas in som startfil. Detta har lösts genom att före kompileringen skapa en *länk* från `../dsp/lib/crt0_32c.s` till den startfil som skall användas.

Minnesmappning

Med filen `../dsp/lib/mem32c.map` delas minnet in i olika sektioner. Denna fil använder sedan länkaren till att styra ut den genererade assemblerkoden till olika platser i minnet. Kompilatorn placerar automatiskt programkod till sektionen *.text*, globala och statiska varibler kommer att hamna under sektionen *.data*. Dessutom finns det en tredje sektion som används till en stack, nämnligen *.bss*.

Listning 2.1 på nästa sida visar hur en mapfil kan se ut. Med `MEMORY` visas hur minnet är indelat. `.mtext` står för minnesbank .text, `o=0x0` talar om var minnesbanken börjar. `l=0x2fff` är dess längd. I `SECTION` styrs de olika segmenten till rätt minnesbank.

Behöver man utnyttja fler än tre sektioner, måste programmet delas upp. Detta inträffar då en del data skall placeras på fixa adresser i ett minne som är åtkomligt från andra processorer. Listningen 2.1 visar hur den separatkompilerade filen `dramdata.o`, med globala och statiska variaber, har placerats med början på adress `0x800000`. Med valet *-m egen_mapfil* vid kompileringen, kommer länkaren att välja *egen_mapfil* i stället för `../dsp/lib/mem32c.map`.

```
MEMORY {
        .mtext:         o=0x0,          l=0x2ffff
        .mdata:         o=0x30000,      l=0xffff
        .mbss:          o=0xfff000,     l=0x1000
        .mdram          o=0x800000,     l=0x100000
}

SECTIONS {
        .dram           {dramdata.o(.data)} > .mdram
        .text:          {}                  > .mtext
        .data:          {}                  > .mdata
        .bss:           {}                  > .mbss
}
```

Lista 2.1 Exempel på en minnesmappningsfil

Assemblerprogrammering

Vid assemblerprogrammering måste man vara uppmärksam på en del saker som kompilatorn normalt administrerar. Beakta speciellt:

- Effekter av pipeliningen, t ex att en DA-instruktion måste följas av 3 stycken CA instruktioner innan instruktionen `ireturn` får exekveras.
- Tilldelning av typ `r1 = 10` kommer endast att påverka de 16 minst signifikanta bitarna i registret, de andra behåller sina värden. `r1e = 10` däremot påverkar alla bitarna i registret.
- Kontrollregister kan ej tilldelas ett värde direkt. Istället skall värdet först läggas i ett register, sedan får kontrollregistret registrets värde.
- En hoppsats, som t ex `goto` eller `call`, medför att även den närmast följande instruktionen exekveras innan ett hopp sker.

C-programmering

Även vid programmering i C (och C++) bör vissa regler beaktas, inte för att resultatet påverkas, utan för att koden skall bli effektiv. Beakta speciellt:

- Deklarera ofta använda variabler som registervariabler.
- Använd postmodifiering av pekare, dvs `ptr++` och `ptr--`, och inte `++ptr` och `--ptr`.
- Använd inte dubbel precision.
- Vid upprepade operationer av typen $a = b + c * d$ blir koden effektivast om ingående variabler deklarerats antingen som `register float` eller som `register float*` (`*` betyder pekare), och om dessa båda typer används omväxlande.
- Referens via pekare är effektivare än array-indexering, speciellt om elementen används i tur och ordning.
- Lägg logiska variabler i egna 32-bitars ord, dvs använd `int` och ej bit-fält.

Ytterligare tips ges i kapitlet *Programming Hints* i C-manualen.

3. C++ för DSP32C

Detta kapitel handlar om hur C++ kompilatorn för Sun har anpassats till signalprocessorns C-kompilator, hur C-kompilatorns snitt mot assemblern är uppbyggt, samt hur detta tillvaratas i C++ anpassningen.

3.1 Anpassningen av kompilatorn

AT&T's C++ kompilator, *Cfront* genererar ANSI-C kod, som används av DSP32C's C-kompilator *d3cc*. Med det egna kommandot *dspCC* anropas de kommandon som skall utföras vid kompilering av program skrivna i C++. Hur dspCC är implementerat och exakt hur den av *Cfront* genererade koden modifieras för att passa assemblersnittet i *d3cc* tillhör inte mina bidrag och faller därför utanför denna rapport.

3.2 Snittet mot assemblern

För att kunna lägga in assemblerinstruktioner i sitt C-program finns två möjligheter. Ett sätt är att lägga in en följd av *asm*-instruktioner:

```
asm(" asm-instruktion 1 ")
asm(" asm-instruktion 2 ")
asm(" asm-instruktion 3 ")
```

Med denna teknik kan man dock inte referera till egna variabler och koden kan inte optimeras med avseende på var komilatorn placerat egna eller temporära variabler. Därför är DSP32C's C-kompilator *d3cc* utökad med ytterligare ett snitt mot assemblern. Med ett sk *asm*-makro kan en C-funktion av typen *asm* deklareras. I följande beskrivning av ett *asm*-makro betyder ett `[xx]*` att xx skall tas med ingen eller flera gånger, `[xx]+` att xx tas med minst en gång, och `[xx]` betyder att xx kan uteslutas.

Syntax:

```
asm [typspecifiering] identifierare ([parameterlista])
{
[ parameter-allokerings-linje
        assembler-kropp ]*
}
```

parameter-allokerings-linjen har följande syntax:

```
% [ allokering [ identifierare [,identifierare ]* ] ; ]+
```

Den består av *en och endast en* rad, som börjar med ett *%* som följs av namnen på identifierarna samt var de formella parametrarna skall allokeras. Har man flera olika allokeringsvarianter av de formella pararametrarna, måste man specificera flera olika allokeringslinjer i samma makro. Om makrot inte har några parametrar behöver linjen inte finnas med.

De allokeringssätt som kompilatorn känner till är:

reg En variabel som kompilatorn allokerat i ett maskinregister.

con En konstant, dvs värdet är känt vid kompileringstillfället.

mem En variabel som matchar alla adresseringssätt, inklusive *reg* och *con*. Genom att specificera *reg* och *con* före *mem* vet man att variablen finns i minnet.

lab En kompilatorgenererad variabel dvs en adress, till vilken man kan referera lokalt inom *asm*-funktionen, men som vid makroexpansionen ges ett inom hela programmet unikt namn.

error Genererar ett kompileringsfel om kompilatorn inte hittar en linje med passande allokering innan *error* nås.

assembler-kropp är den assemblerkod som kompilatorn kommer att lägga in i programmet.

Eventuella returvärden från ett makro skall placeras i olika register berode på vilken typ de har. Är typen *int* kommer returvärdet att finnas i register *r1*, *long int* utnyttjar både *r1* och *r2* med den lägre halvan i *r1* och flyttal returneras i register *a0*.

Exempel

```
asm void dsp2ieee_cp(dest, source, length)
{
%       lab l1; reg dest, source; mem length;
        r3e = length
        r3e = *r3
l1:
        if (length-- >= 0) goto l1
        *d++ = a0 = ieee(*s++)
        @W
}
```

I detta enkla lilla exempel konverteras flyttal från DSP32C's interna format till IEEE-format samtidigt som de kopieras från en plats i minnet till en annan. Exemplet kräver att adresserna `dest` och `source` ligger i register, vilket normalt kan åstadkommas med `register`-deklaration i C++. Notera även att den *mem*-deklarerade variablen först måste hämtas in från minnet innan den används. `@W` medför att ett tillräckligt antal *nop*-instruktioner llläggs in för att DSP32C' eventuella pipelinade instruktioner skall exekveras färdigt innan nästa C++ instruktion påbörjas. I den fullständiga implementeringen finns 12 olika allokeringslinjer för att alla kombinationer av de tre argumenten skall fångas.

3.3 Inkapsling av asm-makro i en klass

Anrop av en *asm*-funktion resulterar typiskt i ett fåtal maskininstruktioner. Vid programmering i C lägger kompilatorn dessa instruktioner *inline*, dvs utan subrutinanrop och annan onödig påverkan av pipeliningen. I C++ vill man ha kvar denna möjlighet till effektiv kodning, men problemet är att *Cfront* inte accepterar C-funktioner av typen *asm*, vilka därför behöver kompileras separat med C-kompilatorn. I C++ kan en procedur deklareras som *inline*, men den kan då inte vara deklarerad som `extern "C"`. Detta innebär att *asm*-funktioner inte kan läggas inline, vilket är vårt krav, utan speciella trick.

Vår lösning är att vid C++ kompileringen inkludera filer (vars namn börjar med `~d3cc`) där *asm*-funktionerna deklarerats som `extern "C"`, och sedan modifiera det genererade C-programmet så att funktioner av typen *asm* används vi den efterföljande C-kompileringen.

För att få en snygg struktur har vi valt att kapsla *asm*-funktionerna från en fil i en klass, vars samtliga attribut deklarerats som `static`. Detta krävs ej för att `asm`-funktioner skall kunna användas, utan ger en kapsling jämförbar med Modula-2's modulbegrepp. En beskrivning av hur man kapslar sina assemblerfunktioner i C++ följer nu:

1. **Skapa en fil som innehåller klassdeklartionen enligt följande:**

```
asm("#include \"../filnamn.asm\"");
#include "../~d3cc_filnamn.H"

 class klassnamn {
   public:
   static [typ] funktionsnamn( [typdekl parmeterlista] ) {
     ::funktionsnamn( [parameterlista] );
   }
   static [typ] funktionsnamn( [typdekl parmeterlista] ) {
     return::funktionsnamn( [parameterlista] );
   }
 };
```

Normalt är `klassnamn` lika med `filnamn`. Som exempel finns klassen `io` i appendix A. För att *Cfront* skall ignorera filen med asm-funktionerna, skrives då includesaten

```
#include "../klassnamn.asm"
```

som

```
asm("#include\"../klassnamn.asm\"");
```

Efter det att *Cfront* exekverats behandlas den genererade C koden nämnligen med ett *awk*-skript, som plockar bort `asm("` och `");`, vilket gör att assemblerkoden sedan inkluderas (inline) vid C-kompileringen.

2. Skapa en *header*-fil

I C++ måste C-funktioner deklareras som `extern "C"` enligt *externdeklaration:*

```
extern "C" typ funktionsnamn( [typ parameter]* );
```

viket görs i en *header*-fil:

```
~d3cc_klassnamn.H
```

där ~ är tänkt att anknyta till **destruktors** i C++. För att inte kompilatorn skall ge felmeddelande, plockas även rader där `~d3cc_` ingår bort när sedan *awk*-skriptet exekveras. Även kod som genererats av satserna i filer vars namn börjar på `~d3cc_` tas bort.

3. Skapa en fil för *asm*-makrona

Till sist skall *asm*-makrona ligga i en fil för sig, vilken sedan kommer att hämtas in av DSP32C's C-kompilator.

Exempel på hur inkaplingen används

Filen med klassdeklarationen måste för det första inkluderas. Anrop sker genom att använda operatorn ::, med klassnamnet på ena sidan och funktionsnamnet på den andra.

I följande exempel sätts signalprocessorns ioc-register (I/O-kontrollregister).

```
#include "io.c"

#define IOC_DEFAULT     0x0CC5

main {
      .
      .
  io::set_ioc(IOC_DEFAULT);
      .
      .
}
```

3.4 Några hårdvarunära klasser

I programvaran som följer med finns det färdiga filer med *asm*-makron. En av dem, io.asm (Appendix A), har kapslats in en klass, io, som har lagts i filen io.c. Dessutom har jag gjort två nya klasser: dsp2ieee för flyttalkonvertering (Appendix B) och intr för hanteringen av avbrott (Appendix C).

4. Realtidsprimitiver

En realtidskärna som är flexibel och har kraftfulla realtidsprimitiver, skall kunna hantera bl a processer, prioriteter och tidsdelningsalgoritmer, semaforer, händelser och avbrott. Men en sådan realtidkärna får man inte gratis. Vid t ex ett processbyte skall en komlicerad algoritm gås igenom. Detta stjäl en massa processortid som istället kunde ha används till att exekvera en reglerloop oftare. Genom att istället skriva en kärna med endast de primitiver som behövs, kan mycket tid vinnas. Ett exempel på detta är en Förgrund/Bakgrund *Scheduler*. Genom att utnyttja DSP32's avbrottshanteringssystem kan en variant på en f/b scheduler skrivas, med de olika avbrottsrutinerna som förgrundsfunktioner.

4.1 Förgrund/Bakgrund Scheduler

En typisk f/b scheduler har två "processer". Förgrundsprocessen är en procedur som anropas vid fixa tidsintervall av shedulern. Förgrundsproceduren får dessutom alltid exekveras klar. När förgrundsproceduren inte exekveras sägs bakgrundsprocessen, som är ett vanligt sekvensiellt program, exekvera. Vanligtvis är förgrundproceduren en reglerloop, medan bakgundsprocessen har hand om operatörskommunikationen, felövervakning etc.

En fördel med f/b schedulern är att den använder endast *en* stack, till skillnad mot en realtidskärna som använder en separat stack för var process. När förgrundsprocessen skall exekveras sparas först processorns status, först därefter börjar förgrundsproceduren att exekvera, och använder då samma stack som bakgrundsprogrammet. Stack och status återställs till sitt ursprung när förgrundproceduren avslutas.

Eftersom det inte finns någon reell process, behövs inga prioriteter eller tidsdelningsalgoritmer. Förgrundsproceduren avbryter bakgrundsprogrammet och har därför automatiskt högre prioritet.

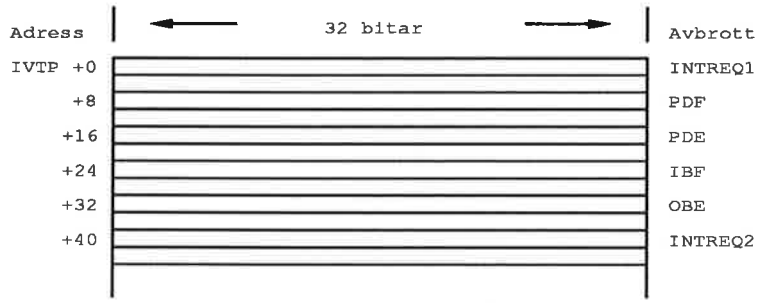
4.2 Avbrotten hos DSP32C

DSP32C har sex olika avbrottskällor, fyra interna och två externa. Avbrotten är "single-leveled", vilket innebär att när en avbrottsrutin har börjat exekvera, kan den inte avbrytas av ett annat avbrott oavsett dess prioretet. Avbrottsrutinen körs alltså alltid klar innan det sker ett hopp tillbaka, vilket kan utnyttjas till att skriva en f/b scheduler. Detta medför också att man måste ha bra programmeringsdisiplin, när man skriver de olika avbrottsrutinerna, för att uppfylla eventuella tidskrav.

De olika avbrottskällorna är, uppräknade i prioritetsordning (INTREQ1 har högst prioritet):

1. External Interrupt One (INTREQ1)
2. Parallel Buffer Full (PDF)
3. Parallel Buffer Empty (PDE)
4. SIO Input Buffer Full (IBF)
5. SIO Output Buffer Empty (OBE)
6. External Interrupt Two (INTREQ2)

För att kunna använda avbrotten måste de först konfigueras med bitarna 10 - 15 i PCW-registret. Dessutom skall r22 (IVTP) innehålla basadressen till en avbrottsvektor.



Figur 4.1 Avbrottsvektor

Vid ett avbrott kommer DSP32C att först exekvera ytterligare en instruktion, och därefter hoppa till rätt plats i avbrottsvektorn. Innan hoppet till avbrottsvektorn sparas automatiskt flyttalsregistren a0-a3, programräknaren, kontrollregistret DUAC, samt övriga interna tillstånd för pipelinen i skuggregister. Allt detta sker inom en cykel, dvs 80 ns. Förutom programräknarens skuggregister (PCSH) är skuggregisterna osynliga för programmeraren. Behövs fler register sparas får programmeraren själv göra detta.

Innan ett hopp tillbaka till det avbrutna programmet sker, måste först egna sparade register hämtas tillbaka, därefter skall instruktionen *ireturn* exekveras. *ireturn* återställer alla de automatiskt sparade tillstånden i signalprocessorn från skuggregisterna. En viktig sak att komma ihåg är att en DA instruktion **måste** följas av minst tre CA instruktioner innan ett anrop av *ireturn* sker. Anledningen är att en DA-instruktion fyller i pipelinen, vilken sedan måste tömmas innan den återställs av skuggregisterna.

4.3 Avbrottshanterare i C++

Då det inte finns några realtidsprimitiver som tar hand om avbrott i C++ har vi fått skriva egna rutiner för detta. Ett sätt att göra detta är att skriva ett *asm*-makro som tar hand avbrottet. Detta medför att man får en mycket snabb avbrottsrutin, 0.16 μs inklusive overhead. En sådan rutin blir dock inte speciellt generell. Därför har vi valt att göra på ett annat sätt. Vi ville nämligen kunna skriva avbrottsrutinen som en funktion i C++.

Vår lösning innebär att programkoden i C++ skall ha ett utseende enligt lista 4.1.

```
#include "../intr.c"

void obe_interrupt() {
  ......
  ......
}

void init() {
  ......
  intr::set_obe_intr_addr(&obe_interrupt);
  intr::EI_obe();
  ......
}

main(){
  init();
  ......
}
```

Lista 4.1 Exempel på C++ kod som utnyttjar assemblermakron

I funktionen `void obe_interrupt()` skrivs den kod som skall exekveras vid avbrott. När man som här skriver sin avbrottsrutin som en C++ funktion, behövs inte registrena r5-r12 sparas eftersom signalprocessorns kompilator automatiskt sparar de register som används av funktionen.

`set_obe_intr_addr(&obe_interrupt)` är ett *asm*-makro, som expanderas till kodmodifierande kod vilken går in och ändrar adressen hos en *call*-instruktion, så att den istället för att anropa en dummy, skall anropa den adress i minnet där funktionen `void obe_interrupt()` börjar.

`EI_obe()` gör obe-avbrottet aktivt, dvs sätter bit 11 i pcw-registret till en etta.

Startfil crt_32c.s

När ett program kompileras med DSP32C's C-kompilator, kommer denna att automatiskt lägga till en speciell startfil med assemblerkod i början av programmet. Filens namn är `../dsp/lib/crt0_32c.s` och innehåller dels kod som behövs för att signalprocessorn skall starta exekveringen på rätt sätt, samt kod (med bl a en avbrottsvektor definierad) för debuggern *rtpi*. *rtpi* använder external interrup 1, vilket medför att detta avbrott ej kan utnyttjas till egna avbrottsfunktioner. Med hjälp av `../dsp/lib/crt0_32c.s` som grund har en två nya startfiler skapats. Filerna har utökats med den kod som behövs för avbrottshanteringen. Den ena filen `initsim.s` (Appendix D) skall används vid simuleringen, medan den andra `initvme.s` vid exekvering i målsystemet. Det är endast den första raden i avbrottsvektorn som skiljer filerna. I `initsim.s` skall INTREQ1 anropa rutinen `_dointrpt`, medan i `initvme.s` kommer `ireq1` att anropas, se lista 4.1. Genom en *environment*-variabel DSPEXEC som sätts till *vme* eller *sim* väljs automatiskt rätt startfil av *dspCC*.

Avbrottsvektor

Först har avbrottsvektorn modifierats till:

```
ivtable:                    /* Interrupt vector ivtable */
  goto ireq1; nop           /* External interrupt 1: */
  goto pd_f;  nop           /* PIO Buffer Full */
  goto pd_e;  nop           /* PIO Buffer Empty */
  goto ib_f;  nop           /* SIO Input Buffer Full */
  goto ob_e;  nop           /* SIO Output Buffer Empty */
  goto ireq2; nop           /* External interrupt 2: */
```

Lista 4.2 Avbrottsvektorn i filen initvme.s

Avbrottsrutiner i C++

Antag sedan attavbrottsrutinen obe skall skrivas. I den skall först de register som inte sparas automatiskt läggas på stacken. Därefter skall återhoppsadressen läggas i r18e (rpe) innan avbrottsfunktionen i C++ anropas med

```
call funktionsnamn (rp)
```

Att en dummy anropas enligt lista 4.3 beror på att adressen till avbrottsfunktionen är okänd före kompileringen av C++ programmet. När sedan C++ funktionen exekverat klart, hämtas de stackade värdena tillbaka. 3*nop tömmer pipelinen innan `ireturn` anropas. Assemblerkoden för detta visas i lista 4.3.

```
ob_e:                   /* SIO Output Buffer Empty */
  *sp++ = r1e
  *sp++ = r2e
  *sp++ = r3e
  *sp++ = r4e
  *sp++ = rpe
  2*nop
  rpe = pc + 8          /* Return addres */
  call dummy (rp)       /* To be changed by                    */
  nop                   /* set_obe_intr_addr(&obe_interrupt) */
  sp -= 4
  rpe = *sp--
  r4e = *sp--
  r3e = *sp--
  r2e = *sp--
  r1e = *sp
  3*nop
ireturn
```

Lista 4.3 Avbrottsrutin i filen initvme.s

Assembler-instruktionen `call dummy (rp)` kommer att ändras till

```
call obe_interrupt (rp)
```

av *asm*-makrot: `set_obe_intr_addr(addr)`, vilket visas i lista 4.4.

```
asm void set_obe_intr_addr(addr)
{
% reg addr ;
    r2e = ob_e
    r2 += 32
    *r2 = addr
    @W

% mem addr ;
    r1e = addr
    r1e = *r1
    r2e = ob_e
    r2 += 32
    *r2 = r1
    @W

% error
}
```

Lista 4.4 *asm*-makro för kodmodifierande kod

Ändring av call-instruktionens anropsadress

Genom att byta ut de 24 minst signifikanta bitarna i `call`-instruktionen kan dess destinationsadress ändras. Detta sker med hjälp av följande *asm*-makro:

4.4 Exempel på användnig av avbrott

Inom detta examensarbe används avbrotten till en f/b scheduler. Schedulerns uppgifter består i att förutom att exekverar reglerloopar, även att vara ett gränssnitt mot en värddator (M68030 kortet) där operatören kan kommunicera med signalprocessorn. Se vidare kapitel 6.

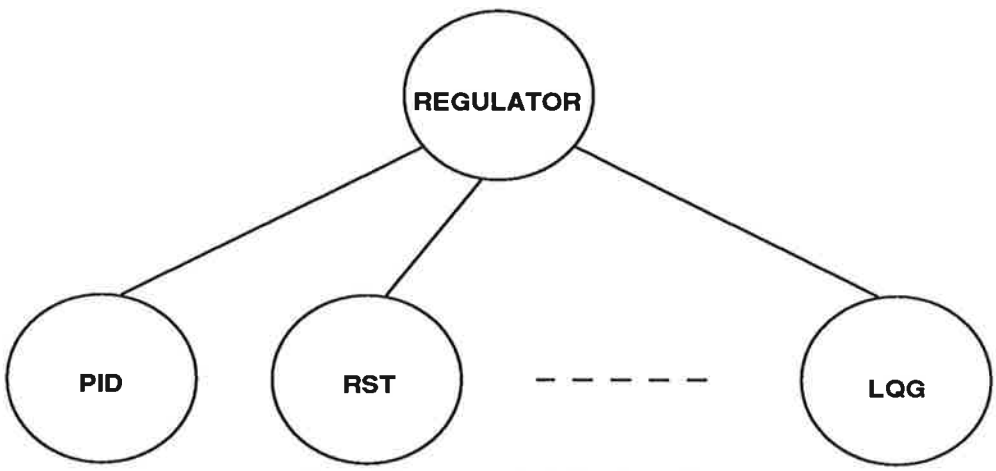
5. Objekt Orienterad Programmering av Regleralgorimer

5.1 Inledning

Med objektorienterad programmerings tre huvudegenskaper, inkapsling, ärvning samt polyformism kan en hierarki av klasser byggas upp. Överst implementeras en basklass. Från denna kan underklasser skapas. Dessa klasser ärver data och metoder från basklassen. En underklass kan sin tur vara basklass till en ny underklass, osv. Vid arv är det tillåtet att längre ner i klasshierarkin omdefiniera metoder, dvs det är tillåtet att längre ned i klasshierarkin deklarera en metod med samma namn som förkommer i någon basklass. Genom att deklarera en metod som virtuell i basklassen, blir det möjligt att knyta olika metoder med samma namn till de olika underklasserna på ett sådant sätt att rätt metod tillämpas på den underklass som anropas vid exekveringen. Genom att tilldela en virtuell funktion värdet 0, visar man att funktionen är odefinierad i klassen. Detta är speciellt för C++ och kallas *pure virtual function.*

5.2 Klasstruktur

Genom att uttnyttja ovanstående kan en basklass, regulator implementeras. Som underklasser kan sedan olika typer av regulatorer skapas.



Figur 5.1 Exempel på klassheirarki

Exempel på hur en basklass kan se ut:

```
class regulator {
public:
  regulator();
  virtual ~regulator();
  virtual void control()=0;
};
```

Lista 5.1 Exempel på basklass

Här har destruktorn `~regulator` deklarerats som `virtual` för att underklassens destruktor skall anropas när objektet skall tas bort. Dessutom är funktionen `control()` definierad som en *pure virtual function*, därför att den inte skall implementeras här utan i en underklass till basklassen `regulator`. Att så sker kontrolleras vid kompileringen.

Exempel på underklasser:

```
class PID : public regulator {
public:
  PID(float, float,.......):regulator();
  ~PID():();
  void control();
};

class RST : public regulator {
public:
  RST(poly,poly,....):regulator();
  ~RST():();
  void control();
};
```

Lista 5.2 Exempel på underklasser

5.3 PID-reglering

Inom examensarbetet har endast en klass för PID-reglering implementerats (Appendix E).

Regulatoralgoritm

Den algoritm som utnyttjats är:

$$u(t) = k\left(e(t) + \frac{1}{T_i}\int e(s) + T_d\frac{de}{dt}\right)$$

där u är styrsignal, k är förstärkningen, $e = r - y$ felet, r är referensvärdet , och y är det uppmätta värdet.

För att få en enklare beräkning har formeln skrivits om till:

$$
\begin{aligned}
P &= k(br - y) \\
D &= a_d D - b_d(y - y_{old}) \\
v &= P + I + D \\
I &= I + b_i(r - y) + b_r(u - v) \\
y_{old} &= y
\end{aligned}
$$

Parametern b används till att minska överslängar vid kraftiga steg i insignalen. Regulatorn är också försedd med anti-windup:

$$
u = \begin{cases} u_{min}, & \text{om } v < u_{min} \\ v, & \text{om } u_{min} \leq v \leq u_{max} \\ u_{max}, & \text{om } v > u_{max} \end{cases}
$$

Regulatorns koefficienter beräknas med hjälp av regulators parametrar:

$$
b_i = \frac{kh}{T_i} \qquad a_d = \frac{t_d}{T_d + Nh} \qquad b_r = \frac{h}{t_r} \qquad b_d = \frac{kT_dN}{T_d + Nh}
$$

där h är sampligstiden, T_i integrationstiden, T_r följningstiden, T_d deriveringstiden, och N är den maximala förstärkningen av deriveringen.

Specialiseringar

Börvärdesändringar. För att minska inverkan av stora ändringar av börvädet pga "multi-rate" sampling, kommer detta att räknas upp i n antal steg innan det kommer upp till sitt rätta värde.

Plotdata. Är en dataarea som skall kopieras ut till DRAM-minnet, där sedan värddatorn hämtar dem för vidare behandlig med t ex matlab.

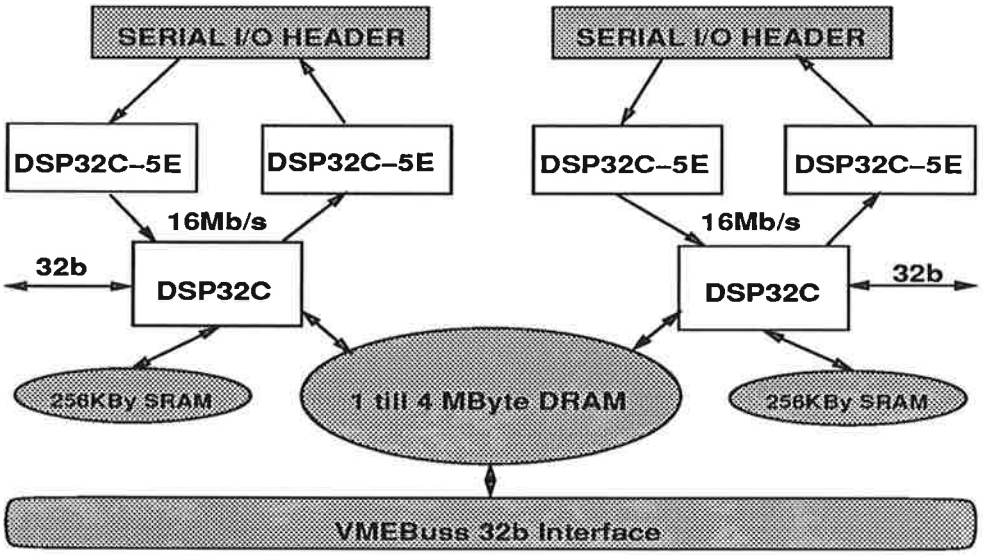
Copydata. En dataarea som omfattar regulatorparametrarna. Hit kopieras nya parametrar som lagts i DRAM-minnet av en värddator.

6. Systemintegration och Igångkörning

I det här kapitlet beskrivs i stora drag hur signalprocessorn har integrerats i ett system, AT&T's SURFboard. Dessutom beskrivs ett program som körts i simulatorn.

6.1 AT&T's SURFboard

Kortet består av fyra sammanbundna delsektioner, se figur 6.1, två DSP32C kluster, DRAM krets samt VMEbussens gränssnitt. De båda klustren är identiskt lika, och består av tre DSP32C processorer, minne och kontrollogik.



Figur 6.1 "SURFboard" arkitektur och datavägar

DSP32C Huvudprocessorn är en DSP32C som har 6KByte inre minne och ett externt minnesgränssnitt. Det interna minnet är komplementerat med ett SRAM på 256KByte, kontrollogik, tranceiver och buffertar till gränssnittet mot DRAM-minnet, semaforregister, och en 32 bitars parallell I/O port. Semaforregistret gör att DSP32C kan skicka interrupt till varandra, sätta interrupt mot VMEbussen, och ta emot interrupt från VMEbussen.

DSP32C-5E används till att för- resp efter-behandla realtidsdata (t ex förfiltrering respektive första ordningens hållkrets) som överföres via serieportarnas "header block". DSP32C-5E har inget yttre minne, och kan därför inte ha program på större än 8 KByte. samtidigt. En speciell krets sköter om fördelningen.

PIO-Portar De 32 bitars parallellportarna enligt figur 6.1 som kan nås från huvudprocessorerna är minnesmappade. Dessutom finns för samtliga sex

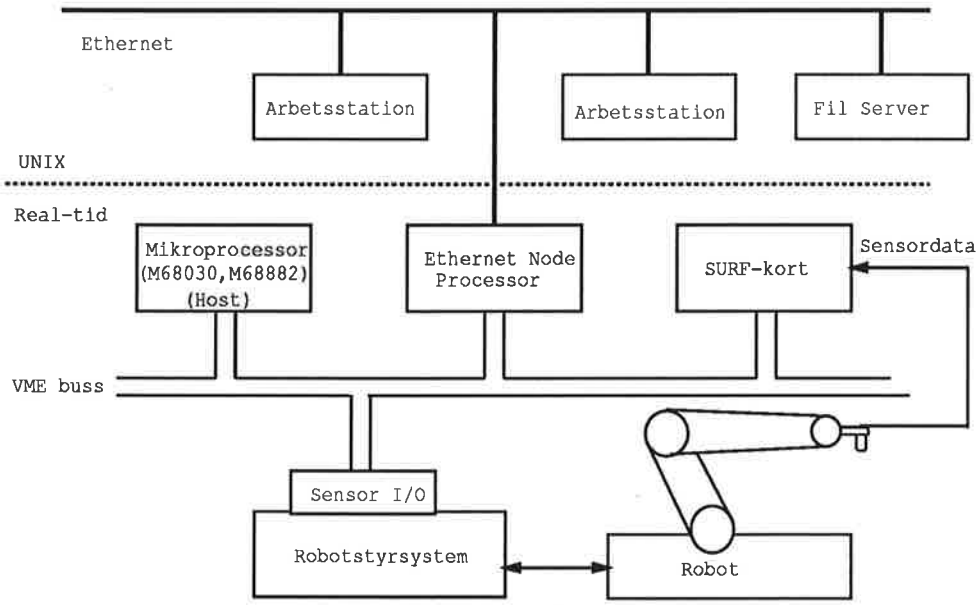
DSP:er parallellportar och kontrollregister, vilka tillsammans bildar SURF-kortets så kallade PIO-fil. PIO-filen nås från VME-bussen och används för programladdning, initiering, etc. PIO-filen visas ej figur 6.1.

Programmerarens Modell av gränssnittet mot VME-bussen

Man har direkt access till DSP32C's PIO-portar, till modulens kontrollregister och hela DRAM-arean. Genom PIO-portarna laddas programmen ner i signalprocessorerna. Med hjälp av kontrollregistrena styrs de olika processorerna. DRAM-arean kan ses som 1 MByte "bank" av ostruktruerat minne.

SURF-kortet i systemet

Kommunikation med SURF-kortet sker med en arbetsstation som via mikroprocessorn kommuniserar med SURF-kortet, genom olika kommandon. Denna uppdelning är för att skilja realtids-"världen" från UNIX-"världen".



Figur 6.2 System

Kommunikation med SURF-kortet

Vid överföring av data mellan SURF-kort och värddator har jag valt att använda en del av DRAM-minnet, eftersom denna minnesarea kan adresseras direkt (utan att gå via PIO-filen) från värddator. DRAM-minnet konfigureras i en separatkompilerad fil bestående av de variabler som behövs vid kommunikation med värddatorn (Host i figur 6.2). Variablerna styrs dit med hjälp av en mapfil. Om SURF-kortet vill kommunisera med värddatorn, sker detta med ett avbrott, som genereras genom en skrivning i pir-registret. Genom att läsa i pir-registret, kan sedan värddatorn avgöra vad som skall göras. Kommunikation åt andra hållet sker med hjälp av INTREQ1-avbrottet, som sätts när värddatorn lagt ut nya data i DRAM-minnet. Denna lösning har valts för att på enklast möjliga sätt överföra data, men fungerar endast för de två DSPer som kan addressera DRAM-minnet. Ett generellt koncept skulle troligen innebära att data till DSP:n skickas via PRD-registret, och att dataöverföringen göres med meddelandeprimitiver (mail-box).

6.2 Simulering

För att prova den skrivna PID-regulatorn har ett program skrivits (Appendix F). I programmet skapas först 6 stycken PID-regulatorer som skall motsvara de olika axlarna hos en robot. I programmet används två olika avbrottsfunktioner. Med avbrottet INTREQ1, som anropar funktionen `ext1_interrupt`, hämtar signalprocessorn nya värden resp parametrar till regulatorerna från DRAM-minnet. Reglerloopen startas genom att funktionen `void ibf_interrupt()`, som anropas varje gång som ett IBF-avbrottet sätts. IBF sätts vid regelbundna tidsintervall med hjälp av DSP32C's interna klocka. `printf()` används för att kunna kontrollera vad programmet utför. Simulering av en mätsignal sker genom att hämta in värden genom den seriella I/O-porten från en fil `sis.d`. Huvudprogrammet utför endast en "dummy"-loop.

Resultat

Utskriften visar att programmet exekverar som det är tänkt. Regleralgoritmen inklusive I/O via DSP:ns portar tar 477 cykler att exekvera, vilket motsvarar en samplingsfrekvens på över 26 kHz.

6.3 Exekvering i målsystem

Efter uttestning i simulatorn skall programmet vara felfritt, och skall kunna exekveras i målsystemet. Som avslutning har detta verifierats, medan studium av vilka reglertekniska fördelar som snabb sampling ger, fallit utanför detta arbetes tidsramar.

För att minimera de reglertekniska inslagen har endast P-reglering av ett mekaniskt servo utförts (programmet i finns i appendix G). Då servots givarsignaler inte anpassats och kopplats till DSP:ns portar, har all I/O gått via värddatorn i VME-systemet (M68k). Börvärde och parametrar skrivs ner i DRAM-minnet av M68k, som sedan genererar avbrott till DSP:n, vilket gör att avbrottsrutinen `host_interrupt` exekveras. Denna hämtar in värdena från DRAM-minnet till regulatorn. Då DSP32C vill sampla respektive ställa ut styrsignal, genereras avbrott till M68k som via PIO-filen och DRAM-minnet kommuniserar med DSP32C, och via I/O-kort på bussen kommuniserar med servot. Detta medför en långsam sampling av reglersystemet jämfört med om SURF-kortet hade skött detta. Det praktiska provet visade en samplingsfrekvens på 1.3 kHz.

De program för M68k, och de kommandon för Sun-arbetsstationerna, som krävs för att hantera SURF-kortet har utvecklats av min handledare, och beskrivs inte i denna rapport.

Exekveringen i målsystemet har visat att programmering i C++ (och assembler), test och felsökning i simulatormiljö, samt nedladdning av exekverbart program fungerar bra. Givarsignalerna måste dock kopplas direkt till DSP:ns portar om dess prestanda skall kunna utnyttjas fullt ut.

7. Referenser

LIPPMAN, STANLEY B. (1989): *C++ Primer*, Addison-Wesley, USA.

AT&T (1988): *WE DSP32 and DSP32C Support Software Library User Manual*, The AT&T Documentation Management Organization, USA.

AT&T (1988): *WE DSP32C Digital Signal Processor*, The AT&T Documentation Management Organization, USA.

AT&T (1988): *WE DSP32 and DSP32C C Language Compiler User Manual*, The AT&T Documentation Management Organization, USA.

AT&T (1988): *WE DSP32 and DSP32C C Language Compiler Library Reference Manual*, The AT&T Documentation Management Organization, USA.

KAPILOW, DAVID (1990): *Pi User's Manual*, AT&T Bell Laboratories, USA.

AT&T (1988): *The Owners's Guide To SURFboard, Yet Another VMEbus DSP Module*, AT&T Bell Laboratories, USA.

AT&T (1988): *The programmer's Guid to SURFboard: dc Device Driver Interfaces and Low-Level Support Software*, AT&T Bell Laboratories, USA.

BRÜCK, DAG M. (1988): "A Foreground/Backgound Real-Time Scheduler for the IBM AT TFRT-7393," Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅSTRÖM, K. J. (1987): "Implemation of PID Regulators TFRT-7344," Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Appendix

A. Inkapsling av klassen io

Listing of io.c

```
//  FILE:         io.c
//
//  DESCRIPTION:  include file
//
//                Contains functions for I/O
//                services

asm("#include \"/sperry/ulfm/dsp/lib/io.asm\"");
#include "/sperry/ulfm/dsp/lib/~d3cc_io.H"

//  The first line will be modified to
//  #include "/sperry/ulfm/dsp/lib/io.asm"
//  before call of d3cc
//
//  The second line includes extern "C" declarations
//  All lines where ~d3cc_* is included will be
//  removed before call of d3cc

class io {
  //  The actual implementation is in io.asm in the standard include
  //  library. That implementation requires some actual parameters
  //  beeing a constant, otherwise d3cc will give an error. Such
  //  arguments are named con_val below.
public:

  /* CONTROL REGISTER INITIALIZATION */

  static void set_dauc(const int con_val) {
    ::set_duac(con_val);
  }
  static void set_ioc(const int con_val) {
    ::set_ioc(con_val);
  }

  /* I/O SYNCRONIZATION */

  static int wait_on_pif() {
    return ::wait_on_pif();
  }
  static int wait_on_pie() {
    return ::wait_on_pie();
  }
  static int wait_on_ibf() {
    return ::wait_on_ibf();
```

```
  }
  static int wait_on_ibe() {
    return ::wait_on_ibe();
  }
  static int wait_on_obf() {
    return ::wait_on_obf();
  }

  /* I/O READ AND WRITE */

  static void put_obuf(const int val) {
    ::put_obuf(val);
  }
  static int get_ibuf() {
    return ::get_ibuf();
  }
  static int get_pdr() {
    return ::get_pdr();
  }
  static void put_pdr(const int val) {
    ::put_pdr(val);
  }
  static int get_pir() {
    return ::get_pir();
  }
  static void put_pir(const int val) {
    ::put_pir(val);
  }
};
```

Listing of ~d3cc_io.H

```
//  FILE:         ~d3cc_io.H
//
//  DESCRIPTION:  "header" file
//
//                Contains extern "C" declarations
//                for the file io.c

extern "C" void set_duac(const int);
extern "C" void set_ioc(const int);
extern "C" int get_ibuf();
extern "C" int wait_on_pif();
extern "C" int wait_on_pie();
extern "C" int wait_on_ibf();
extern "C" int wait_on_ibe();
extern "C" int wait_on_obf();
extern "C" int get_pdr();
extern "C" int get_pir();
extern "C" void put_obuf(int);
extern "C" void put_pdr(int);
extern "C" void put_pir(int);
```

Listing av io.asm

```
/*  FILE:         io.asm
//
//  DESCRIPTION:  file containing asm macros
//
//                Used by io.c
*/

#define NO_MEM_DELAY asm("@NO_MEM_DELAY");

/* CONTROL REGISTER INITIALIZATION */

asm void set_dauc(con_val)
{
% con con_val ;
dauc = con_val

% error
}

asm void set_ioc(con_val)
{
% con con_val ;
ioc = con_val

% error
}

/* I/O SYNCRONIZATION */

asm int wait_on_pif()
{
% lab l1;
l1:
if( pif ) goto l1
nop
}

asm int wait_on_pie()
{
% lab l1;
l1:
if( pie ) goto l1
nop
}

asm int wait_on_ibf()
{
% lab l1;
l1:
if( ibf ) goto l1
nop
}

asm int wait_on_ibe()
{
```

```
% lab l1;
l1:
if( ibe ) goto l1
nop
}

asm int wait_on_obf()
{
% lab l1;
l1:
if( obf ) goto l1
nop
}

/* I/O READ AND WRITE */

asm void put_obuf(val)
{
% reg val ;
obuf = val

% con val ;
r1 = val
obuf = r1
@W

% mem val ;
r1 = val
r1 = *r1
nop
obuf = r1
@W

% error
}

asm int get_ibuf()
{
% lab wait_loop ;

wait_loop:
if(ibe) goto wait_loop
nop

r1 = ibuf
@W
}

asm int get_pdr()
{
r1 = pdr
@W
}

asm void put_pdr(val)
{
% reg val ;
```

```
pdr = val

% con val ;
r1 = val
pdr = r1
@W

% mem val ;
r1 = val
r1 = *r1
nop
pdr = r1
@W

% error
}

asm int get_pir()
{
r1 = pir
@W
}

asm void put_pir(val)
{
% reg val ;
pir = val

% con val ;
r1 = val
pir = r1
@W

% mem val ;
r1 = val
r1 = *r1
nop
pir = r1
@W

% error
}
```

B. Inkapsling av klassen dsp2ieee

Listing of dsp2ieee.c

```
//  FILE:         dsp2ieee.c
//
//  DESCRIPTION:  include file
//
//                Copies data from one location in the memory
//                to another location, during a conversion
//                from one standard to another standard

asm("#include \"/sperry/ulfm/dsp/lib/dsp2ieee.asm\"");
#include "/sperry/ulfm/dsp/lib/~d3cc_dsp2ieee.H"

//  The first line will be modified to
//  #include "/sperry/ulfm/dsp/lib/dsp2ieee.asm"
//  before call of d3cc
//
//  The second line includes extern "C" declarations
//  All lines where ~d3cc_* is included will be
//  removed before call of d3cc

class dsp2ieee {
public:
  static void dsp2ieee_cp(register int s,
                          register float *d,
                          register int len) {
    //  Copies data from the memory location at address s
    //  to the memory address that d points out, during a
    //  conversion from dsp standard to ieee standard
    ::dsp2ieee_cp(s, d, len);
  }

  static void ieee2dsp_cp(register float *s,
                          register int d,
                          register int len) {
    //  Copies data from the memory address that s points out
    //  to the memory location at address d, during a conversion
    //  from ieee standard to dsp standard
    ::ieee2dsp_cp(s, d, len);
  }

  static float ieee2dsp_cp_ref(register float *s) {
    //  Fetch data from the memory address that s points out,
    //  during a conversion from ieee standard to dsp standard
    return ::ieee2dsp_cp_ref(s);
  }
};
```

Listing of ~d3cc_dsp2ieee.H

```
//  FILE:         ~d3cc_dsp2ieee.H
//
//  DESCRIPTION:  "header" file
//
```

```
//                Contains extern "C" declarations
//                for the file dsp2ieee.c

extern "C" void  dsp2ieee_cp(register int,
                             register float*,
                             register int);
extern "C" void  ieee2dsp_cp(register float*,
                             register int,
                             register int);
extern "C" float ieee2dsp_cp_ref(register float*);
```

Listing av dsp2ieee.asm

```
/*  FILE:         dsp2ieee.asm
//
//  DESCRIPTION:  file containing asm macros
//
//                Used by dsp2ieee.c
*/

asm void dsp2ieee_cp(s, d, len)
{
%       lab l1; reg s, d, len;
l1:
        if(len-- >= 0) goto l1
        *d++ = a0 = ieee(*s++)
        @W

%       lab l1; reg s, d; con len;
        r3e = len
l1:
        if(r3-- >= 0) goto l1
        *d++ = a0 = ieee(*s++)
        @W

%       lab l1; reg s, d; mem len;
        r3e = len
        r3e = *r3
l1:
        if(r3-- >= 0) goto l1
        *d++ = a0 = ieee(*s++)
        @W

%       lab l1; mem s; reg d, len;
        r2e = s
        r2e = *r2
l1:
        if(len-- >= 0) goto l1
        *d++ = a0 = ieee(*r2++)
        @W

%       lab l1; mem d; reg s, len;
        r1e = d
        r1e = *r1
l1:
        if(len-- >= 0) goto l1
        *r1++ = a0 = ieee(*s++)
```

```
        @W

%       lab l1; reg d; mem s; con len;
        r2e = s
        r2e = *r2
        r3e = len
l1:
        if(r3-- >= 0) goto l1
        *d++ = a0 = ieee(*r2++)
        @W

%       lab l1; reg d; mem s,len;
        r2e = s
        r2e = *r2
        r3e = len
        r3e = *r3
l1:
        if(r3-- >= 0) goto l1
        *d++ = a0 = ieee(*r2++)
        @W

%       lab l1; reg s; mem d; con len;
        r1e = d
        r1e = *r1
        r3e = len
l1:
        if(r3-- >= 0) goto l1
        *r1++ = a0 = ieee(*s++)
        @W

%       lab l1; reg s; mem d, len;
        r1e = d
        r1e = *r1
        r3e = len
        r3e = *r3
l1:
        if(r3-- >= 0) goto l1
        *r1++ = a0 = ieee(*s++)
        @W

%       lab l1; mem s, d; reg len;
        r1e = d
        r1e = *r1
        r2e = s
        r2e = *r2
l1:
        if(len-- >= 0) goto l1
        *r1++ = a0 = ieee(*r2++)
        @W

%       lab l1; mem s, d; con len;
        r1e = d
        r1e = *r1
        r2e = s
        r2e = *r2
        r3e = len
l1:
```

```
        if(len-- >= 0) goto l1
        *r1++ = a0 = ieee(*r2++)
        @W

%       lab l1; mem s, d, len;
        r1e = d
        r1e = *r1
        r2e = s
        r2e = *r2
        r3e = len
        r3e = *r3
l1:
        if(len-- >= 0) goto l1
        *r1++ = a0 = ieee(*r2++)
        @W

%       error

/* end of asm void dsp2ieee_cp(s, d, len) */
}

asm void ieee2dsp_cp(s, d, len)
{
%       lab l1; reg s, d, len;
l1:
        if(len-- >= 0) goto l1
        *d++ = a0 = dsp(*s++)
        @W

%       lab l1; reg s, d; con len;
        r3e = len
l1:
        if(r3-- >= 0) goto l1
        *d++ = a0 = dsp(*s++)
        @W

%       lab l1; reg s, d; mem len;
        r3e = len
        r3e = *r3
l1:
        if(r3-- >= 0) goto l1
        *d++ = a0 = dsp(*s++)
        @W

%       lab l1; reg s, len; con d;
        r1e = d
l1:
        if(len-- >= 0) goto l1
        *r1++ = a0 = dsp(*s++)
        @W

%       lab l1; reg s, len; mem d;
        r1e = d
        r1e = *r1
l1:
        if(len-- >= 0) goto l1
        *r1++ = a0 = dsp(*s++)
```

```
        @W

%       lab l1; mem s; reg d, len;
        r2e = s
        r2e = *r2
l1:
        if(len-- >= 0) goto l1
        *d++ = a0 = dsp(*r2++)
        @W

%       lab l1; reg s; mem d; con len;
        r1e = d
        r1e = *r1
        r3e = len
l1:
        if(r3-- >= 0) goto l1
        *r1++ = a0 = dsp(*s++)
        @W

%       lab l1; reg s; mem d, len;
        r1e = d
        r1e = *r1
        r3e = len
        r3e = *r3
l1:
        if(r3-- >= 0) goto l1
        *r1++ = a0 = dsp(*s++)
        @W

%       lab l1; reg d; mem s; con len;
        r2e = s
        r2e = *r2
        r3e = len
l1:
        if(r3-- >= 0) goto l1
        *d++ = a0 = dsp(*r2++)
        @W

%       lab l1; reg d; mem s, len;
        r2e = s
        r2e = *r2
        r3e = len
        r3e = *r3
l1:
        if(r3-- >= 0) goto l1
        *d++ = a0 = dsp(*r2++)
        @W

%       lab l1; mem s, d; reg len;
        r1e = d
        r1e = *r1
        r2e = s
        r2e = *r2
l1:
        if(len-- >= 0) goto l1
        *r1++ = a0 = dsp(*r2++)
        @W
```

```
%       lab l1; mem s, d; con len;
        r1e = d
        r1e = *r1
        r2e = s
        r2e = *r2
        r3e = len
l1:
        if(r3-- >= 0) goto l1
        *r1++ = a0 = dsp(*r2++)
        @W

%       lab l1; mem s, d, len;
        r1e = d
        r1e = *r1
        r2e = s
        r2e = *r2
        r3e = len
        r3e = *r3
l1:
        if(r3-- >= 0) goto l1
        *r1++ = a0 = dsp(*r2++)
        @W

%       error
/* End of asm void ieee2dsp_cp(s, d, len) */
}



asm float ieee2dsp_cp_ref(s)
{
%       reg s;
        a0 = dsp(*s)
        @W

%       mem   s;
        r1e = s
        r1e = *r1
        a0 = dsp(*r1)
        @W

%       error
}
```

C. Inkapsling av klassen intr

Listing of intr.c

```
//  FILE:         intr.c
//
//  DESCRIPTION:  include file
//
//                Contains functions for interrupt
//                services

asm("#include \"/sperry/ulfm/dsp/lib/intr.asm\"");
#include "/sperry/ulfm/dsp/lib/~d3cc_intr.H"

//  The first line will be modified to
//  #include "/sperry/ulfm/dsp/lib/intr.asm"
//  before call of d3cc
//
//  The second line includes extern "C" declarations
//  All lines where ~d3cc_* is included will be
//  removed before call of d3cc

class intr {
public:
  static int set_pcw(const int con_val) {
    //  The pcw register will be set to con_val
    return ::set_pcw(con_val);
  }

  static int get_pcw() {
    //  Returns the value of the pcw register
    return ::get_pcw();
  }

  static void set_ext1_intr_addr(register void *addr) {
    //  Changes the address in the interrupt routine
    //  in file start_up.s, see chapter 3
    ::set_ext1_intr_addr(addr);
  }

  static void set_pdf_intr_addr(register void *addr) {
    //  Changes the address in the interrupt routine
    //  in file start_up.s, see chapter 3
    ::set_pdf_intr_addr(addr);
  }

  static void set_pde_intr_addr(register void *addr) {
    //  Changes the address in the interrupt routine
    //  in file start_up.s, see chapter 3
    ::set_pde_intr_addr(addr);
  }

  static void set_ibf_intr_addr(register void *addr) {
    //  Changes the address in the interrupt routine
    //  in file start_up.s, see chapter 3
    ::set_ibf_intr_addr(addr);
```

```
}

static void set_obe_intr_addr(register void *addr) {
  //  Changes the address in the interrupt routine
  //  in file start_up.s, see chapter 3
  ::set_obe_intr_addr(addr);
}

static void set_ext2_intr_addr(register void *addr) {
  //  Changes the address in the interrupt routine
  //  in file start_up.s, see chapter 3
  ::set_ext2_intr_addr(addr);
}

static void EI_ireq1() {
  //  Enable intrrupt INTREQ1
  ::EI_ireq1();
}

static void EI_pdf() {
  //  Enable intrrupt PDF
  ::EI_pdf();
}

static void EI_pde() {
  //  Enable intrrupt PDE
  ::EI_pde();
}


static void EI_ibf() {
  //  Enable intrrupt IBF
  ::EI_ibf();
}

static void EI_obe() {
  //  Enable intrrupt OBE
  ::EI_obe();
}

static void EI_ireq2() {
  //  Enable intrrupt INTREQ2
  ::EI_ireq2();
}


static void DI_ireq1() {
  //  Disable intrrupt INTREQ1
  ::DI_ireq1();
}

static void DI_pdf() {
  //  Disable intrrupt PDF
  ::DI_pdf();
}

static void DI_pde() {
```

```
    // Disable intrrupt PDE
    ::DI_pde();
  }

  static void DI_ibf() {
    // Disable intrrupt IBF
    ::DI_ibf();
  }

  static void DI_obe() {
    // Disable intrrupt OBE
    ::DI_obe();
  }

  static void DI_ireq2() {
    // Disable intrrupt INTREQ2
    ::DI_ireq2();
  }
};
```

Listing of ~d3cc_intr.H

```
//  FILE:         ~d3cc_intr.H
//
//  DESCRIPTION:  "header" file
//
//                Contains extern "C" declarations
//                for the file intr.c

extern "C" int set_pcw(int);
extern "C" int get_pcw();
extern "C" void set_ext1_intr_addr(register void*);
extern "C" void set_pdf_intr_addr(register void*);
extern "C" void set_pde_intr_addr(register void*);
extern "C" void set_ibf_intr_addr(register void*);
extern "C" void set_obe_intr_addr(register void*);
extern "C" void set_ext2_intr_addr(register void*);
extern "C" void EI_ireq1();
extern "C" void EI_pdf();
extern "C" void EI_pde();
extern "C" void EI_ibf();
extern "C" void EI_obe();
extern "C" void EI_ireq2();
extern "C" void DI_ireq1();
extern "C" void DI_pdf();
extern "C" void DI_pde();
extern "C" void DI_ibf();
extern "C" void DI_obe();
extern "C" void DI_ireq2();
```

Listing of intr.asm

```
/*  FILE:          intr.asm
//
//  DESCRIPTION:   file containing asm macros
//
//                 Used by intr.c
*/

asm int set_pcw(con_val)
{
%       con     con_val ;
        r1e = con_val
        r2e = pcw
        nop
        pcw = r1
        nop
        r1e = r2e
        @W

%       error
}

asm int get_pcw()
{
        r1e = pcw
        @W
}

asm void set_ext1_intr_addr(addr)
{
%       reg addr ;
        r2e = ireq1
        r2 += 32
        *r2 = addr
        @W

%       mem addr ;
        r1e = addr
        r1e = *r1
        r2e = ireq1
        r2 += 32
        *r2 = r1
        @W

%       error
}

asm void set_pdf_intr_addr(addr)
{
%       reg addr ;
        r2e = pd_f
        r2 += 32
        *r2 = addr
        @W

%       mem addr ;
```

```
        r1e = addr
        r1e = *r1
        r2e = pd_f
        r2 += 32
        *r2 = r1
        @W

%       error
}

asm void set_pde_intr_addr(addr)
{
%       reg addr ;
        r2e = pd_e
        r2 += 32
        *r2 = addr
        @W

%       mem addr ;
        r1e = addr
        r1e = *r1
        r2e = pd_e
        r2 += 32
        *r2 = r1
        @W

%       error
}

asm void set_ibf_intr_addr(addr)
{
%       reg addr ;
        r2e = ib_f
        r2 += 32
        *r2 = addr
        @W

%       mem addr ;
        r1e = addr
        r1e = *r1
        r2e = ib_f
        r2 += 32
        *r2 = r1
        @W

%       error
}

asm void set_obe_intr_addr(addr)
{
%       reg addr ;
        r2e = ob_e
        r2 += 32
        *r2 = addr
        @W

%       mem addr ;
```

```
        r1e = addr
        r1e = *r1
        r2e = ob_e
        r2 += 32
        *r2 = r1
        @W

%       error
}

asm void set_ext2_intr_addr(addr)
{
%       reg addr ;
        r2e = ireq2
        r2e += 32
        *r2 = addr
        @W

%       mem addr ;
        r1e = addr
        r1e = *r1
        r2e = ireq2
        r2e += 32
        *r2 = r1
        @W

%       error
}

asm void EI_ireq1()
{
        r1e = pcw
        nop
        r1e |= 0x8000
        pcw = r1
        @W
}

asm void EI_pdf()
{
        r1e = pcw
        nop
        r1e |= 0x4000
        pcw = r1
        @W
}

asm void EI_pde()
{
        r1e = pcw
        nop
        r1e |= 0x2000
        pcw = r1
        @W
}

asm void EI_ibf()
```

```
{
        r1e = pcw
        nop
        r1e |= 0x1000
        pcw = r1
        @W
}

asm void EI_obe()
{
        r1e = pcw
        nop
        r1e |= 0x0800
        pcw = r1
        @W
}

asm void EI_ireq2()
{
        r1e = pcw
        nop
        r1e |= 0x0400
        pcw = r1
        @W
}

asm void DI_ireq1()
{
        r1e = pcw
        nop
        r1e &= 0x7fff
        pcw = r1
        @W
}

asm void DI_pdf()
{
        r1e = pcw
        nop
        r1e &= 0xbfff
        pcw = r1
        @W
}

asm void DI_pde()
{
        r1e = pcw
        nop
        r1e &= 0xdfff
        pcw = r1
        @W
}

asm void DI_ibf()
{
        r1e = pcw
        nop
```

```
        r1e &= 0xefff
        pcw = r1
        @W
}

asm void DI_obe()
{
        r1e = pcw
        nop
        r1e &= 0xf7ff
        pcw = r1
        @W
}

asm void DI_ireq2()
{
        r1e = pcw
        nop
        r1e &= 0xfbff
        pcw = r1
        @W
}
```

D. Startfilen vmeinit.s assemblerkod

```
/*********************************************************************
 *                                                                   *
 *      FILE:           initvme.s                                    *
 *                                                                   *
 *      DESCRIPTION:    A run-time startoff file for DSP32C          *
 *                      programs.                                    *
 *                      Modified from ../dsp/lib/crt0_32c.           *
 *                                                                   *
 *      $Compile:       d3cpp initvme.s > initvme.i;                 *
 *                      d3cc -Q -c initvme.i; rm initvme.i           *
 *                                                                   *
 *********************************************************************/
#define EXTIRQ1         0x8000
#define MEMWAITS        0x000D
#define REGSAVE         _regsave

#define GO_RUN          1       /* Start executing                */
#define GO_BRKSTEP      2       /* Step out of breakpoint handler */

#define STAT_RUNNING    0       /* Program is running             */
#define STAT_INTRPT     1       /* In interrupt handler           */
#define STAT_BRKPT      2       /* In breakpoint handler          */
#define STAT_BRKSTEP    3       /* Stepping after a breakpoint    */
#define STAT_HANG       4       /* Loaded, but not started        */
#define STAT_STOPPED    5       /* Stopped, really STAT_INTRPT    */
#define STAT_EXITED     6       /* Program exited                 */
#define STAT_RESET      7       /* Processor was reset            */
#define STAT_NODBMON    8       /* Debug monitor not present      */

#define fp      r13             /* frame pointer                  */
#define sp      r14             /* stack pointer                  */
#define spe     r14e            /* stack pointer                  */
#define rp      r18             /* return pointer                 */
#define rpe     r18e            /* return pointer                 */
#define ir      r19             /* increment register             */

#define STACK_INC       4       /* stack increment                */

.text
        .global _start
        .global _statflag
        .global _exit
        .global errno
        .global dummy
        .global ivtable
        .global ireq1
        .global pd_f
        .global pd_e
        .global ib_f
        .global ob_e
        .global ireq2

        .align  4
```

```
__start:
                goto _start
                nop
_arg0ptr:       int24 _stack    /* Pointer to arg0, filled in by
                                   loader                          */
_initsp:        int24 _stack    /* Pointer to start sp, filled in
                                   by loader                       */
_statflag:      int24 STAT_HANG /* Status of the processor         */
_goflag:        int24 0         /* Request from monitor            */
_bkpt:          goto _dobkpt
                nop
_regsave:       27 * int24
                int24 _start    /* Make it look like initial pc is
                                   _start                          */
                6 * int24
_saver1:        int24           /* Temp location to save r1 in
                                   interrupt                       */
_bsaver1:       int24           /* Temp location to save r1 in bkpt */
_bsavepcw:      int24           /* Temp location to save pcw       */

_start: *_statflag = STAT_RUNNING /* Change status to running      */
        dauc = 0                /* Set round to int                */
        ioc = 0x0cc5            /* Disable any Serial DMAs         */
        fp = 0                  /* Clear fp, for tracing           */
        spe = *_initsp          /* Stack is empty at bottom        */
        ir = STACK_INC          /* stack increment                 */
        r22e = ivtable          /* Handle interrupts               */
        r1 = EXTIRQ1|MEMWAITS
        pcw = r1                /* enable interrupts, memory wait  */
        4*nop                   /* states                          */
        call _onstart (rp)      /* Allow special startup code      */
        nop                     /* before main                     */
        call main (rp)          /* main(argc, argv)                */
        nop
        *sp++r19 = r1e
        call exit (rp)          /* exit (main return value)        */
        nop
        /* No return, but just in case fall through to _exit       */
_exit:
        r1e = sp - 4
        r1e = *r1
        r2e = STAT_EXITED
        *_statflag = r2e
        *REGSAVE = r1e          /* Store return code in r0 save area */
L5:     goto L5
        nop

.data

errno:  int

.bss    _stack,1,1

.text

dummy:
        goto rp
```

```
        nop

/*
 * The Interrupt vector table
 */

  ivtable:
        goto ireq1; nop         /* External interrupt 1:          */
        goto pd_f; nop          /* PIO Buffer Full                */
        goto pd_e; nop          /* PIO Buffer Empty               */
        goto ib_f; nop          /* SIO Input Buffer Full          */
        goto ob_e; nop          /* SIO Output Buffer Empty        */
        goto ireq2; nop         /* External interrupt 2:          */


/*
 * The interrupt handler
 */

  ireq1:                        /* External interrupt 1           */
        *sp++ = r1e
        *sp++ = r2e
        *sp++ = r3e
        *sp++ = r4e
        *sp++ = rpe
        2*nop
        rpe = pc + 8
        call dummy (rp)
        nop
        sp -= 4
        rpe = *sp--
        r4e = *sp--
        r3e = *sp--
        r2e = *sp--
        r1e = *sp
        3*nop                   /* The last DA instruc-
                                   tion must be followed
                                   by three CA instructions
                                   before the ireturn.      */
  ireturn

  pd_f:                         /* PIO Buffer Full                */
        *sp++ = r1e
        *sp++ = r2e
        *sp++ = r3e
        *sp++ = r4e
        *sp++ = rpe
        2*nop
        rpe = pc + 8
        call dummy (rp)
        nop
        sp -= 4
        rpe = *sp--
        r4e = *sp--
        r3e = *sp--
        r2e = *sp--
        r1e = *sp
```

```
      3*nop
ireturn

pd_e:                          /* PIO Buffer Empty              */
      *sp++ = r1e
      *sp++ = r2e
      *sp++ = r3e
      *sp++ = r4e
      *sp++ = rpe
      2*nop
      rpe = pc + 8
      call dummy (rp)
      nop
      sp -= 4
      rpe = *sp--
      r4e = *sp--
      r3e = *sp--
      r2e = *sp--
      r1e = *sp
      3*nop
ireturn

ib_f:                          /* SIO Input Buffer Full         */
      *sp++ = r1e
      *sp++ = r2e
      *sp++ = r3e
      *sp++ = r4e
      *sp++ = rpe
      2*nop
      rpe = pc + 8
      call dummy (rp)
      nop
      sp -= 4
      rpe = *sp--
      r4e = *sp--
      r3e = *sp--
      r2e = *sp--
      r1e = *sp
      3*nop
ireturn

ob_e:                          /* SIO Output Buffer Empty       */
      *sp++ = r1e
      *sp++ = r2e
      *sp++ = r3e
      *sp++ = r4e
      *sp++ = rpe
      2*nop
      rpe = pc + 8
      call dummy (rp)
      nop
      sp -= 4
      rpe = *sp--
      r4e = *sp--
      r3e = *sp--
      r2e = *sp--
      r1e = *sp
```

```
        3*nop
    ireturn

    ireq2:                          /* External interrupt 2             */
        *sp++ = r1e
        *sp++ = r2e
        *sp++ = r3e
        *sp++ = r4e
        *sp++ = rpe
        2*nop
        rpe = pc + 8
        call dummy (rp)
        nop
        sp -= 4
        rpe = *sp--
        r4e = *sp--
        r3e = *sp--
        r2e = *sp--
        r1e = *sp
        3*nop
    ireturn

/*
 * The monitor's interrupt handler
 */
_dointrpt:                          /* Not used now, used by ireq1      */
        *_saver1 = r1e              /* store in saver1,in case we are
                                       in _dobkpt                       */
        r1e = *_statflag            /* statflag == STAT_BRKSTEP ->      */
        nop                         /* brkstep                          */
        r1e - STAT_BRKSTEP
        if (ne) goto L9             /* Stepping after breakpoint ?      */
        r19e - 4                    /* has r19 been restored to 4 ?     */
        if (eq) goto L9             /* If so, out of break routine      */
        r1e = *_saver1              /* If not, keep stepping until true */
        nop
        ireturn
L9:     *(REGSAVE + 8) = r2e
        call _regdump (r2)          /* Save the registers               */
        nop
        r1e = *_saver1
        r2e = pcshe
        r2e = r2 - 4
        r3 = pcw
        *(REGSAVE + 108) = r2e
        *(REGSAVE + 4) = r1e
        *(REGSAVE + 124) = r3
        r1e = STAT_INTRPT           /* Indicate in the interrupt routine*/
        *_statflag = r1e
L10:    r3e = *_goflag              /* Wait for go command              */
        nop
        if (eq) goto L10
        nop
        call _regrestore (r2)       /* Save the registers               */
        nop
        r1 = *(REGSAVE + 124)
        r19e = *(REGSAVE + 76)
```

```
        pcw = r1
        r2e = *(REGSAVE + 8)
        r1e = *(REGSAVE + 4)
        nop
_dummy: ireturn

/*
 * The monitor's breakpoint handler
 */
_dobkpt:
        *_bsaver1 = r1e
        r1 = pcw
        nop
        *_bsavepcw = r1
        r1e = r1 & ~EXTIRQ1     /* Mask off interrupt              */
        pcw = r1
        *(REGSAVE + 8) = r2e
        call _regdump (r2)      /* Save the registers              */
        nop
        r1e = *_bsaver1
        r2e = 4
        r3e = *_bsavepcw
        r19e = r19 - 8
        *(REGSAVE + 4) = r1e    /* r1                              */
        *(REGSAVE + 76) = r2e   /* r19                             */
        *(REGSAVE + 108) = r19e /* pc                              */
        *(REGSAVE + 124) = r3e  /* pcw                             */
        r1e = STAT_BRKPT        /* Indicate in the breakpoint
                                   routine                         */
        *_statflag = r1e
L20:    r3e = *_goflag          /* Wait for go command             */
        nop
        if (eq) goto L20
        r3e - GO_BRKSTEP
        if (ne) goto _bkptout   /* Step or continue?               */
        nop
        /* step after a breakpoint */
        *_goflag = 0            /* Clear go flag                   */
        r1e = STAT_BRKSTEP
        *_statflag = r1e        /* Inform host its OKAY to interrupt*/
L21:    r3e = *_goflag          /* Wait for go command             */
        nop
        if (eq) goto L21
        nop
_bkptout:
        call _regrestore (r2)   /* Restore the registers           */
        nop
        a0 = *r1++              /* r1 is set up by _regrestore     */
        a1 = *r1++
        a2 = *r1++
        a3 = *r1++
        r19e = *(REGSAVE + 108)
        r1 = *(REGSAVE + 124)
        r2e = *(REGSAVE + 8)
        pcw = r1
        r1e = *(REGSAVE + 4)
        return (r19)
```

```
        r19e = 4

/*
 * Dump the registers into the monitor's save area.
 */
_regdump:
        r1e = REGSAVE + 12
        *r1++ = r3e
        *r1++ = r4e
        *r1++ = r5e
        *r1++ = r6e
        *r1++ = r7e
        *r1++ = r8e
        *r1++ = r9e
        *r1++ = r10e
        *r1++ = r11e
        *r1++ = r12e
        *r1++ = r13e
        *r1++ = r14e
        *r1++ = r15e
        *r1++ = r16e
        *r1++ = r17e
        *r1++ = r18e
        *r1++ = r19e
        *r1++ = r20e
        *r1++ = r21e
        *r1++ = r22e
        *r1++ = a0 = a0
        *r1++ = a1 = a1
        *r1++ = a2 = a2
        *r1++ = a3 = a3
        nop
        return (r2)
        nop

/*
 * Restore the registers.  We don't bother with the accumulators
 * because in the interrupt handler, they are restored from the
 * shadow registers. Also, we skip r19 because the breakpoint
 * handler uses this register for the address of the breakpoint.
 * r19 must be restored to 4 in the latent branch instruction
 * after the breakpoint handlers return.                          */
_regrestore:
        *_goflag = 0    /* Clear go flag before restarting        */
        r1e = REGSAVE + 12
        r3e = *r1++
        r4e = *r1++
        r5e = *r1++
        r6e = *r1++
        r7e = *r1++
        r8e = *r1++
        r9e = *r1++
        r10e = *r1++
        r11e = *r1++
        r12e = *r1++
        r13e = *r1++
        r14e = *r1++
```

```
r15e = *r1++
r16e = *r1++
r17e = *r1++
r18e = *r1++
r1e = r1 + 4
r20e = *r1++
r21e = *r1++
r22e = *r1++
return (r2)
nop
```

E. PID-regulatorns C++ kod

```
//  FILE:         pid.c
//
//  DESCRIPTION:  Contains a PID controller
//
//  TO COMPILE:   dspCC -c pid.c

#include "/sperry/ulfm/dsp/lib/io.c"

class PIDreg {
  float r, P, I, D, u, yold;      // Plot data.
  float b, K, Tr, Ti, Td, N, h;   // Copy data.
  float from_r, to_r, bi, br, ad, bd, invNoOfItr;
  int itr, NoOfItr;
  float Ulow, Uhigh;
public:
  PIDreg(                         // Constructor
         float,    // to reduce over shoot at step changes
         float,    // gain
         float,    // tracking time
         float,    // integration time
         float,    // derivative time
         float,    // maximum derivative time
         float,    // sampling time
         float,    // anti wind up low limit
         float,    // anti wind up high limit
         int       // number of step between old and
                   // new setpoint
  );
  ~PIDreg();                      // Destructor
  void new_param();               // Computes new coefficients
  void set_ref(float);            // Sets a new set point
  void control();                 // Regulator loop
  float get_u();                  // Implemented for simulation
  float get_ref();                // Implemented for simulation
  int   get_y();                  // Implemented for simulation
  int plot_data;                  // Plot data area definition.
  int plot_data_len;
  int copy_area;                  // Copy data area definition.
  int copy_area_len;
};

PIDreg::PIDreg(
               float bii, float Ki, float Tri, float Tii,
               float Tdi, float Ni, float hi, float low,
               float high, int itr ){
  b  = bii; K = Ki; Tr = Tri; Ti = Tii; Td = Tdi;
  N  = Ni;  h = hi; Ulow = low; Uhigh = high; NoOfItr = itr;
  bi = K*h/Ti;
  br = h/Tr;
  ad = Td/(Td+N*h);
  bd = K*Td*N/(Td+N*h);
  I = D = u = from_r = to_r = r = 0;
  itr = NoOfItr;
  invNoOfItr = 1.0 / NoOfItr;
```

```
  plot_data = (int) &r;
  plot_data_len = ((int) &yold) + sizeof(float) - plot_data;
  copy_area = (int) &b;
  copy_area_len = ((int) &h) + sizeof(float) - copy_area;
}

PIDreg::~PIDreg() {
}

void PIDreg::new_param() {
  bi = K*h/Ti;
  br = h/Tr;
  ad = Td/(Td+N*h);
  bd = K*Td*N/(Td+N*h);
}

void PIDreg::set_ref(float new_ref) {
  from_r = r;
  to_r   = new_ref;
  itr    = NoOfItr;
}

void PIDreg::control()  {
  float v, new_y;
  if (itr > -1 )
    r = to_r - (to_r - from_r)*invNoOfItr*itr--;
  new_y = io::get_ibuf();  // Read input
  P = K*(b*r-new_y);
  D = ad*D-bd*(new_y-yold);
  v = P + I + D;
  I += bi*(u-v);
  yold = new_y;

  if (v < Ulow) u = Ulow;                        // anti-windup
  else if (v > Uhigh) u = Uhigh;
  else u = v;
// Set output u
}

float PIDreg::get_u() {              // Returns control variable
  return u;                          // Implemented for simulation
}

float PIDreg::get_ref() {            // Returns set point
  return r;                          // Implemented for simulation
}

int PIDreg::get_y() {                // Returns measured value
  return (int)yold;                  // Implemented for simulation
}
```

F. Simuleringsprogrammets C++ kod

```
//  FILE:         PIDs.c
//
//  DESCRIPTION:  Contains a PID controller
//
//  TO COMPILE:   dspCC -o PIDs -m dram.map meminit.o pid.o PIDs.c -lC

#include "/sperry/ulfm/dsp/lib/d3printf.H"
#include "/sperry/ulfm/dsp/lib/io.c"
#include "/sperry/ulfm/dsp/lib/intr.c"
#include "/sperry/ulfm/dsp/lib/dsp2ieee.c"
#include "pid.H"
#include "meminit.H"
#define LOOP while(1)

extern "C" { void printf(...); }

  float *sendptr;
  int i = 1;

  enum Interrupt {
         GENERAL,
         READ_1,
         OUTPUT_1
  };

  PIDreg reg1(0.8, 0.1, 1.5, 20.5, 2.0, 5, 10, -100, 100, 10);
  PIDreg reg2(0.8, 0.2, 1.5, 19.5, 2.0, 5, 10, -100, 100, 10);
  PIDreg reg3(0.8, 0.3, 1.5, 18.5, 2.0, 5, 10, -100, 100, 10);
  PIDreg reg4(0.8, 0.2, 1.5, 17.5, 2.0, 5, 10, -100, 100, 10);
  PIDreg reg5(0.8, 0.1, 1.5, 18.5, 2.0, 5, 10, -100, 100, 10);
  PIDreg reg6(0.8, 0.2, 1.5, 19.5, 2.0, 5, 10, -100, 100, 10);

void ibf_interrupt() {
  printf("IBF\n");
  reg1.control();
  reg2.control();
  reg3.control();
  reg4.control();
  reg5.control();
  reg6.control();
  if (data_fl==0) {
    if (sendptr < &send_v[0] + send_size) {
      dsp2ieee::dsp2ieee_cp(reg1.plot_data,
                            sendptr,
                            reg1.plot_data_len);
      dsp2ieee::dsp2ieee_cp(reg2.plot_data,
                            sendptr,
                            reg2.plot_data_len);
      dsp2ieee::dsp2ieee_cp(reg3.plot_data,
                            sendptr,
                            reg3.plot_data_len);
      dsp2ieee::dsp2ieee_cp(reg4.plot_data,
                            sendptr,
                            reg4.plot_data_len);
```

```
      dsp2ieee::dsp2ieee_cp(reg5.plot_data,
                            sendptr,
                            reg5.plot_data_len);
      dsp2ieee::dsp2ieee_cp(reg6.plot_data,
                            sendptr,
                            reg6.plot_data_len);
    }
    else {
      intr::EI_ireq1();      // For simulation  address 0x105c
      data_fl = 1;
      io::put_pir(GENERAL);// 0x1084
      sendptr = &send_v[0];  // reset sendptr
    }
  }
  // printf() is implemented for simulation
  printf(" set point: %f  u1=%f y= %d\n",
         reg1.get_ref(), reg1.get_u(), reg1.get_y());
  printf(" set point: %f  u2=%f y= %d\n",
         reg2.get_ref(), reg2.get_u(), reg2.get_y());
  printf(" set point: %f  u3=%f y= %d\n",
         reg3.get_ref(), reg3.get_u(), reg3.get_y());
  printf(" set point: %f  u4=%f y= %d\n",
         reg4.get_ref(), reg4.get_u(), reg4.get_y());
  printf(" set point: %f  u5=%f y= %d\n",
         reg5.get_ref(), reg5.get_u(), reg5.get_y());
  printf(" set point: %f  u6=%f y= %d\n",
         reg6.get_ref(), reg6.get_u(), reg6.get_y());
}

void ext1_interrupt() {
  printf("INTREQ1\n");
  if (ref_fl) {
    reg1.set_ref(dsp2ieee::ieee2dsp_cp_ref(&ref1));
    reg2.set_ref(dsp2ieee::ieee2dsp_cp_ref(&ref2));
    reg3.set_ref(dsp2ieee::ieee2dsp_cp_ref(&ref3));
    reg4.set_ref(dsp2ieee::ieee2dsp_cp_ref(&ref4));
    reg5.set_ref(dsp2ieee::ieee2dsp_cp_ref(&ref5));
    reg6.set_ref(dsp2ieee::ieee2dsp_cp_ref(&ref6));
    ref_fl = 0;
  }
  if (par1_fl) {
    dsp2ieee::ieee2dsp_cp(&par1_v[0],
                          reg1.copy_area,
                          reg1.copy_area_len);
    reg1.new_param();
    par1_fl = 0;
  }
  if (par2_fl) {
    dsp2ieee::ieee2dsp_cp(&par2_v[0],
                          reg2.copy_area,
                          reg2.copy_area_len);
    reg2.new_param();
    par2_fl = 0;
  }
  if (par3_fl) {
    dsp2ieee::ieee2dsp_cp(&par3_v[0],
                          reg3.copy_area,
```

```
                              reg3.copy_area_len);
    reg3.new_param();
    par3_fl = 0;
  }
  if (par4_fl) {
    dsp2ieee::ieee2dsp_cp(&par4_v[0],
                              reg4.copy_area,
                              reg4.copy_area_len);
    reg4.new_param();
    par4_fl = 0;
  }
  if (par5_fl) {
    dsp2ieee::ieee2dsp_cp(&par5_v[0],
                              reg5.copy_area,
                              reg5.copy_area_len);
    reg5.new_param();
    par5_fl = 0;
  }
  if (par6_fl) {
    dsp2ieee::ieee2dsp_cp(&par6_v[0],
                              reg6.copy_area,
                              reg6.copy_area_len);
    reg6.new_param();
    par6_fl = 0;
  }
  intr::DI_ireq1();          //  for simulation
}

void init() {
  sendptr = &send_v[0];

  reg1.set_ref(5.1);
  reg2.set_ref(10.1);
  reg3.set_ref(5.1);
  reg4.set_ref(10.1);
  reg5.set_ref(5.1);
  reg6.set_ref(10.1);

  intr::set_ibf_intr_addr(&ibf_interrupt);
  intr::set_ext1_intr_addr(&ext1_interrupt);
  io::set_ioc(0x480);        // starts i/o clock at simulation
  intr::EI_ibf();
}

main() {
    init();
    LOOP { }
}
```

G. Målsystemets testprogram

```
// To compile: dspCC -o p -m dram.map meminit.o cP.o pvme.c

#include <io.c>
#include <intr.c>
#include <dsp2ieee.c>

#include "meminit.H"
#include "cP.H"

#define LOOP while(1)

float *sendptr;
enum PIF_Mess {
        GENERAL=1,
        READ_0=2,
        WRITE_0=4
  };

Preg reg0(20, -300.0, 300.0);


void host_interrupt() {
  if (ref_fl) {         // New ref
    ref_fl = 0;
    reg0.set_ref(dsp2ieee::ieee2dsp_cp_ref(&ref0));
  }
  if (par0_fl) {        // New parameters
    par0_fl = 0;
    dsp2ieee::ieee2dsp_cp(&par0_v[0],
                          reg0.copy_area,
                          reg0.copy_area_len);
  }
}

void init() {
  sendptr = &send_v[0];
  reg0.set_ref(0);
#ifdef sim
  intr::set_ext2_intr_addr(&host_interrupt);
  intr::EI_ireq2();
#elif vme
  intr::set_ext1_intr_addr(&host_interrupt);
  intr::EI_ireq1();
#endif
}

main() {
    init();

    int loopcnt = 0;
    LOOP {
      io::put_pir(READ_0 | WRITE_0);
      reg0.control();
      if (data_fl==0) {
```

```
        if (sendptr < &send_v[0] + send_size ) {
          dsp2ieee::dsp2ieee_cp(reg0.plot_data,
                                sendptr,
                                reg0.plot_data_len);
          sendptr += reg0.plot_data_len;
        }
        else {
          data_fl = 1;
          io::put_pir(GENERAL);
          sendptr = &send_v[0];  // reset sendptr
}
      }

      loopcnt++;
      if (loopcnt == 20000) {
        loopcnt = 0;
        reg0.set_ref(-reg0.get_ref());
      }
    }
}
```