ISSN 0280–5316
ISRN LUTFD2/TFRT--5472--SE

Objektorienterad implementering för signalprocessor av motorstyrning och givaravläsning

Lars Engelin

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
June 1993

Document Number
ISRN LUTFD2/TFRT--5472--SE

Author(s)
Lars Engelin

Supervisor
Klas Nilsson, Björn Wittenmark

Sponsoring organisation
Swedish National Board for Technical Development (NUTEK), contract 92-04758.

Title and subtitle
Objektorienterad implementering för signalprocessor av motorstyrning och givaravläsning.
Eng: Object oriented implementation for signal processors of motor control and sensor readings.

Abstract

An experimental platform is used for research in robot programming and senor-based motion control. The platform is built around a commercially available robot with AC motors. The robot is controlled from a VME-based embedded control computer, which is also connected via Ethernet with workstations used as host computers. The VME system contains micro processors for general computing and control, but also a board with six floating point digital signals processors (DSPs) for certain demanding control algorithms. Interfaces to the AC motor drive system, and from the resolver measurement system, are directly connected to serial ports on the DSPs. The purpose of this work was to implement the low level software handling the serial interfaces, including the algorithms for AC motor control and sensor handling.

Signal processing and control schemes are naturally expressed in block diagrams. For the implementation, object oriented programming (OOP) was selected as software paradigm. The conclusion is that the blocks (from the block diagram) map well on the software objects from OOP. Furthermore, the inheritance concept of OOP is a convenient way to cope with properties that are in common for different types of motor controls, or for different types of sensor handlers.

The software was implemented in the C++ programming language, and cross compiled for the DSP32C processors. The richness of C++ allowed powerful tailoring of the software for the quite special hardware, and for the demands on efficiency and timing. However, extensive programming experience and great care was needed to really achieve these performances.

Key words
Control systems, Object oriented software, Signal processing, Robot control, Actuators, Sensors, Memory management, Industrial robots

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
Swedish

Number of pages
49

Recipient's notes

Security classification

Innehåll

Förord 2

1. Inledning 3

2. Maskinvarumiljö 5
Experimentsystemets uppbyggnad 5
Signalprocessorn DSP32C 6
Gränssnitt mot robotsystemet 8
Problembeskrivning 9

3. Programvarumiljö 10
Programmering med C++ 10
Utvecklingshjälpmedel 10

4. Objekt orienterat gränssnitt mot snabba ställdon 12
Gränssnitt mot ställdon. 12
Omvandling av momentreferenser till strömreferenser 13
Objektorienterad implementering 14
Hantering av seriekanalen till robotsystemet 14

5. Objekt orienterat gränssnitt mot snabba givare 15
Seriekommunikationen 15
Klassindelning 16
Omvandling från fixtal till vinkelvärde i radianer 16
Hantering av seriekanalen 17

6. Speciella programvaruproblem 18
Hantering av ofullständigheter vid C++ kompileringen 18
Allokering av minne hos DSP32C 18
Problem med avbrotts hanteringen 20

7. Test och igångkörning 22

8. Sammanfattning 24

9. Referenser 25

Appendix: Implementeringar . . . . .

A. Omvandling till strömreferenser 26
actuator 26
ACmotor 28
serialoutport 32

B. Hanteringen av vinkelvärdena 36
sensor 36
resolver 38
serialinport 40

C. Dynamisk minnesallokering 43
malloc och free 43
xmalloc och xfree 48

Förord

Denna rapport beskriver ett examensarbete inom Instutionen för Reglerteknik vid Lunds Tekniska Högskola. Arbetet har bestått i att utveckla, testa och starta upp programvara avsedd för signalprocessorer. Arbetet har till sin karaktär varit mycket likt det som återfinns i den typ av konstruktionsarbete som en civilingenjör från E-linjen ofta bedriver i industrin. Detta innebär framför allt systematisering och utveckling av mjukvara till processorer som är inbyggda i elektronikkonstruktioner.

Jag vill här passa på att tacka min handledare Klas Nilsson för hans stöd, råd och aktiva medverkan i arbetet.

1. Inledning

Detta examensarbete har bestått i att utveckla realtidsprogram för bearbetning av seriell data till och från en industrirobot av typen IRB-2000 från ABB. Informationen från roboten består av vinkelvärden i fixtalsformat från resolvrar, och till roboten består data av strömreferenser för synkronmotorerna på robotens axlar. Eftersom roboten är sexaxlig så finns det sex vinkelvärden och strömreferenser till sex motorer.

Bearbetningen av data sker i (digitala) signalprocessorer (DSP). I denna tillämpning är det valt signalprocessorn DSP32C från AT&T. Signalprocessorer är specialgjorda för att utföra en specifik typ av flyttalsberäkningar väldigt snabbt, nämligen $a = b + c * d$. DSP32C kan med klockfrekvensen 50 MHz utföra 12,5 miljoner sådana beräkningar i sekunden, vilket innebär 25 MFlops.

Samtidigt som det är viktigt att kunna behandla informationen med hög hastighet är det betydelsefullt att ha mjukvaran väl strukturerad. För detta har programmet implementarats objektorienterat. I detta fall har språket C++ valts. Objektorienteringen innebär att data (attribut) och tillhörande procedurer (metoder) samlas i klasser. En klass bör beskriva en väl avgränsad logisk, fysisk eller beräkningsmässig enhet. Ett klassnamn kan ses som ett typnamn, och klassen beskriver objekten som kan ses som de variabler som skapas (instansieras) utifrån klassbeskrivningen. Data bör vara interna i objekten, som då endast kan påverkas utifrån via tillhandahållna (publika) metoder. Språket C++ understöder detta, samtidigt som C:s primitiver för lågnivåprogrammering kan användas.

Robotens vinkelgivare består av resolvrar. Signalerna från dessa omvandlas av RDC-omvandlare (*R*esolver to *D*igital *C*onverter) innan de skickas på en seriell datakanal till en signalprocessor. DSP:n tar emot värdena som kommer i fixtalsformat, och kontrollerar så att de är giltiga. Ett vinkelvärde kan omvandlas till flyttalsformat och radianer av DSP:n. Detta är uppdelat så att mottagningen och kontrollen av vinkelvärdet återfinns inkapslat för sig i ett objekt. Själva omvandlingen till radianer återfinns i de objekt som representerar varje axel, och utförs då någon (t. ex. en regulator) begär värdet.

Informationen till roboten består av styrsignaler till motorerna som driver robotens leder. Motorerna är synkronmotorer, och för varje motor skickas strömreferenser för två av motorns faser, medan den tredje fasen skapas i respektive drivdon. Beräkningen av strömreferenserna baseras på en given momentreferens till motorn och en given kommuteringsvinkel. Kommuteringsvinkeln beräknas ur resolvervinkeln av en DSP. Uppdelningen av mjukvaran i denna delen av projektet är gjord så att det finns sex objekt (ett för varje motor) som beräknar strömreferenser och omvandlar dessa till fixtal från flyttal. Därefter tar ett objekt som administrerar sändningarna ut på seriekanalen över. Detta objekt ser till så att alla sex motorerna får sina nya strömreferens värden så snabbt som möjligt.

Till projektet kom att infogas nya delar eftersom programvaran som användes för utvecklingen var bristfällig. Den ena delen tillkom när kompilatorn inte klarade av att allokera variabler som ska vara gemensamma för alla objekt av en och samma klass, så kallade *static* deklarerade variabler. Genom en extra processning av den fil som C++ kompilatorn lämnade ifrån sig löstes detta. Den andra delen bestod i att biblioteksrutinerna för att dynamiskt allokera och frigöra minne inte fungerade. Detta löstes genom att jag skrev nya rutiner för detta. Dessutom fanns ett problem med att avbrottshanteringen

inte fungerade tillfredsställande. Detta åtgärdades genom att jag ändrade i systemdelen av avbrottsrutinerna.

Denna rapport är uppdelad på följande sätt: Kapitel 2 innehåller en presentation av hårdvaran, samt beskrivningar av hela experimentsystemet, av signalprocessorn DSP32C och av själva snittet mot robotsystemet. Dessutom finns en beskrivning av vilka problem som detta arbete behandlar. Kapitel 3 innehåller en övergripande beskrivning av C++. Även kopplingar till DSP32C tas upp. Dessutom beskrivs de utvecklingshjälpmedel som använts. Kapitel 4 beskriver hur mjukvaran för gränssnittet mot synkronmotorerna är utformad, hur klassindelningen är gjord och de ingående klasserna. Kapitel 5 behandlar mjukvaran för gränssnittet mot vinkelgivarna, och på samma sätt som i kapitel 4 beskrivs klassindelningen och de ingående klasserna. Kapitel 6 beskriver de problem som uppstod med felaktigheterna i programvaran. Dels beskrivs det upptäckta felet med kompilatorn och en lösning presenteras, dels beskrivs problemet med den dynamiska minnesallokeringen och dess lösningen. Även problemen med avbrottshanteringen beskrivs och lösningen visas. Kapitel 7 behandlar utprovning och igångkörning av systemet, och erhållen prestanda och funktionalitet sammanfattas.

Utvecklingen av programmet visar att objektorienterad programmering är ett tilltalande sätt att implementera motorstyrningar och givaravläsningar. Testning i simulator är ett smidigt sätt att få programvaran så felfri att testning i målsystem snabbt blir meningsfull. Det är alltid viktigt att ägna tidskraven för realtidssystem med väldigt höga samplingsfrekvenser stor omtanke. Därför bör man på ett så tidigt stadium av utvecklingsarbetet som möjligt försöka att uppskatta, eller så snart det är möjligt kontrollera att tidskraven är möjliga att uppfylla. Vidare är det viktigt att ha i åtanke att basprogramvara som man använder för utvecklingsarbetet kan vara bristfällig och därmed fördröja projektets genomförande.

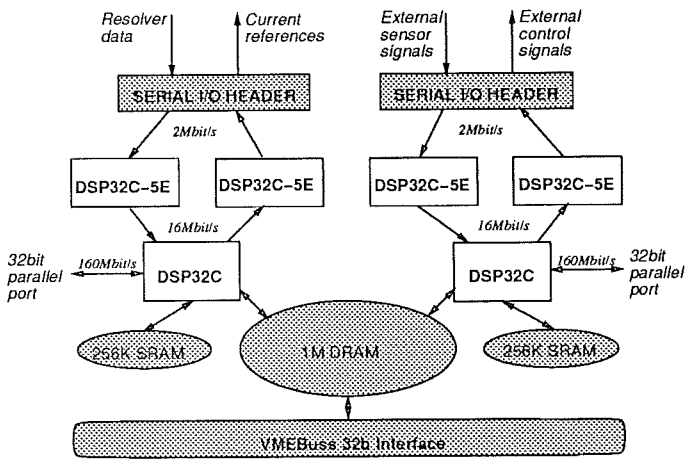
2. Maskinvarumiljö

Detta kapitel beskriver hur systemet där algoritmerna för motorstyrning och givaravläsningen finns är uppbyggt. Dessutom beskrivs hur själva signalprocessorn fungerar, hur gränssnitten mot robotsystemet är konstruerade, och vad som ska göras i systemet. Slutligen ges en problembeskrivning för projektet.

Experimentsystemets uppbyggnad

Experimentsystemet är uppbyggt kring en industrirobot av typ IRB-2000 från ABB. För styrning och reglering av robotens rörelser finns en dator som är uppbyggd kring en VME-buss. Som huvudprocessor används ett datorkort med en Motorola 68040 processor, vidare finns ett VME kort med signalprocessorer från AT&T [9] vilket beskrivs närmare nedan. Ett tredje processorkort med en Motorola 68030 processor hanterar tillsammans med ett digitalt IO kort alla kontrollsignaler till roboten. Detta gör att personsäkerheten återfinns på detta kortet, vars program normalt inte skall ändras. Det finns ett gränssnitt till som är kopplat mellan roboten och signalprocessorkortet. Detta gränssnitt överför informationen seriellt mellan robotsystemet och datorn. Data består då av robotens position (vinklarna på robot axlarna) samt strömreferenser till motorerna, se figur 1.

Den seriella informationen till och från roboten skickas med bithastigheten 2 Mbit/s vilket ger samplingsfrekvensen 62.5 kHz. För att klara av att hantera indata med den hastigheten och att räkna ut utdata med samma höga hastighet behövs en betydande beräkningskapacitet. Dessutom ska den seriella kommunikationen klara en hög överförings hastighet. Vilket ytterligare motiverar användning av *D*igitala *S*ignal *P*rocessorer. Den processor som är vald är AT&T:s 32-bitars signalprocessor DSP32C. Några av anledningarna till att man valde AT&T signalprocessor var att den är relativt enkel i sin struktur.Dessutom fanns ovan nämnda VME-kort med sex st. DSP32C, vilket ger en tilltalande beräkningskapacitet. Två av processorena är försedda med egna externa RAM minnen på vardera 256 kByte. De är dessutom kopplade till ett stort trippelportat minne, som även kan nås direkt från VME bussen. Detta stora minne är på 1 MByte. (Kortet finns också i en variant med 4 MByte.) Som framgår av figur 1 har de 4 processorerna som sitter utåt mot

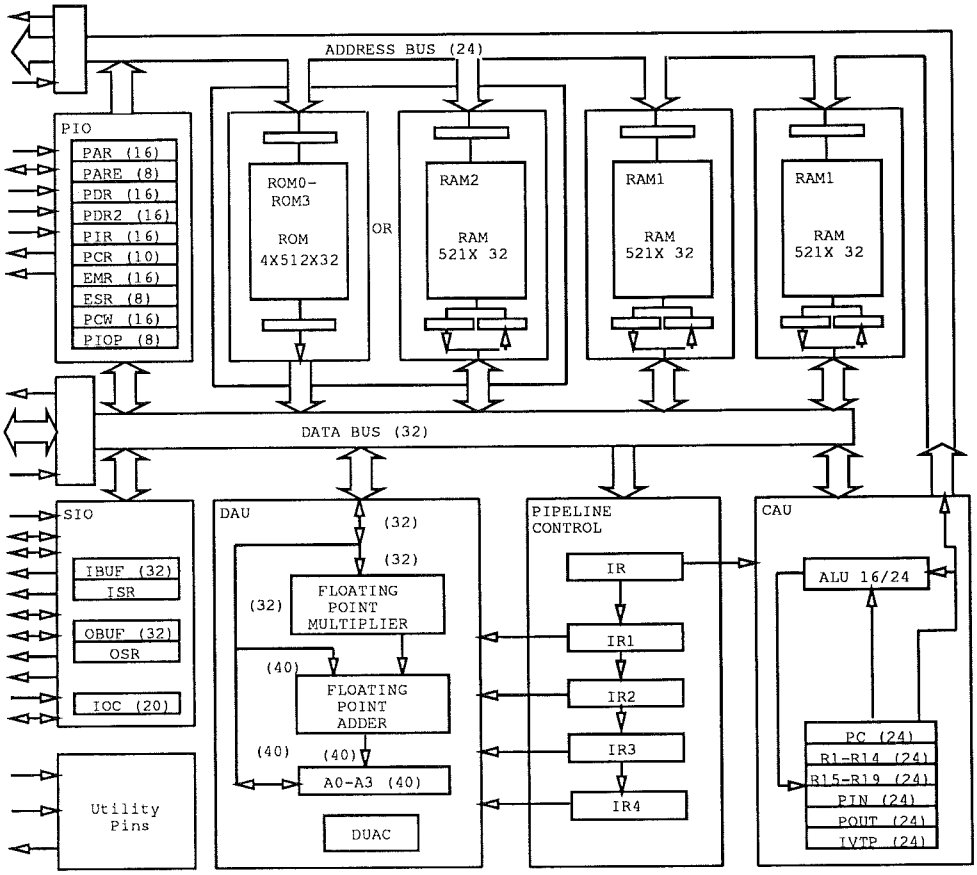


Figur 1. Princip uppbyggnad för VME-kortet med DSP32C processorerna [2]. Dessutom finns ett gränssnitt mellan varje DSP:s portar (PIO i figur 2) och VME-bussen vilket inte visas i figuren.

seriekanalerna (DSP32C-5E) inget externt minne utan får klara sig med det interna RAM minnet på 8 kByte. De 2 RAM minnena på vardera 256 kByte är av SRAM typ (Static RAM) och är lika snabba som det interna minnet i DSP32C, dvs. 0 wait states. Det stora minnet på 1 MByte är av DRAM typ (Dynamic RAM). Adresseringen av DRAM minnet medför upp till 4 wait states för DSP32C.

Signalprocessorn DSP32C

Signalprocessorn DSP32C [8] är en signalprocessor vilket innebär att den optimerats för att utföra successiva flyttalsberäkningar av typen $a = b + c * d$. För att med flyttalsräkning göra detta på ett så snabbt sätt som möjligt har man valt en arkitektur enligt figur 2. Det finns två enheter som utför ex-



Figur 2. Blockschema för DSP32C

ekveringen i processorn *CAU* (control arithmetic unit) och *DAU* (data arithmetic unit). Med hjälp av delen *PIPELINE CONTROL* ,där det finns en fyrstegs pipeline, kan *DAU* och *CAU* arbeta parallellt för att öka beräkningskapaciteten. *CAU* har hand om de logiska operationerna (bit manipulering) och heltalsaritmatiken, kontrollaritmatiken arbetar med 16 eller 24 bitar. *DAU* har hand om alla flyttalsberäkningar och omvandlingar mellan dels olika flyttalsformat och dels mellan DSP:s egna interna flyttalsformat och två komplements 16 eller 24 bitars heltal.

Data transporteras internt med en 32 bitars databuss. Med hjälp av denna databuss kan *PIPELINE CONTROL* enheten under en instruktions cykel (80

ns vid 50 MHz klockfrekvens) utföra fyra minnesaccesser eftersom varje instruktionscykel består av fyra klockcykler (på 20 ns vardera). En instruktionscykel består av:

- Hämtning av en instruktion.
- Hämta två operander från minnet eller en I/O-port (detta tar en klockcykel per operand).
- Skriva till minnet eller en I/O-port.

Den externa databussen är också 32 bitar bred, och det finns möjlighet att definiera wait states (det vill säga man kan få processorn att vänta ett visst antal klockcykler på att det externa minnet ska bli klart) så att det går att använda långsammare minnen än vad klockfrekvensen implicerar. Det går dessutom att definiera olika wait states för olika delar av det externa minnet. Adress bussen är 24 bitar bred, vilket gör det möjligt att adressera upp till 16 MByte. Den externa delen av adressbussen har hjälp av på chipet inbyggd logik för att utan extra kretsar hantera RAM/ROM minnen med 8 bitars databredd. Dessa kan direkt parallellkopplas för att nå 32 bitars databredd.

CAU control arithmetic unit
Denna delen av processorn står för adressberäkningar, kontroll av vilkorliga hopp, 16 och/eller 24 bitars fixtals beräkningar och logiska funktioner. *CAU* består av tre delar:

- 24 bitars ALU som utför fixtalsberäkningarna och de logiska operationerna.
- 24 bitars programräknare (PC).
- Tjugotvå 24 bitars generela register.

CAU har två arbetssätt. Dels görs instruktioner knutna till *CAU*, dvs transport av data, kontroll av hopp, fixtalsberäkning och logiska operationer. Vidare görs adressering av operanderna till *DAU*, vilken kan adressera minnet upp till fyra gånger per instruktion. Adresseringssättet som används vid *DAU* instruktioner är register indirekt adressering (dvs registret pekar ut minnet som ska användas). Om adressen i registret ska ändras (register indirekt adressering med automatisk inkrementering) ökas eller minskas den efter det att minnet har pekats ut. De tjugotvå 24 bitars registren används lite olika beroende på om det är en adress genering till *DAU* eller om det är en 'vanlig' *CAU* instruktion. Till de 'vanliga' *CAU*-instruktionerna kan man använda alla tjugotvå registerna som generella register, men eftersom dom används lite olika när man genererar adresser till *DAU*, och några av registerna används till speciella ändamål, bör man vara försiktig med att använda dem alltför generellt och oövertänkt. När man pekar ut adresser till *DAU* används register 1 till 14 som adresspekare och register 15 till 19 används för att öka eller minska adressen som pekades ut av adresspekaren. Vad det gäller de tre registerna som är kvar används register 22 som basadress till avbrottsvektorn, medans register 19 och 20 används som in- respektive ut-adresspekare när DMA används för seriell data överföring via I/O delen.

DAU data arithmetic unit
Den viktigaste delen av signalprocessorn är *DAU*. Det är den som tillsammans med *PIPELINE CONTROL* utför flyttalsberäkningarna så man kan säga att det är denna delen som är speciell för att DSP32C är en signalprocessor. *DAU* består av följande delar:

- En 32 bitars flyttalsmultiplicerare.

- En 40 bitars flyttalsadderare.
- Fyra 40 bitars ackumulatorer.
- Ett kontrollregister (DUAC).

När *DAU* utför en flyttals beräkning sker det med pipelining av själva instruktionen så när en instruktion som ser ut så här exekveras:

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

Så kommer exekveringen av den instruktionen att ske i fyra steg på följande sätt:

1. Hämta X och Y.
2. Multiplicera X och Y.
3. Addera produkten med ackumulator aM och lägg i ackumulator aN.
4. Skriv till minne eller I/O port.

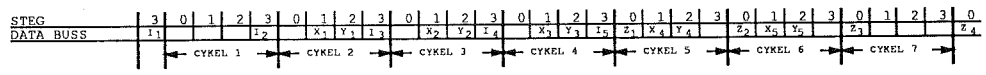
Detta gör att om flera sådana instruktioner ska exekveras i en följd kommer DSP:n automatiskt att lämna ett resultat per instruktionscykel tack vare pipeliningen. Så om följande instruktioner ska exekveras:

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. $Z = aN = aM + X_5 * Y_5$

Så kommer DSP att exekvera instruktionerna i följande ordning:

1)				hämta$_1$ XY
2)			multiplicera$_1$	hämta$_2$ XY
3)		ackumulera$_1$	multiplicera$_2$	hämta$_3$ XY
4)	skriv$_1$	ackumulera$_2$	multiplicera$_3$	hämta$_4$ XY
5)	skriv$_2$	ackumulera$_3$	multiplicera$_4$	hämta$_5$ XY
6)	skriv$_3$	ackumulera$_4$	multiplicera$_5$	
7)	skriv$_4$	ackumulera$_5$		
8)	skriv$_5$			

I figur 3 kan man se hur pipelingen yttrar sig på databussen. I_x står för instruktions hämtning, X_x och Y_x står för hämtning av data och Z_x står för att data sparas i minnet.

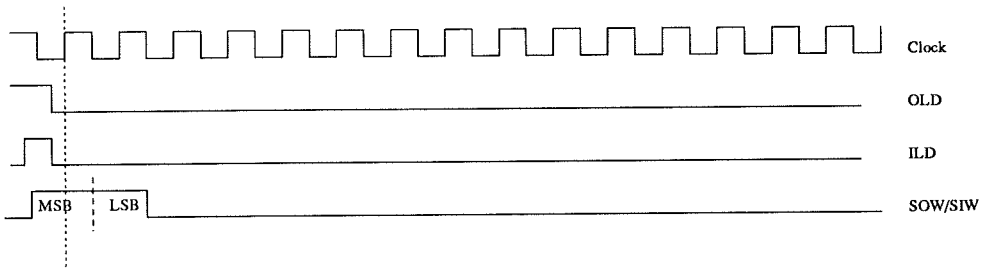


Figur 3. Pipeling på databussen vid *DAU* instruktioner

De flyttals beräkningarna som finns inskränker sig till $+$, $-$ och $*$ men inte / (eftersom division normalt inte ingår i filteralgoritmer), däremot finns en algoritm med hårdvarustöd för att beräkna inverser av flyttal.

Gränssnitt mot robotsystemet

De snitt mot robot systemet som jag har arbetat med är seriekanalerna till styrningen av AC motorerna och för avläsning av vinkelgivarna. Seriekanalerna



Figur 4. Tidsdiagram för seriekanalerna

går direkt mellan robotsystemet och DSP-kortet, där de går direkt till DSP:ernas serieportar. Detta innebär att övriga VME-kort inte berörs av detta stora dataflöde. Seriekanalernas tidsdiagram framgår av figur 4. Signalerna för serieöverföringen in till DSP:n är Clock och för seriekanalen ut från DSP:n OLD och SOW. Seriekanalen in till DSP:n använder ILD och SIW. Signalerna används på följande sätt:

- Clock behövs eftersom det är en synkron seriekanal och den behöver någon klocksignal att arbeta mot. Signalen skapas av det gränssnitt som byggts till robotsystemet.
- SOW (serial output word) är de från DSP:n utgående databitarna. Dessa skiftas ut från DSP:n när Clock byter från låg till hög nivå.
- OLD (output load) markerar genom en förändring från hög till låg att ett nytt ord ska börja sändas från DSP:n med början på nästa bit som skickas. Det är DSP:ns serieport som genererar OLD signalen.
- SIW (serial input word) är de till DSP:n ingående databitarna, DSP:n läser av bitarna när Clock skiftar från låg till hög nivå.
- ILD (input load) markerar genom en förändring från hög till låg att ett nytt ord börjar när nästa bit tas emot av DSP:n. Det är gränssnittet i robotsystemet som genererar ILD signalen utifrån Clock och OLD.

De data som skickas är 32 bitars ord och de kan skickas kant i kant, dvs orden kan följa direkt efter varandra. Signalen Clock har frekvensen är 2 MHz, vilket ger att orden kan skickas med 2/32 MHz = 62.5 kHz. Dessutom är det så att robotsystem både kan ta emot och sända data samtidigt så frekvensen 62.5 kHz gäller för ingående respektive utgående data var för sig.

Problembeskrivning

Målsättningen med detta examens arbetet är att utifrån den hårdvara som finns och med hjälp av programspråket C++, som beskrivs i nästa kapitel, skriva tillämplig programvara. Programvaran ska sköta omvandlingen av data från vinkelgivarna till vinklar (i radianer). Vidare skall det utifrån givna momentreferenser genereras strömreferenser till synkronmotorernas faser i ett sådant format att det kan skickas via den befintliga seriekanalen.

3. Programvarumiljö

I detta kapitel beskrivs kortfattat programmeringsspråket C++ som har använts inom detta examensarbete. Dessutom beskrivs de utvecklings hjälpmedel som har stått till mitt förfogande. Det har i huvudsak varit ett programpaket som heter RTPI, vilket består av en högnivådebugger och en DSP-simulator.

Programmering med C++

C++[1] är ett objektorienterat programspråk vilket innebär stöd för en något annorlunda programmeringsmetodik än med konventionella programspråk. Framför allt understödjs möjligheten att strukturera sitt program utifrån de data (attribut) som ska behandlas, istället för att strukturera utifrån själva funktionen som programmet ska utföra. För att kunna få en objektorienterad struktur på programmet, finns tre huvudegenskaper som utmärker ett sådant språk:

Inkapsling: Genom att samla data (attribut) och operationer (metoder) på dessa data i en typ kallad klass, kan man skapa ett antal objekt (instanser) av den klassen.

Ärvning: Genom att nya klasser kan ärva (ta del av) attribut och metoder från redan befintliga klasser (basklasser) kan man skapa underklasser som blir mer specifika.

Polymorfism: Genom att implementera olika metoder med samma namn i olika underklasser kan man skapa objekt, där rätt sorts metod automatiskt tillämpas, även när anropen ser likadana ut till alla objekt.

Genom att använda mig av C++ möjligheter, framför allt med klasser och ärvning, har jag kunnat göra en implementering som har blivit mer strukturerad än om en lösning i t.ex. C eller Pascal valts. Eftersom C++ bygger på C finns C:s lågnivåprimitiver även i C++, vilket jag har kunnat utnyttja med framför allt bitmanipulering. Detta sammantaget gör att C++ har varit ett för denna uppgiften väl lämpat språk eftersom det är fråga om att omvandla och bearbeta data som ska 'passera igenom' en signalprocessor, se kapitel 2. Detta gör det naturligt att utgå från data (attribut), och i detta fallet kommer även lågnivåprimitiverna väl till pass eftersom det är viktigt att snabbt kunna manipulera vissa bitar i de ord som ska skickas till och från robotsystemet.

Vad det gäller C++ för DSP32C så används inom Institutionen för Reglerteknik AT&T:s C++ kompilator *Cfront* som gör om C++ koden till vanlig C kod. Därefter tar AT&T:s "vanliga" C kompilator för DSP32C *d3cc*[6] över och kompilerar till assembler kod. Assemblering och länkning kan göras med speciella programverktyg, men göres normalt via *d3cc*. *Cfront* är ett generellt program som inte är processorberoende utan kan användas tillsammans med nästan vilken C kompilator som helst. C kompilatorn innehåller dock några utvidningar, såsom möjligheten att infoga assemblerfunktioner med in- och utargument i C koden. I ett tidigare examensarbete[4] har det utvecklats rutiner för att från C++ med ett enkelt anrop kunna nå maskinberoende funktioner, såsom att skriva avbrottsrutiner i C++ eller att kunna nå specifika register eller funktioner såsom seriekanalernas buffertar.

Utvecklingshjälpmedel

Allt utvecklingsarbete från att editera källkoden till testkörning i målsystemet har utförts på en Sun arbetsstation som körts under Unix och X-windows. Dessutom har stationen ingått i ett nätverk även innehållande en filserver.

De utvecklingshjälpmedel som jag har använt är en kombinerad högnivådebugger och simulator som heter RTPI (Real Time Processor Inspector). När simuleringen inte längre har bidragit till programutvecklingen har testkörning i målsystemet utförts. RTPI [3] är en av AT&T utvecklad simulator som kan användas på arbetsstationer och som bland annat kan simulera program avsedda att köras i en DSP32C- signalprocessor, eller en Sun arbetsstation. I RTPI är det möjligt att sätta brytpunkter i C++ kod, i C kod och i assemblerkod. Det går att ha brytpunkter både i C++ och i assembler samtidigt. Det går att läsa och modifiera inte bara hela minnesarean utan också DSP:ns samtliga register. Simulatorn hanterar också symbolisk minnesadressering, det vill säga att det går att använda variabelnamn och även funktionsnamn istället för fysiska adresser. För att göra detta kräver RTPI att kompilatorn och länkaren har producerat en symboltabell. En sådan skapas om man ger direktivet *-g* till både kompilator och länkare. Symboltabellen kan även användas för lokala variabler i funktioner. Fastän dessa bara är allokerade på stacken när funktionen exekveras så fungerar det med symboliskt adresseringen när man simulerar exekvering av funktionen.

RTPI har ett avancerat användargränssnitt med ett fönsterhanteringssystem. Det finns olika fönster för att sätta brytpunkter i källkoden, övervaka globala variabler och möjlighet att temporärt öppna fönster för att övervaka lokala variabler när exekvering sker i någon funktion. Dessutom kan man öppna fönster där man kan få assemblerkod, för att undersöka själva minnet och för att undersöka DSP:ns register.

RTPI gör det möjligt att sätta upp komplicerade testvilkor på ett enkelt sätt, och dessutom är samtidig övervakning av ett stort antal variabler möjligt. Dock är programmet enligt min mening lite tungrott när man ska upprepa en viss test flera gånger i simulatorn med små ändringar i källkoden. Detta beroende på att när man ska simulera en ny programversion så måste man börja om från början med att ladda om programmet i simulatorn, sätta upp alla fönster som man använder, sätta om brytpunkter, markera om variabler som man vill titta på och så vidare. Detta får till följd att när man sysslar med små förändringar av programmet blir simulatorn omständlig att använda.

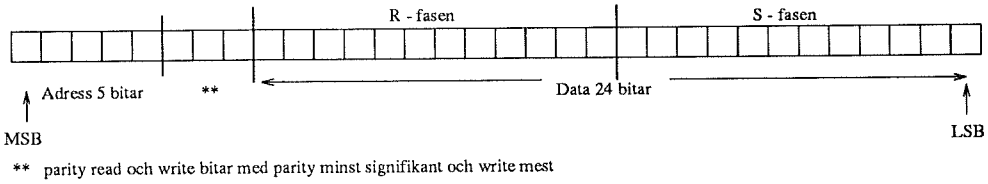
I övrigt är det så med alla simulatorer att realtidskrav såsom att avbrott ska hinna med att exekveras färdigt eller att de ska komma åt att köra utan att behöva vänta, inte helt går att testa med verkliga förutsättningar. Detta gör att det återstår en del arbete med testning när man övergår till målsystemet, detta beskrivs vidare i kapitel 7.

4. Objekt orienterat gränssnitt mot snabba ställdon

Den ena halvan av examensarbetet bestod enligt specifikationen av att från momentreferenser räkna ut strömreferenser till robotens motorer. Uppdateringen av strömreferenserna skall dessutom reagera snabbt på ändringar av motorvinkeln. En beskrivning av hur omvandlingen går till ges i detta kapitel.

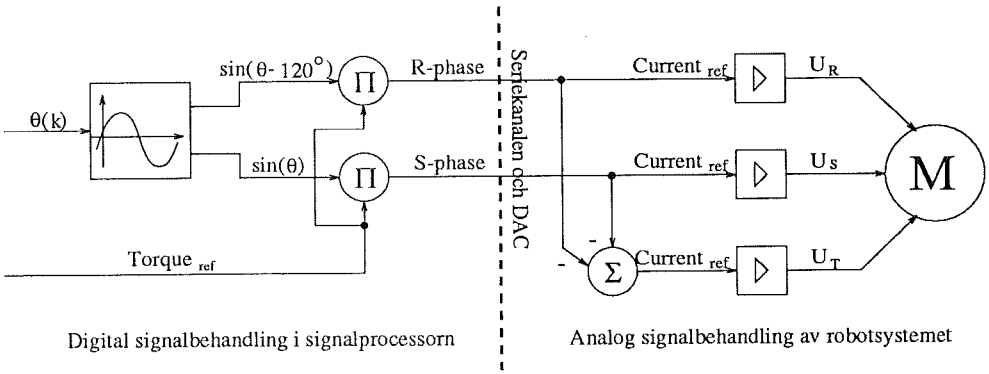
Gränssnitt mot ställdon.

Motorerna är fyrpoliga trefas synkronmotorer. Drivelektroniken innehåller en strömregulator för varje fas. Eftersom summan av strömmarna i de tre faserna alltid är noll, så behövs det bara skickas referenser för två av faserna. Detta gör att det som ska skickas på seriekanalen är två stycken strömreferenser tillsammans med en adress som talar om vilken motor det är. Som visades i kapitel 2 så är det ett 32 bitars ord som skickas på seriekanalen. Varje ord innehåller både adress och strömreferenser och är samansatt enligt figur 5. De



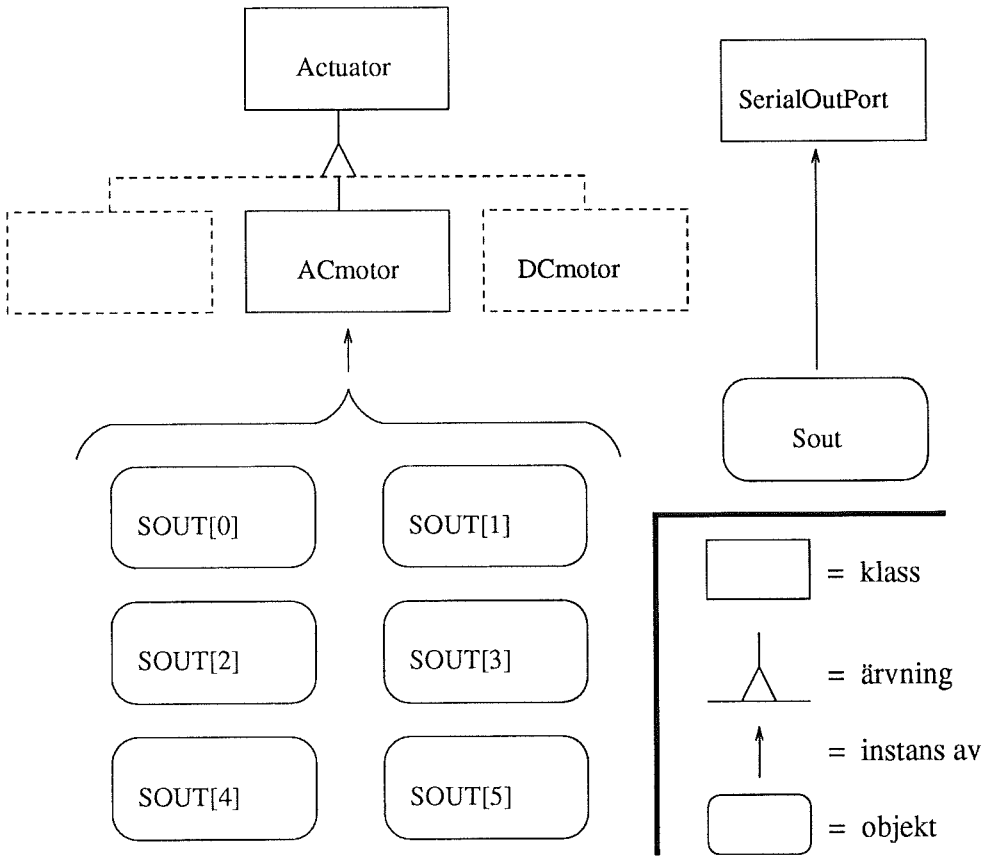
Figur 5. Formatet hos det inkommande 32 bitars ordet på seriekanalen

fem mest signifikanta bitarna anger adressen till motorn/axeln. Sedan följer tre kontrollbitar för att styra skrivning och läsning till och från robotsystemet. De 24 bitar som är kvar används till strömreferenserna, vilket ger 12 bitar per fas. Upplösningen blir då $1/4096 \approx 2.4 * 10^{-4}$ (från -2048 till 2047). Den digitala delen av motorstyrningen enligt figur 6 är uppdelad i två delar. Beräkningarna av strömreferenserna är en del. Den andra är hanteringen av snittet mellan den analoga och den digitala delen av signalhanteringen,vilket innehåller seriekanalen mellan robotsystemet och DSP:n. Eftersom skrivningen till seriekanalen lämpligen utförs i ett avbrott är uppdelning dessutom gjord som en 'Foreground/Background' princip. Beräkningarna av strömreferenser återfinns i en klass (basklassen `actuator` och underklassen `ACmotor`). I bakgrundsprocessen kommer det att finnas ett objekt till varje motor. Klassen `SerialOutPort` innehåller avbrottsrutinen som skickar data till roboten. I



Figur 6. Principbild för omvandlingen från momentreferenser till huvudspänningar för AC-motorn.

avbrottsrutinen säkerställs det att något alltid skickas. Det finns bara ett objekt av `SerialOutPort`. Klass indelningen och objekt uppdelningen framgår av figur 7. Implementeringen återfinns i kapitel A.



Figur 7. Princip bild för klass indelningen och objekt uppdelning av utdata hanteringen, samt förklaring av symboler.

Omvandling av momentreferenser till strömreferenser

Själva omvandlingen från momentreferens till strömreferenser består av en relativt okomplicerad algoritm [5]. I princip använder man sig av motorns kommuteringsvinkel, vilken är känd via en resolvervinkel, se kapitel 5. Kommuniteringsvinkeln multipliceras med antalet poler. Sinus av den nya vinkeln multiplicerat med momentreferensen ger strömreferensen för den ena fasen. För den andra fasen subtraheras 120 grader innan sinus beräknas och man multiplicerar med momentet. Utöver denna grundprincip innehåller algoritmen kompenseringar och offset justeringar vilket ger den följande utseende:

- Korrigera för avvikelse mellan kommuniteringsvinkeln och vinkeln från vinkelgivaren, lägg till eventuell annan kompensering (fördröjningar eller liknande).
- Multiplicera med antalet poler för motorn.
- Beräkna sinus för den nya vinkeln och sinus för vinkeln - 120 grader.
- Multiplicera de två värdena med momentreferensen.
- Addera till eventuella offset strömmar till respektive fas.

- Trunkera resultaten så att de får plats inom ett 12 bitars ord (-2048 – 2047).
- Maska ihop strömreferenserna till ett 24 bitars ord.

Sinus beräkningarna som görs i beräkningarna av strömreferenserna är en serieutveckling av $\sin x$ nämligen $\sin x = x + c_1x^3 + c_2x^5 + c_3x^7 + c_4x^9$ se [7]. AT&T anger onoggranheten till $\leq 10^{-7}$, förutsatt att vinkeln är mellan $\pm \pi/2$. Hela beräkningen av strömreferenser kan ses som en speciell formattering av utdata. Algoritm är därför inplementerad som en operator (<< operatorn i C++), vilket anknyter till hur in- och utmatning normalt sker i C++program. Det ger användaren intrycket av att bara 'skicka iväg' en momentreferens till motorn utan att se/märka den underliggande hanteringen av momentreferensen.

Objektorienterad implementering

Dataomvandlingen sker inom en basklass `actuator` och en tillämpningsspecifik underklass `ACmotor`. Basklassen `actuator` innehåller strömreferenserna lagrade i ett 32 bitars ord, och en metod för att föra strömreferenserna vidare till objektet som hanterar själva sändningen av data till robotsystemet. Underklassen `ACmotor` innehåller den för denna motortyp specifika metoden för att göra omvandling från moment till strömreferenser enligt figur 6. Den objektorienterade uppbyggnaden gör det enkelt att skapa nya underklasser för andra typer av motorer. Basklassen är då inte inblandad i omvandlingen, utan tillhandahåller generiska metoder som gäller för alla typer av ställdon.

Hantering av seriekanalen till robotsystemet

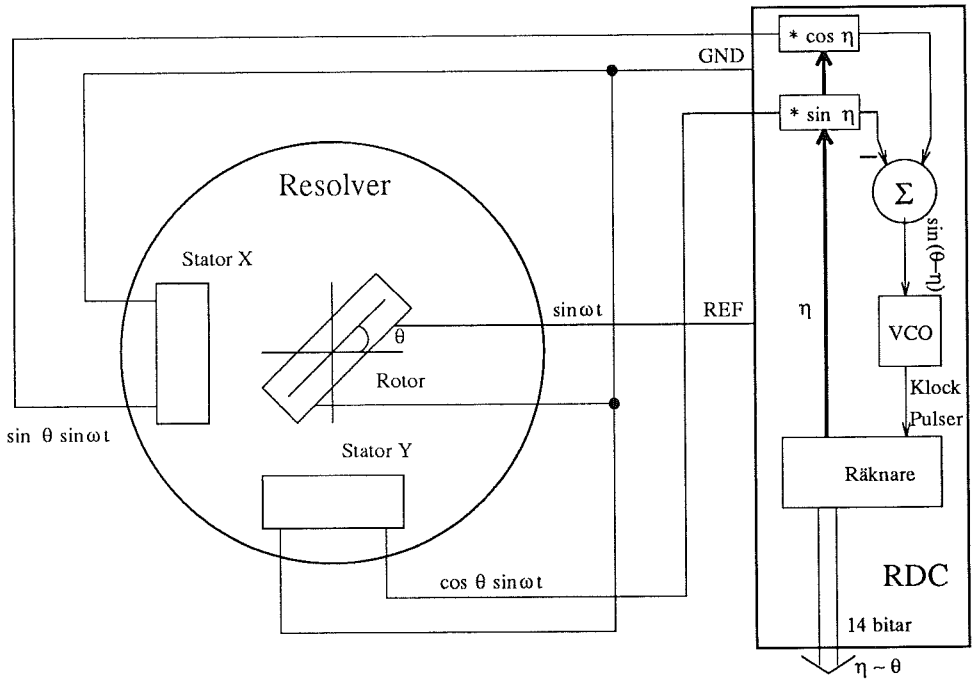
Skickningen av strömreferenserna via seriekanalen hanteras av en egen klass `SerialOutPort`. I denna klassen finns avbrottsrutinen som anropas då den utgående buffferten är tom (dvs ett nytt ord kan skickas). Det är i detta avbrott som distribueringen av data, dvs strömreferenserna, sker. Här tas bara hänsyn till hur de 8 mest signifikanta bitarna i 32 bitars ordet ser ut, dvs adress och kontroll bitarna. Adressen ska vara giltig (motsvarande motor ska finnas) och både write och read bitarna sätts innan sändningen. De 24 minst signifikanta bitarna bryr sig denna klass inte om. När ett avbrott kommer så ser man till så att det mest angelägna värdet skickas. Detta sker efter principen först in först ut, dvs om det har kommit mer än en uppsättning värden när ett avbrott exekveras skickas det äldsta. Om det har kommit mer än en begäran att skicka värden till en adress så skickas endast det nyaste värdet till den adressen. När det inte finns några nya värden att sända så upprepas sändningen av de senaste värdena i tur och ordning till de adresser som finns. Klassen innehåller ett avbrott som bara kan finnas i en upplaga då avbrottshanteringen inte kan skilja på olika objekt. Avbrottet är static deklarerat vilket gör att det kodas som en fristående rutin, som avbrottshanteringen hittar utan problem.

5. Objekt orienterat gränssnitt mot snabba givare

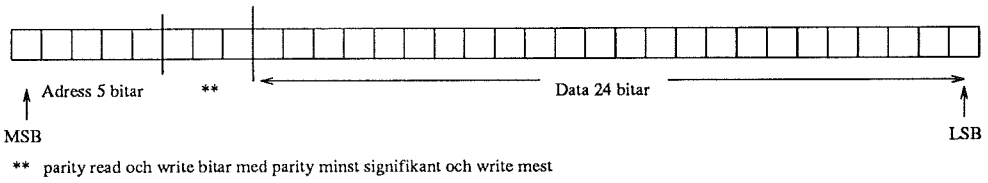
Robotens motorvinklar mäts med resolvrar. En resolver fungerar i princip enligt figur 8. Rotorn matas med en signal $\sin\omega t$, och de två statorlindningarna kommer att skicka tillbaka signalerna som $\sin\theta\sin\omega t$ respektive $\cos\theta\sin\omega t$. Trigonometrin ges av figuren där θ är den sökta vinkeln. Signalerna från statorlindningarna är dämpade jämfört med signalen till rotorn, men detta kompenserar RDC-kretsen för. Signalerna matas till RDC:n som producerar en digital utsignal η, svarande mot resolverns vridningsvinkel θ. Detta sker på följande sätt: Referenssignalen $\sin\omega t$ moduleras bort och η återkopplas så att signalen $\sin\theta\cos\eta - \cos\theta\sin\eta = \sin(\theta-\eta)$ skapas. Då blir $\sin(\theta-\eta)$ ett mått på felet mellan θ och η (för små x är $\sin x = x$). Felsignalen skickas till en "Voltage Controlled Oscillator" (VCO), vilken ger klockpulser till en räknare. Utsignalen från räknaren är η i digital form.

Seriekommunikationen

Vinkelvärdena från robotens resolvrar överförs via seriekanalen från robotsystemet till DSP32C. Formatet är ett 32 bitars ord vilket har ett utseende enligt figur 9. De 5 mest signifikanta bitarna innehåller adressen, dvs från vilken robotaxel som vinkeln kommer ifrån. De tre bitarna därefter är read, write och parity bitarna. Vinkel är relevant om *read*biten är satt, annars är värdet odefinierat. de första 8 bitarna har därmed samma innebörd som för motorstyrningen enligt föregående kapitel. De första 7 bitarna ges dessutom samma värden av den adresserade enheten. *Write*biten används således främst när man skickar data till robotsystemet, men kan vara till nytta om man vill läsa av givarna oftare än man reglerar. Paritetskontrollen har ännu inte implementerats, varken i maskinvaran eller i programvaran. De 24 minst signifikanta bitarna är de som innehåller själva vinkelvärdet. De 16 minst sig-



Figur 8. Princip bild för en resolver med RDC *R*esolver to *D*igital *C*onverter. Värdet på den digitala vinkeln η kommer att ställa in sig på värdet av den fysiska vinkeln θ. Snabbheten bestäms av bandbredden på den faslåsta slingan i RDC:n.



Figur 9. Formatet hos 32 bitars ordet på seriekanalen för givaravläsning.

nifikanta bitarna anger motorvinkeln och de övriga 8 är en varvräknare. Detta medför, beroende på att RDC:n för närvarande inte använder de två minst signifikanta bitarna, att vinkeln får en upplösning på $2^{-14} = 1/16384 \approx 0.022°$ och att varvräknaren kommer att vara mellan -128 till 127. RDC-kretsen understödjer omkoppling till 16 bitar/varv vid låga varvtal. Då kan även de två sista bitarna användas, men elektroniken understöder för närvarande inte detta. När vinkelvärdet omvandlas till flyttalsformat kommer hela mantissan i DSP:ns flyttalsrepresentation att innehålla signifikant data, eftersom vinkeln är ett fixtal på 24 bitar och mantissan också innehåller 24 bitar. Detta bör man ha i åtanke så att man inte förlorar information när man behandlar vinkelvärdena.

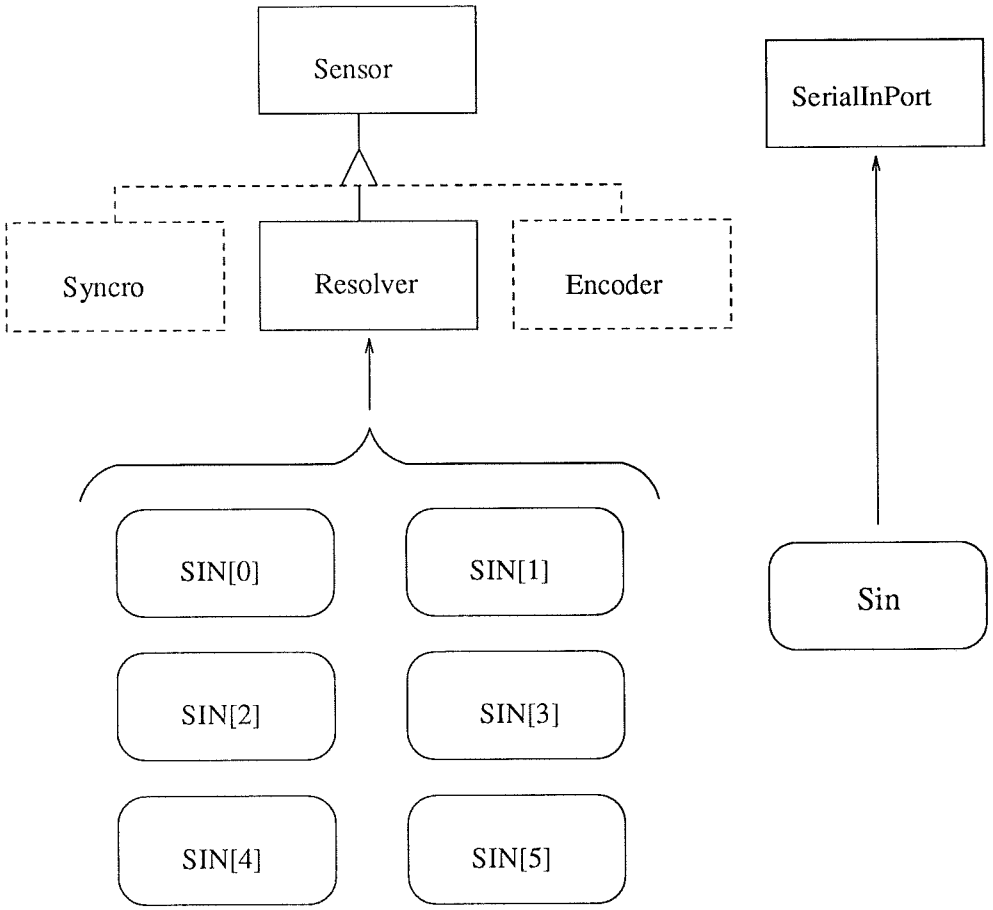
Klassindelning

Hanteringen av resolverdata är uppdelad i två delar. En klass `SerialInPort` som hanterar serieporten genom att den läser av inkommande data. Den ser också till så att bara relevanta data skickas vidare in i systemet, dvs det krävs att robotsystemet har satt *read* biten. Det finns också en basklass `Sensor` som tillsammans med en underklass `Resolver` står för omvandlingen till flyttalsformat och radianer av fixtalet som kommer via seriekanalen, se figur 10. Implementeringen återfinns i kapitel B. `SerialInPort` skickar giltiga vinkelvärden vidare till rätt objekt av `Sensor/Resolver` klassen. Till varje robotaxel finns ett objekt av `Sensor/Resolver`-typ dvs i fallet med IRB-2000 finns sex `Resolver`objekt i programmet eftersom det finns sex resolvrar på roboten. Finns inget motsvarande objekt för ett inkommande värde, så går värdet förlorat.

Omvandling från fixtal till vinkelvärde i radianer

Klassen `Resolver/Sensor` har hand om själva omvandlingen från fixtal till flyttal och radianer. Varje objekt av klassen lagrar det senaste värdet från sin givare i fixtalsformat, detta återfinns i basklassen `Sensor`. Själva omvandlingen till radianer är beroende av givartyp och är därför implementerad i underklassen `Resolver`. Detta gör att om resolvrarna skulle bytas ut mot någon annan typ av vinkelgivare, så är det bara att skapa en ny underklass till `Sensor` med den nya omvandlingen. Förutom att det nya objektet måste skapas eller deklareras behöver programmet för övrigt inte ändras. Omvandlingen till radianer är implementerad i operatorn >>, precis som inläsning från fil normalt utförs från C++ program. Detta ger användaren av objekten intrycket av att bara hämta ett vinkelvärde, utan att känna till hur omvandlingen går till eller för den delen inte heller i vilken form som värdena kommer till DSP:n. Själva omvandlingen till radianer är en tämligen enkel operation och ser i princip ut så här:

- Omvandla 24 bitars fixtalet till flyttal (en instruktion med DSP:ns instruktionsuppsättning).



Figur 10. Princip bild för klass indelningen av indata hanteringen

- Utnyttja att ett halvt varv = π radianer = $2^{16}/2 = 32768$ vilket gör att vinkelvärdet ska multipliceras med $\pi/32678 = 9.587379 * 10^{-5}$.

Hantering av seriekanalen

SerialInPort klassen innehåller en avbrottsrutin (dvs en "static member function" som kopplats till ett avbrott) som körs när det kommer något på seriekanalen. Om vinkelvärdet kommer från någon av vinkelgivarna, dvs adressen är giltig och om *read*biten är satt kommer vinkelvärdet att skickas till rätt `Sensor`objekt. Dessutom registeras avbrottet i objektet av SerialInPortklassen. Detta används för att tillhandahålla en tidbas för andra delar av programmet. Detta är möjligt eftersom avbrotten kommer med en konstant frekvens (62.5 kHz = 16 μs). T. ex. kan regulatorer och andra rutiner som ligger utanför avbrotten fås att exekvera en gång per avbrott från den ingående seriekanalen. På samma sätt som i kapitel 4 kommer avbrottet att arbeta i förgrunden, medan objekten `Resolver/Sensor` som tillhandahåller omvandlingarna till radianer är tänkta att användas av en bakgrundsprocess.

6. Speciella programvaruproblem

Examensarbetet kom att utvidgas med tre problem med programvaran som jag använde. Det första bestod i att C++ kompilatorn inte hanterade static direktivet för variabler på ett sätt som passade DSP:ns C-kompilator. Det andra bestod i att de medföljande biblioteksrutinerna för dynamiskt allokering och frigörande av minne inte fungerade. Slutligen så fanns det ofullständigheter i avbrottshanteringen med framför allt fel i stackmanipuleringen.

Hantering av ofullständigheter vid C++ kompileringen

Problemet uppstod när man i C++ koden deklarerade variabler som 'static'. Det gör man när variablerna ingående i en klass ska vara gemensamma för alla objekt av klassen. Problemet påträffades vid deklarationen av avbrottsrutiner. Dessa måste deklareras som static för att slippa argumentet *this* (se nedan) till den genererade C-funktionen, vilket inte skulle fungera eftersom anropet sker direkt från hårdvaran.
Till alla objekt av en klass finns en pekare som heter *this* och som pekar på objektet. Denna pekare används för att nå variabler och funktioner ingående i objektet. Dock finns två undantag, static deklarerade variabler finns endast i en uppsättning så att *this* pekaren pekar inte ut dessa variabler. Det behövs ju inte då variablerna kan adresseras direkt. Det andra undantaget är att static deklarerade funktioner inte kan nå *this*pekaren, eftersom de inte tillhör något specifikt objekt utan är gemensamma för alla objekt. Detta gör att static deklarerade funktioner bara kan nå static deklarerade variabler.

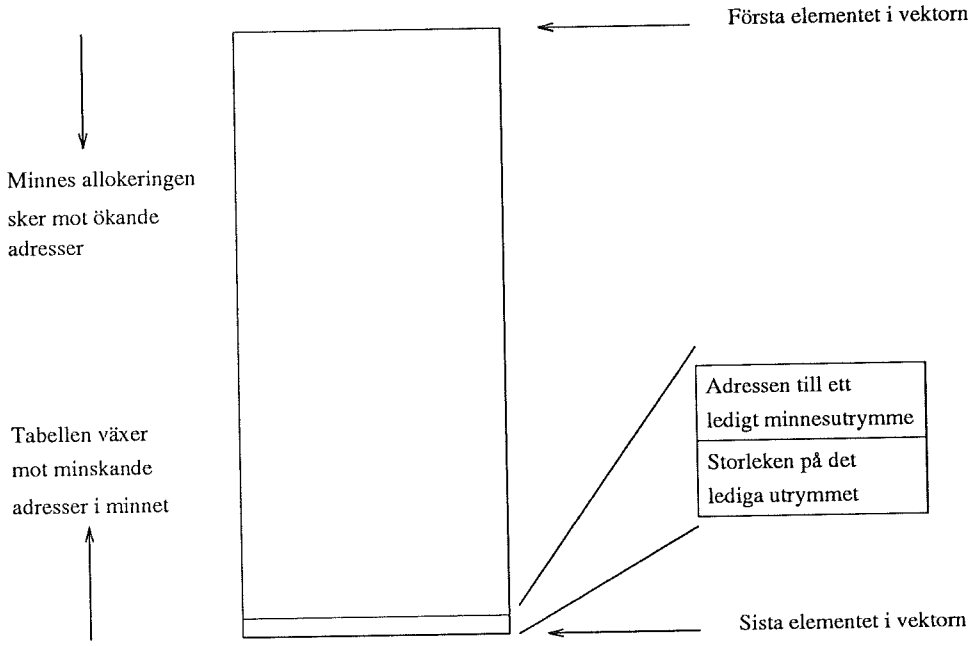
C++ kompilatorn som används är från AT&T och den är sådan att den av C++ koden gör 'vanlig' C kod som den sedan överlämnar till en C-kompilator och dess länkare. Problemet består i att DSP:ns länkare inte klarar av variabler deklarerade mer än en gång, dvs typ-deklarationen och extern deklarationen måste göras samtidigt. Den generade C-koden blir inte sådan trots att jag deklarerar och initierar alla static variablerna en gång utanför klassen. Anledningen är att deklarationerna av klasserna som innehåller de static deklarerade variablerna finns i så kallade headerfiler. Dessa inkluderas i flera olika filer med C++ kod som alla kompileras var för sig. Om kompilatorn inte hittar någon allokering och initiering av en static deklarerad variabel så lägger den till en deklaration sist i C filen som skapas. Länkaren kommer då att hitta en deklaration i varje fil som har med den ursprungliga klassen att göra. Detta ska vara tillåtet, och accepteras t. ex. av C-kompilatorn för Sun, men inte av den för DSP32C. Lösningen är att förbehandla filen (med hjälp av ett awk-script) som skapas av C++ kompilatorn och ta bort alla deklarationer av static deklarerade variabler som inte är extern deklarerade eller innehåller initieringen av variablen. Detta gör att endast deklarationen med initieringen är kvar vilket accepteras av länkaren.

Allokering av minne hos DSP32C

Den andra svårigheten var att de medföljande biblioteksrutinerna för att dynamiskt allokera och frigöra minne inte fungerade. Detta medförde att operatorn *new* i C++ inte fungerade och då troligen inte operatorn *delete* heller. Min handledare och jag försökte att skaffa oss en uppfattning om hur biblioteksrutinerna fungerade, och vilka variabler som de använde. Eftersom vi inte hade tillgång till källkoden utan var hänvisade till den assemblerkod som debuggern RTPI (se kapitel 3) kunde disassemblera åt oss, gav vi upp försöken att den vägen åstadkomma något. Istället löstes problemet genom att skriva helt

nya C-rutiner malloc och free. Dessutom utvecklades ytterligare två rutiner nämligen xmalloc och xfree mer om dessa senare. Implementeringen återfinns i kapitel C.

Allokeringen (malloc) och frigörningen av minne (free) fungerar på följande sätt: När malloc och free rutinerna länkas med kommer ett relativt stort minnessegment att allokeras statiskt som en stor vektor, i nuläget är det en vektor på 64 Kbyte. Detta är avpassat i någon mån efter SRAM-minnena, se kapitel 2. Denna vektor används som minnesutrymme för allokeringen i malloc rutinen. Det allokeringsbara utrymmet delas av en tabell som håller reda på ej allokerat minnesutrymme. Tabellen visar var det finns minne och hur många bytes i följd som kan allokeras, se figur 11. När malloc rutinen får



Figur 11. Disponering av dynamiskt minne

en begäran att allokera ett visst antal bytes (inparameter till rutinen), så letar den i tabellen efter den första lediga platsen som är tillräckligt stor, reserverar så mycket plats som behövs, och ger adressen till minnesarean som returvärde. Om det blir plats över, dvs om den lediga platsen är större än den minnesarea som allokeras, så kommer det resterande utrymmet att markeras som ledigt i tabellen. Detta förfarande leder, åtminstonde på stora datorsystem, till en kraftig fragmentering av minnet. Detta bedöms dock inte som ett problem i denna tillämpning där de flesta variabler allokeras när programmet initieras. Detta gör att komplicerade algoritmer som tar stor plats och arbetar med att få bästa möjliga utnyttjande av minnet vid varje ny allokering inte är speciellt meningsfulla att tillgripa för signalprocessorer utan det är bättre att göra rutinerna korta och rubusta i stället. Vad det gäller rutinen för frigörande av minne free(adress) så gör den inget annat än markerar den återigen lediga minnesarean i tabellen över ledigt minne. Dessutom ser den efter om minnet närmast före och/eller efter är ledigt. Detta för att skapa ett så stort ledigt område som möjligt och för att undvika onödigt stor fragmentering av minnet.

De två tidigare nämnda rutinerna xmalloc och xfree används för att styra allokeringen av minne till det gemensamma DRAM minnet på 1 Mbyte, se kapitel 2. Detta är av intresse för objekt (t. ex. regulatorparametrar) som au-

tomatiskt skall kunna nås från VME-bussen. När xmalloc och xfree länkas med allokeras statiskt en stor vektor på 64 Kbyte eller mer i DRAM arean. Xmalloc och xfree talar om för malloc och free att de ska arbeta med minnesarean (vektorn) i DRAM minnet, innan de anropar dem. Genom att kombinera xmalloc och xfree med möjligheten att i C++ omdefiniera operatorerna *new* och *delete* för klasser kan man på ett smidigt sätt själv styra i vilken del av minnet som variablerna ingående i en specifik klass ska allokeras. Här bör nämnas att C++ än så länge inte klarar att omdefiniera operatorerna så att en vektor med objekt kan allokeras av den egna operatorn. Förslag på ny standard är att man även ska kunna definiera operatorerna new[] och delete[] för att hantera även detta fall.

Problem med avbrotts hanteringen

För att kunna skriva avbrottsrutiner i C++ finns det inplementerat rutiner skrivna i assembler [4]. Dessa rutiner stackar de register som enligt kompilatorn används som temporära variabler. Sådana variabler används t. ex. för lagring av mellanresultat i beräkningar, och för lagring av returvärden från funktionsanrop. Eftersom kompilatorn inte kan ta hänsyn till att avbrott kan komma in och ändra på värdet i registren, måste man i avbrottshanteringen se till att dessa register återfår sina ursprungliga värden efter ett avbrott. Problemen som uppstod var relaterade till stackningen av registren. Problemet uppstod för att avbrottsrutinerna förutsatte att stackpekaren alltid inkrementerades efter att något stoppats på stacken. Det såg länge ut som om kompilatorn alltid inkrementerade, t. ex. så ökades värdet alltid vid stackningar i samband med funktionsanrop. Men i ett fall som jag fick problem med så användes stacken för att lagra mellanresultat utan att stackpekaren ökades. Kompilatorn genererade kod som såg ut i princip så här:

```
*spe = a0 = int24(*r1) /*omvandla från flyttal till fixtal */
...
...
r1e = *sp
```

Detta fick till följd att när avbrottet kom mellan omvandlingen och lagringen i r1 så skrevs avbrottet över det som stackpekaren pekade ut (sp*). Lösningen blev att alltid öka stackpekaren i avbrottsrutinen innan den användes.

Dessutom innehåller biblioteksrutinerna (t. ex. sinus) i applikationsbiblioteket ett fel som ser ut så här:

```
sp -= 8
r1e = *sp++
r2e = *sp++
sp -= 8
return
```

När avbrott kommer in emellan det första och det sista `sp -= 8` kommer abrottet att skriva över de värden som r1 och r2 skulle ha tilldelats. Lösningen här blev att i avbrottsrutinen ytterligare öka stackpekaren så mycket att man säkert skrev på ledigt minnesutrymme.

Ett annat problem (ej förosakat av AT&T) med assemblerrutinerna för avbrottshanteringen var att rutinen avslutades så här:

```
r1 = *sp--
ireturn
```

Felet är att instruktionen `ireturn` under en cykel återställer hela pipelinen (tillsammans med pc och flyttalsregisterna a0 - a3). Eftersom instruktionen `r1`

= `*sp--` behöver mer än en instruktionscykel på sig för att föra över värdet från stacken till r1 registret, kommer `ireturn` instruktionen att förstöra värdet innan det är överfört. Problem av denna karaktären är enkla att lösa men svåra att hitta. I detta fall var lösningen att lägga in en `nop` instruktion före `ireturn`.

7. Test och igångkörning

Det utvecklade programmet ingående delar kan man huvudsakligen dela in i följande tre delar.

- Hantering av indata.
- Hantering av utdata.
- Hantering av minnesallokering.

De är alla uttestade var för sig i DSP-simulatorn via debuggern RTPI se kapitel 3. De är dessutom uttestade tillsammans som en programmodul i RTPI. Dock är det så att simulatorn inte kan hantera avbrott så att själva realtids hanteringen såsom t. ex. ömsesidig uteslutning och tidskrav kan inte testas i simulatorn. Avbrotts rutinerna är testade genom att det läggs in anrop till dem i bakgrundsprocessen. Detta medför att en väsentlig del av utprovningen och testningen av programmodulen måste ske i målsystemet, som beskrivits i kapitel 2. Övervakning över testkörningen fås via ett datorkort med en Motorola 68040 processor på, varifrån man kan läsa av minnet på DSP-kortet via VME bussen. En första körning i målsystemet visade på att det fanns lite problem med den ömsesidiga uteslutningen och även med tidskraven. DSP:n hade stora problem med att hinna med att behandla avbrotten och följdaktligen även med att exekvera den kod som finns i bakgrundsprocessen.

Vad det gäller den ömsesidiga uteslutningen så är all kod genomsökt med lupp efter alla tänkbara (och otänkbara) ställen där bakgrundsprocessen hanterar gemensamma data. I avbrotten kan det inte uppstå problem med gemensamma data eftersom DSP32C är sådan att när den exekverar ett avbrott så är avbrotten avstängda tills man återvänder från avbrottet. Det går således inte att bli avbruten i ett avbrott, vilket garanterar att hela avbrottet blir en odelbar operation. Ett par eventuella problemställen identifierades och modifierades så att den ömsesidiga uteslutningen alltid garanteras. Provkörning visar att programmet fungerar bra.

Vad det gäller tidskraven så är hastigheten på seriekanalen 2 Mbit/s, vilket gör att hela 32 bitars ord skickas med 62.5 kHz. Avbrotten kommer alltså med 16 μs intervall. Eftersom det är både indata och utdata på varsin kanal så kommmer avbrotten med i genomsnitt 8 μs intervall. En instruktion i *CAU*delen av processorn tar 80 ns det gör att man bara har tid med 100 instruktioner i genomsnitt mellan varje avbrott. *DAU*delen kan man få att gå med dubbla hastigheten men andelen flyttals beräkningar är förhållandevis liten i denna tillämpningen, så det påverkar inte nämvärt resonomanget ovan. 100 instruktioner är förstås alldeles för lite, speciellt om man betänker att det är mer än avbrotten som ska exekveras. Genom att DSP-kortet begär att roboten ska skicka informationen, blir det DSP-kortet som avgör överföringshastigheten. När det skickas nya värden till motorerna på roboten sätts samtidigt *read*biten, se kapitel 4, vilket får robotsystemet att omedelbart skicka tillbaks vinkelvärdet för samma axel till DSP kortet. När DSP-kortet får ett nytt vinkelvärde så ska det göras databearbetning innan det kan skicka ett nytt värde tillbaka till robotsystemet. Detta gör att robotsystemet väntar på DSP-kortet. Programvaran är nu i sådant skick att DSP-kortet förmår att skicka nya värden med 100 – 120 μs intervall. Det motsvarar att avbrotten och bakgrundsprocessen utför ca 1250 – 1500 instruktioner mellan varje nytt värde till robotsystemet.

Omvandlingen av vinkelvärdena till radianer och omvandlingen från moment till strömreferenser fungerar som det ska. Den önskvärda förbättringar av

programvaran ligger i att öka snabbheten, så att det blir möjligt att klara tidskravet 16 μs mellan sändning respektive mottagning av data på seriekanalen. För att nå dit behövdes en 6 - 8 gångers ökning av snabbheten.

Ett efterföljande projekt med syfte att snabba upp systemet genomfördes direkt efter examensarbetets slutförande. Genom att utnyttja att DSP-kortet innehåller fler DSP32C processorer, kan man fördela beräkningarna på mer än en processor. På DSP-kortet sitter det på serie kanalerna till respektive från roboten en extra DSP32C processor på varje kanal. Dessa processorer har använts för att utföra en del av beräkningarna, t. ex. sinus beräkningarna i strömreferensberäkningarna. Dessutom överförs momentreferenserna och kommuteringsvinkeln till utprocessorn med hjälp av DMA mellan serieportarna. Därmed halveras avbrottsfrekvensen. Detta innebär att reglering av robotens samtliga sex leder kan göras med samplingsfrekvensen 8.333 kHz vilket får betraktas som mycket bra. Speciellt snabb sampling kan vara av intresse vid test av envariabla servoalgoritmer. Genom att då endast reglera en led, så kan regleralgoritmer som skrivits i C++ exekveras med den imponerande samplingsfrekvensen 50 kHz.

8. Sammanfattning

Detta arbete har ingått i utvecklingen av en experimentmiljö för forskning inom reglerteknik och robotik. I denna miljö ingör en i industrin vanligt förekommande industrirobot av typen ABB IRB-2000. Robotens ordinarie datorkort har ersatts med ett fristående datorsystem i vilket nya tekniska lösningar enkelt skall kunna provas. Detta innebär att avläsning av robotens inbyggda givare, och styrning av robotens växelströmsmotorer, måste implementeras i det egna datorsystemet. Avsikten med detta arbete var att utföra denna implementering, med beaktande av speciella krav på programmets struktur, flexibilitet och effektivitet.

Strukturering och implementeringen är utförd objektorienterad. Detta ger en rad idag välkända fördelar beträffande programvarans flexibilitet. Dessutom utgör givarna och motorerna väl avgränsade fysiska objekt, vilket ytterligare motiverar en objektorienterad lösning eftersom objekten i mjukvaran då direkt kan representera de fysiska objekten. Programspråket C++ har använts. Erfarenheten är att objektorienteringen och C++ givit följande fördelar.

Ärvning ger en strukturerad uppdelning av egenskaper som är gemensamma för samtliga sensorer alternativt specifika för en viss sensortyp. Detsamma gäller för motorerna.

Virtuella procedurer för generella givare kan bindas till specifika procedurer för specifika givare i en viss systemkonfiguration. Detsamma gäller för ställdonen. Detta ger flexiblel programvara.

Operatorer kan definieras för egna klasser. Till exempel har omvandlingen av moment till strömreferenser definierats som en operator >>. På motsvarande sätt hanteras givarna via en motsvarande operator <<. Denna form av dataabstraktion gör att givare och ställdon hanteras som vanliga in- och utenheter vid implementering av regleralgoritmerna.

Bitmanipulering kan utföras på samma sätt som i C, vilket varit till nytta för vissa maskinnära funktioner.

Assemblermakron med parametrar har tidigare utvecklats [4]. Genom att utnyttja detta kan struktureringen av mjukvaran bibehållas utan att man ger avkall på snabbheten.

Mjukvaran behöver vara mycket snabb och är därför implementerad för Digitala Signal Processorer (DSP:er). Prestandakraven beror på att samplingsfrekvensen för givaravläsningen och motorstyrning ska kunna hållas mycket hög så att det blir möjligt att prova olika regleralgoritmer.

Arbetet kom även att innebära att brister i den medföljande basprogramvaran upptäcktes och korrigerades, Detta innebar att nya rutiner för dynamisk allokering och frigörning av minne inplementerades. Hantering av så kallade `static`-deklarerade variabler korrigerades, på ett sådant sätt att länkning av separata programdelar kunde ske på ett tillfredställande sätt. Assemblerrutiner för avbrottshantering korrigerades så att inte avbrott ska kunna förstöra variabler på stacken tillhörande den avbrutna bakgrundsprocessen.

Programvaran med alla funktioner implementerade fungerade utmärkt efter implementering i en DSP. Dock behövde snabbheten ökas 6 – 8 gånger för att uppfylla önskemålet att kunna läsa och skriva med vardera 62.5 kHz till och från roboten. I ett efterföljande projekt fördelades programmet på tre DSP:er. Detta resulterade i att läsningen och skrivningen till och från roboten kan utföras med 50 kHz samplingsfrekvens. Detta resultat får anses som mycket bra.

9. Referenser

[1] Stroustrup Bjarne *The C++ programming language*
second edition Addison-Wesley USA, 1991.

[2] Nilsson Klas *Application Oriented Programming and Control of Industrial Robots* TFRT-3212 Instutionen för Reglerteknik Lunds Tekniska Högskola, 1992.

[3] Kapilow David *Pi user's manual*
AT&T Bell laboratories USA, 1990.

[4] Matsson Ulf *Digital reglering med signalprocessorer och C++*
Teknisk rapport TFRT-5434 Instutionen för Reglerteknik Lunds Tekniska Högskola, 1991.

[5] Pär Larsson och Lars Åkerlind *Momentreglering av växelströmsmotorer för robotservo*
Teknisk rapport Teknikum Uppsala universitet, 1987.

[6] AT&T *WE DSP32 and DSP32C C Language Compiler User Manual.*
The AT&T Documentation Management Organization, 1988.

[7] AT&T *WE DSP32 and DSP32C C Language Compiler Library Reference Manual.*
The AT&T Documentation Management Organization, 1988.

[8] AT&T *WE DSP32C Digital Signal Processor Information Manual.*
The AT&T Documentation Management Organization, 1988.

[9] AT&T *WE DSP32C VME Board User Manual.*
AT&T Microelectronics, 1991.

A. Omvandling till strömreferenser

actuator

Filen actuator.h

```
//
//
//   File created by Lars Engelin started 920929
//
#ifndef __ACTUATOR_H
#define __ACTUATOR_H

struct TorqueAndPos{
  float torque;
  float res_pos;
};

class Actuator  {

public:

  Actuator(int HWadr);

  virtual Actuator& operator <<(TorqueAndPos& in)=0;


  int error_flag;

protected:

  friend class SerialOutPort;

  struct SerialOutFormat {
    unsigned long S_phase    : 12;
    unsigned long R_phase    : 12;
    unsigned long parity_bit :  1;
    unsigned long write_bit  :  1;
    unsigned long read_bit   :  1;
    unsigned long adress     :  5;
  };

  union SerialOutData {
    SerialOutFormat raw_data_bitmap;
    long            raw_data_signed;
    unsigned long   raw_data;
    unsigned char   raw_data_array[4];
  };

  SerialOutData out_to_serial;

  int new_value;
```

```
  int adr;

  long SendSerialData();

private:

  Actuator();
  // Prevent construction of a Actuator without using
  // the constructor with argument's

};

#endif
```

Filen actuator.c

```
/*

     File created by Lars Engelin started 920929

     Compile with:
       dspCC -g -c actuator.c
*/


#include "actuator.h"
#include "serialoutport.h"

long Actuator::SendSerialData(){

  new_value=0;
  return out_to_serial.raw_data_signed;
}

Actuator::Actuator(int HWadr){

  new_value=0;
  out_to_serial.raw_data=0;
  adr=HWadr;
  error_flag = !SerialOutPort::InstallActuator(HWadr,this);
}
```

ACmotor

Filen ACmotor.h

```
//
//
//   File created by Lars Engelin started 920929
//
#ifndef __AC_MOTOR_H
#define __AC_MOTOR_H

#include <stddef.h>
#include "actuator.h"

class AC_motor: public Actuator  {

public:

  Actuator& operator <<(TorqueAndPos& in);

  AC_motor(int HWadr,
   float n_o_p            = 4.0,
   float res_to_com_angle = 0.0,
   float comp             = 0.0,
   float ir_off           = 0.0,
   float is_off           = 0.0 );

  void* operator new(size_t size);
  void  operator delete(void* p);

private:

  float reshape(float angle);

  AC_motor();
  // So that nobody could use the constructor without argument's

  float res_to_comutation_angle,compensation,no_of_poles,
ir_offset,is_offset;

};

#endif
```

Filen ACmotor.c

```
/*

    File created by Lars Engelin started 920929

    Compile with:

      dspCC -g -c ACmotor.c
```

```
*/

#include "serialoutport.h"
#include "actuator.h"
#include "xmalloc.h"
#include "AC_motor.h"

extern "C" float sin(float);

typedef void (*PFVV)();
extern PFVV _new_handler;

void* AC_motor::operator new(size_t size)
{
  void* _last_allocation;

        while ((_last_allocation=xmalloc(unsigned(size)))==0)
{
                if(_new_handler && size)
                        (*_new_handler)();
                else
                        return 0;
        }
        return _last_allocation;
}

void AC_motor::operator delete(void* p)
{
        if (p) xfree( (char*)p );
}


float AC_motor::reshape(float angle){

/*
 Auxillary function to translate the angle to a value between
 -pi/2 and pi/2  so that the fast sinus routine will work.

 1/2pi  = 0.15915494
 pi/2   = 1.5707966
 pi     = 3.1415927
 3*pi/2 = 4.7123890
 2*pi   = 6.2831853
 5*pi/2 = 7.8539816

*/

  angle -= ( (int) (angle*0.15915494) ) * 6.2831853;
// angle should now be between -2*pi and 2*pi

  if (angle > 1.5707966) {
```

```
    if (angle >= 4.7123890){   // 3*pi/2 > angle
      angle -= 6.2831853;
    }
    else{                      //  pi/2 < angle < 3*pi/2
      angle = (angle-3.1415927) * -1.0;
    }
  }
  else if (angle < -1.5707966 ){

    if (angle < -4.7123890 ){  // -3*pi/2 > angle
      angle += 6.2831853;
    }
    else{                      // -3*pi/2 < angle < -pi/2
      angle = (angle+3.1415927) * -1.0;
    }
  }
  return angle;
}


Actuator& AC_motor::operator << (TorqueAndPos& in){

  register float angle;
  register float is;
  float ir;
  register int  ir_int, is_int;
  long ir_is;

  angle = (in.res_pos + res_to_comutation_angle + compensation )
        * no_of_poles;
  ir = ir_offset + in.torque * sin( reshape(angle) );
  is = is_offset + in.torque
       * sin( reshape(angle - 2.0943951) );
       // angle - 2pi/3 120 grader

  if (ir > 2047.0)   ir =  2047.0;
  else if (ir < -2048.0 ) ir = -2048.0;

  if (is > 2047.0)   is =  2047.0;
  else if (is < -2048.0 ) is = -2048.0;

  ir *= 4096.0;
  ir_int  = ( (int) ir ) & 0xFFF000;
  is_int  = ( (int) is ) & 0x000FFF;

  ir_is = (long)(ir_int | is_int);

  SerialOutPort::DisableSerialOutInterrupt();
     // To guarantee mutual exclusion
     out_to_serial.raw_data_signed  = ir_is;
     new_value++;
```

```
  SerialOutPort::EnableSerialOutInterrupt();

return *this;
}

AC_motor::AC_motor(int HWadr,
   float n_o_p ,
   float res_to_com_angle ,
   float comp ,
   float ir_off ,
   float is_off ):Actuator(HWadr)
{
  no_of_poles = n_o_p;
  res_to_comutation_angle = res_to_com_angle;
  compensation = comp;
  ir_offset = ir_off;
  is_offset = is_off;
}
```

serialoutport

Filen serialoutport.h

```
// File created by Lars Engelin in july 92

#ifndef __SERIALOUTPORT_H
#define __SERIALOUTPORT_H

#include "actuator.h"

class SerialOutPort {

/*
This class contains an interrupt routine ,which must be declared
static so that the routines in the intr module can manipulate
the interrupt vector in a 'proper' way. Of course having a
routine declared static will force you to have the variables in
that routine also declared static, this isn't any problem since
there ywill be only one variable of the type SerialOutPort.
*/

public:

  static void SerialOutInterrupt();

  static int InstallActuator(int HWadr ,Actuator* a);

  static void DisableSerialOutInterrupt();

  static void EnableSerialOutInterrupt();

  SerialOutPort();

protected:

  static int FindMostDemanding();


  struct SerialOutData{
    unsigned int  sending_demands;
    unsigned int  queue_number;
    Actuator*     actuator_pointer;
    union{
      unsigned long to_be_sent;
      long          to_be_sent_signed;
      unsigned char to_be_sent_array[3];
    };
  };

  static SerialOutData serial_out_queue[32];

  static unsigned int  serial_out_port_counter,
```

```
                        with_to_send_when_idle;

  static unsigned int  number_of_actuators;

  static unsigned long send_next_time;
};

#endif
```

Filen serialoutport.c

```
/*

   File created by Lars Engelin in July 92 modified in
   September and October

   Compile with:
     dspCC -g -c serialoutport.c

*/

#include <stddef.h>
#include "actuator.h"
#include "serialoutport.h"
#include <intr.h>
#include <io.h>

#ifndef NULL
#define NULL 0
#endif


unsigned      SerialOutPort::serial_out_port_counter = 0;

unsigned      SerialOutPort::with_to_send_when_idle  = 1;

unsigned      SerialOutPort::number_of_actuators     = 0;

unsigned long SerialOutPort::send_next_time          = 0;

SerialOutPort::SerialOutData
   SerialOutPort::serial_out_queue[32]=

{ {0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},
  {0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},
  {0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},
  {0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},
  {0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},
  {0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},{0,0,NULL},
  {0,0,NULL},{0,0,NULL} };

void SerialOutPort::SerialOutInterrupt(){
```

```
  // This is the obe interrupt routine

  io::put_long_obuf(send_next_time); // Send as fast as possible

  register unsigned int k;
  register unsigned int n_o_a = number_of_actuators;
  register int most_demanding = 1;
  register unsigned int l     = 0;


  for ( k=1; k <= n_o_a; k++ ){
    if ((serial_out_queue[k].actuator_pointer->new_value > 0))
        {
  serial_out_queue[k].to_be_sent_signed =
    serial_out_queue[k].actuator_pointer->SendSerialData();
  serial_out_queue[k].to_be_sent_array[3] = ( (k*8) | 6);
  // adress, write and read bits are added
  serial_out_queue[k].sending_demands++;
  serial_out_queue[k].queue_number = serial_out_port_counter++;
}

    if (serial_out_queue[k].sending_demands  |= 0){

      if(  serial_out_queue[k].sending_demands
 > serial_out_queue[most_demanding].sending_demands ){
   most_demanding =k;
   l++;
 }
      else
if ((  serial_out_queue[k].sending_demands
     == serial_out_queue[most_demanding].sending_demands )
    &&
    (serial_out_queue[k].queue_number
     <= serial_out_queue[most_demanding].queue_number) ){
       most_demanding = k;
       l++;
     }
    }
  }

  if( !l ){
    most_demanding = with_to_send_when_idle;
    serial_out_port_counter = 0;
    if (++with_to_send_when_idle > n_o_a){
     with_to_send_when_idle =1;
    // so that it never will overflow
    }
  }

  send_next_time=serial_out_queue[most_demanding].to_be_sent;
  // So that it will be sent next interrupt
```

```
  serial_out_queue[most_demanding].sending_demands = 0;
  // Clear all demands i.o.w. the data is sent

}

SerialOutPort::SerialOutPort(){    // Constructorn
#ifdef vme
  intr::set_obe_intr_addr(SerialOutPort::SerialOutInterrupt);
#endif
}

int SerialOutPort::InstallActuator(int HWadr ,Actuator* a){

  if ( (HWadr >= 32)||(HWadr < 0)||
       (serial_out_queue[HWadr].actuator_pointer != NULL) )
        return 0;
  serial_out_queue[HWadr].actuator_pointer=a;
  serial_out_queue[HWadr].to_be_sent_array[3]=
   ( (HWadr*8) | 6 );
  // So that something with a proper address, read and writebit
  // will be sent in idle mode before SendSerialData is called
  // for the first time
  number_of_actuators++;
  return 1;
}

void SerialOutPort::EnableSerialOutInterrupt(){

#ifdef vme
  intr::EI_obe();
#endif
}


void SerialOutPort::DisableSerialOutInterrupt(){

#ifdef vme
  intr::DI_obe();
#endif
}
```

B. Hanteringen av vinkelvärdena

sensor

Filen sensor.h

```
//
//    File created by Lars Engelin  started 920918
//
#ifndef __SENSOR_H
#define __SENSOR_H

typedef float PosVal;

class Sensor {
  // this is a general class with deals with inputs from
  // various sensors


public:

  int error_flag;

  int new_value;

  int adr;

  void GetSensorValue();

  Sensor(int HWadr);

  virtual Sensor& operator >>( PosVal& pos) = 0;

protected:

friend class SerialInPort;

  void StoreSensorData(register unsigned long* s);

  union{
    unsigned int sensor_data;
    int sensor_data_signed;
  };

private:
  Sensor();
// So that nobody could construct a Sensor variable without
// using the constructor with argument's

};
#endif
```

Filen sensor.c

```
/*

       File created by Lars Engelin started 920918

       Compile with:
        dspCC -g -c sensor.c
*/

#include "sensor.h"
#include "serialinport.h"

void Sensor::StoreSensorData(register unsigned long* s){

  sensor_data = (int)*s;
  new_value++;
}

Sensor::Sensor(int HWadr ){

  new_value    = 0;
  sensor_data  = 0;
  adr          = HWadr;
  error_flag   = !(SerialInPort::InstallSensor(HWadr , this));
}
```

resolver

Filen resolver.h

```
//
//
//    File created by Lars Engelin started 920918
//

#ifndef __RESOLVER_H
#define __RESOLVER_H

#include "sensor.h"

class Resolver : public Sensor {

public:

  Sensor& operator >>( PosVal& pos);
  // Since the return type is by defualt Resolver

  void* operator new(size_t size);
  void  operator delete(void* p);

  Resolver(int HWadr);


protected:

  float factor;
  Resolver();
  // So that nobody could use the constructor without argument's
};

#endif
```

Filen resolver.c

```
/*

      File created by Lars Engelin started 920918

      Compile with:
       dspCC -g -c resolver.c
*/


#include <stddef.h>
#include "xmalloc.h"
#include "serialinport.h"
#include "sensor.h"
#include "resolver.h"
```

```
typedef void (*PFVV)();
extern PFVV _new_handler;

Sensor& Resolver::operator >>( PosVal& pos){

  SerialInPort::DisableSerialInInterrupt();
      register float sensor = (float)sensor_data_signed;
      new_value=0;
  SerialInPort::EnableSerialInInterrupt();

  pos= sensor * factor; // (=9.587379926E-5)
                        // pi / 0x8000  enligt Klas 0x7FFF
  return *this;

}

void* Resolver::operator new(size_t size)
{
  void* _last_allocation;

        while ((_last_allocation=xmalloc(unsigned(size)))==0)
        {
                if(_new_handler && size)
                        (*_new_handler)();
                else
                        return 0;
        }
        return _last_allocation;
}

void Resolver::operator delete(void* p)
{
        if (p) xfree( (char*)p );
}


Resolver::Resolver(int HWadr)
:Sensor(HWadr){

  factor=9.587379926E-5;
}
```

serialinport

Filen serialinport.h

```
//
//
//    File created by Lars Engelin from 920918
//
#ifndef __SERIALINPORT_H
#define __SERIALINPORT_H

#include "sensor.h"

class SerialInPort {
// This class contains a static interrupt routine, that leads to
// that many of the variables have to be declared static.
// This fact doesn't have any effect since there will only be
// 1 variable of the type SerialInPort.

public:

  static int control_flag;
// This 'semaphore' is used to control program flow that
// should be executed after the interrupt is executed.

  SerialInPort();

  static void SerialInInterrupt();

  static void ClearControlFlag();

  static int InstallSensor(int HWadr ,Sensor* s);

  static void EnableSerialInInterrupt();

  static void DisableSerialInInterrupt();

private:

  static Sensor* adr_map[32];
// the location in the vector is equal to the adress from
// the hardware, and the location holds a pointer to the sensor.
};

#endif
```

Filen serialinport.c

```
/*

     File created by Lars Engelin from 920918

     Compile with:
```

```
      dspCC -g -c serialinport.c
*/

#include <stddef.h>
#include "sensor.h"
#include "serialinport.h"
#include <intr.h>
#include <io.h>

#ifndef NULL
#define NULL 0
#endif

int SerialInPort::control_flag = 0;


Sensor* SerialInPort::adr_map[32] = { NULL, NULL, NULL, NULL,
                                      NULL, NULL, NULL, NULL,
                                      NULL, NULL, NULL, NULL,
                                      NULL, NULL, NULL, NULL,
                                      NULL, NULL, NULL, NULL,
                                      NULL, NULL, NULL, NULL,
                                      NULL, NULL, NULL, NULL,
                                      NULL, NULL, NULL, NULL };

void SerialInPort::SerialInInterrupt(){
// Since this is an interupt routine it's running with all
// interrupts disabled and the mutal exclusion is therefore
//  quaranteed.

  register union{
    unsigned long m_unsigned;
    unsigned char m_array[4];
  };

  register unsigned char k;

#ifdef vme
  m_unsigned = io::get_long_ibuf();
#endif


  k=m_array[3];
  if( k & 0x04 ) // if the read-bit is set
    {
      k/=8;
      if (adr_map[k]!= NULL){
       // So that data from nonexsisting sensors
       // will not cause any trouble
m_array[3]=0; // remove adress and read/write bits
adr_map[k]->StoreSensorData(&m_unsigned);
      }
```

```
    }

  control_flag++; // So that routins waiting could run
}

void SerialInPort::ClearControlFlag(){
  DisableSerialInInterrupt();
    control_flag=0;
  EnableSerialInInterrupt();
}

SerialInPort::SerialInPort(){ // Constructorn
// The static variables are initiated at the begining
// of this file

#ifdef vme
  intr::set_ibf_intr_addr(SerialInPort::SerialInInterrupt);
#endif
}

int  SerialInPort::InstallSensor(int HWadr ,Sensor* s){

if ( (HWadr >= 32)||(HWadr < 0)||(adr_map[HWadr] != NULL) ){
   return 0;
}
adr_map[HWadr]=s;
return 1;
}

void SerialInPort::EnableSerialInInterrupt(){
#ifdef vme
  intr::EI_ibf();
#endif
}

void SerialInPort::DisableSerialInInterrupt(){
#ifdef vme
  intr::DI_ibf();
#endif
}
```

C. Dynamisk minnesallokering

malloc och free

```
/*
    Compile with:  (note It's the DSP32C C compiler)
            d3cc -g -c malloc.c

 Malloc and free for dsp32C created by Lars Engelin in
 October 1992
*/
#include "heapdef.h"

#ifndef NULL
#define NULL 0
#endif

#define heap_size 16384/4  /* Size of free memory */

long heap[heap_size];

struct HeapDef internal_heap =
       { &heap[0], &heap[heap_size-1], 1, &heap[heap_size-2],1};

struct HeapDef* heap_def_ptr = &internal_heap;

/*
       Figure of the heap
      --------------       heap[0]
      |             |
      |             |       allocation of new elements take place
      |             |        in increasing address order.
      |             |        The 4 bytes before each allocation
      |             |        holds the size of the allocation in
      |             |        long (1 long = 4 char(bytes) )
      |             |
      |             |
      |             |       Malloc allocates in the first segment
      |             |        big enough found in the Free List
      |             |        (i.o.w. it could be in any order in
      |             |        the heap).
      |             |
      |             |
      |             |       the Free List ( a table of free
      |             |        blocks in the heap) is expanding
      |             |        in decreasing address order
      --------------       heap[heap_size-1]

      Each allocation of an element in the Free List takes 2
       places of 4 bytes each, the first holds the address to
       the beginning of the free block and the second holds the
       length of the block.
```

```
      Each time malloc is called a additional 4 bytes is
       allocated in the address before the address pointed out
       by the returning pointer.
       These 4 bytes hold the size of the segment allocated
       including the 4 bytes.
*/


char* malloc(size)
  unsigned size;
{


  long *free_ptr,*extra;

  if(size == 0) return NULL;
  /*No point in allocating a block with the size 0 */

  if(heap_def_ptr->NumberOfFreeBlocks == 0) return NULL;
  /* if no free blocks at all */

  /* Check if not initialized then initialize it */

  if( heap_def_ptr->not_init ) {
    *(heap_def_ptr->LastFreeBlock)
     = (long) heap_def_ptr->HeapStart;
    *(heap_def_ptr->LastFreeBlock+1)
     = ( (long) (heap_def_ptr->HeapEnd-1)
      -(long)(heap_def_ptr->HeapStart) ) /4;
    heap_def_ptr->not_init=0;
  }

  /* At the very first adjust the size so it's even
     dividable with 4 since size is in bytes so far */

  size +=4;       /* So that the size itself could be stored */
  if(size & 3) { /* adjust the size so that it's */
    size +=4;     /* even dividable with 4 */
  }
  size/=4 ;

  /* First find a empty block large enough by locking
     in the free list */

  free_ptr=heap_def_ptr->HeapEnd-1;
  while ((free_ptr >= heap_def_ptr->LastFreeBlock)
         && (*(free_ptr+1) < size)){
    free_ptr -=2;
  }

  if (free_ptr < heap_def_ptr->LastFreeBlock) return NULL;
   /* couldn't find any block or any block big enough */
```

```
  extra=(long*) *free_ptr;

  /* Check for fragmentation */
  if( *(free_ptr+1) > size){ /* some fragmentation */
    *(free_ptr+1) -= size;
    *free_ptr += (size*4);
  }
  else {

    /* This is the case with no fragmentation i.o.w.
      remove the entry from the Free List */

    *free_ptr      = *heap_def_ptr->LastFreeBlock;
    *(free_ptr+1)  = *(heap_def_ptr->LastFreeBlock+1);
    heap_def_ptr->LastFreeBlock += 2;
    heap_def_ptr->NumberOfFreeBlocks--;
  }
  *extra = size;
  return (char*) extra+4;
/* returns a pointer to the allocated space */
/* after the type cast you have to add 4 instead of 1*/

} /* End of malloc */




free(address)
  char* address;
{
  long *to_be_free,*resized_free_block,*free_ptr;
  int add_to_exsisting_free_block=0;
  long first_after_to_be_free =0;


  if(address == NULL) return;
/* No point in testing any more */

  to_be_free = (long*) address-1;

  if ( (to_be_free < heap_def_ptr->HeapStart) ||
       (to_be_free > heap_def_ptr->HeapEnd-1) ) return;
  /* Check to see so that's a valid address */

  free_ptr = heap_def_ptr->HeapEnd-1;

  /* search to see if there is a free block infront of the
     new free block if so make it into 1 big free block */
  while(( (long*) (*free_ptr+(*(free_ptr+1))*4) != to_be_free)
         && (free_ptr >= heap_def_ptr->LastFreeBlock)){
```

```
  free_ptr -=2;
}
  if (free_ptr >= heap_def_ptr->LastFreeBlock) {
    /* found empty space in front of the block
       about to be deallocated */

    *(free_ptr+1) += *to_be_free;
    /* add the new bit to the existing block */
    add_to_exsisting_free_block++;
    resized_free_block = free_ptr;
  }
  free_ptr = heap_def_ptr->HeapEnd-1;

    /* search to see if there is a free block
       after the new free block */
  first_after_to_be_free = (long) (to_be_free)+(*to_be_free)*4;

  while( (*free_ptr != first_after_to_be_free ) &&
(free_ptr >= heap_def_ptr->LastFreeBlock)){
  free_ptr -=2;
}
  if (free_ptr >= heap_def_ptr->LastFreeBlock) {
    /* found empty space after the block
       about to be deallocated */

    if (add_to_exsisting_free_block) {
      /*If there will be 1 new block instead of
        3 seperate blocks */
      *(resized_free_block+1) += *(free_ptr+1);
      if (free_ptr != heap_def_ptr->LastFreeBlock){
        /* if the block to remove from the FreeList isn't the
           last one in the list */
*free_ptr     = *heap_def_ptr->LastFreeBlock;
*(free_ptr+1) = *(heap_def_ptr->LastFreeBlock+1);
      }
      heap_def_ptr->LastFreeBlock +=2;
      /* remove the last entry in the FreeList */
      heap_def_ptr->NumberOfFreeBlocks--;
    }
    else{
    /* add the new free block to the existing free block */
      *free_ptr      = (long) to_be_free;
      *(free_ptr+1) += *to_be_free;
    }
    add_to_exsisting_free_block++;
  } /* if... found empty space after the block about ... */

  if (!add_to_exsisting_free_block) {
    /* if there isn't any free space infront or after the
       block about to be deallocated */
    heap_def_ptr->LastFreeBlock -=2;
    *(heap_def_ptr->LastFreeBlock) = (long) to_be_free;
```

```
    *(heap_def_ptr->LastFreeBlock+1) = *to_be_free;
    heap_def_ptr->NumberOfFreeBlocks++;
  }
}
```

xmalloc och xfree

```
/*
    Compile with:  (note It's the DSP32C C compiler)
            d3cc -g -c xmalloc.c

This file implements xmalloc and xfree which are used to
allocate and deallocate memory in the DRAM memory with is
shared with others.

 xmalloc calls malloc which is found in the file malloc.c, and
 xfree calls free which also is found in malloc.c
*/

#include "heapdef.h"
extern char* malloc();
extern void free();

#define heap_size 0x4000/4

long ext_heap[heap_size];

struct HeapDef external_heap =
       { &ext_heap[0], &ext_heap[heap_size-1], 1,
         &ext_heap[heap_size-2],1};

char *xmalloc(size)
unsigned int size;
 {
   char *ptr;
   struct HeapDef *oldptr = heap_def_ptr;
   heap_def_ptr = &external_heap;
   ptr = malloc(size);
   heap_def_ptr = oldptr;
   return ptr;
 }

void xfree(pointer)
char* pointer;
 {
   struct HeapDef *oldptr = heap_def_ptr;
   heap_def_ptr = &external_heap;
   free(pointer);
   heap_def_ptr = oldptr;
 }
```