ISSN 0280–5316
ISRN LUTFD2/TFRT--5528--SE

Öppna regulatorer för inbyggda system

Olof Laurin

Institutionen för Reglerteknik
Lunds Tekniska Högskola
April 1995

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

Document name
MASTER THESIS

Date of issue
April 1995

Document Number
ISRN LUTFD2/TFRT--5528--SE

Author(s)
Olof Laurin

Supervisor
Klas Nilsson

Sponsoring organisation
NUTEK – the National Board for Industrial and Technical Development

Title and subtitle
Öppna regulatorer för inbyggda system
(Open regulators for embedded systems)

Abstract

Embedded regulators today usually allow parameter changes, and possibly activation of different preimplemented algorithms. Full reprogramming using the complete source code is not allowed for safety, efficiency, and proprietary reasons. This means that for these reasons, embedded regulators are quite rigid and closed concerning the control structure.

In several applications, like industrial robots, there is a need to tailor the low level control to meet specific application demands. Considering the efficiency and safety demands, a way of building more generic and open regulators has been developed. The key idea is to use compiled executable code as control parameters. In an object oriented framework, this means that new methods can be added to controller objects after implementation of the basic control, and even while the controller is running.

The implementation was carried out in the industrially well accepted language C++. The dynamic binding at run-time differs from ordinary dynamic linking in that only a subset of the symbols can be used. This subset is defined by the fixed part of the system. The safety demands can therefore still be fulfilled. Encouraged by results from some fully implemented test cases, we believe that extensive use of this concept will allow more open, still efficient, embedded systems to be developed.

Key words
Embedded systems, Open systems, Real-time systems, Object oriented programming, Dynamic binding.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
Swedish

Number of pages
34

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, Fax +46 46 110019, Telex: 33248 lubbis lund.

Innehåll

1. Inledning . . . 2

2. Bakgrund . . . 3

2.1 Struktur . . . 3
2.2 Experimentplattform . . . 3
2.3 Exempel . . . 3
2.4 Kompilerade dynamiska metoder . . . 4
2.5 Omgivning och inkopplingspunkter . . . 5

3. Dynamiska metoder . . . 6

3.1 Översikt . . . 6
3.2 Action . . . 8
3.3 ActionSlot . . . 9
3.4 ActionSlotHolder . . . 10
3.5 ActionLoader . . . 10
3.6 ActionIntParameter . . . 11
3.7 ActionFloatParameter . . . 11
3.8 ActionDoubleParameter . . . 12
3.9 ListableObject . . . 12
3.10 ListableObjectList . . . 12

4. Klient/Serverprogram . . . 13

4.1 Översikt nätverk . . . 13
4.2 SocketActionLoader . . . 13
4.3 Översikt värddatorn . . . 14
4.4 Kompilering och länkning . . . 14
4.5 Paketering . . . 15
4.6 Sändning . . . 16

5. Objektorienterat gränssnitt mot realtidskärnan . . . 17

5.1 Översikt . . . 17
5.2 Semaphore . . . 17
5.3 Monitor . . . 18
5.4 Mailbox . . . 19
5.5 Time . . . 20
5.6 RTProcess . . . 21
5.7 MatCommSocket . . . 24
5.8 SocketLoadServer . . . 24

6. Vidareutveckling av actions . . . 26

7. Kommentarer . . . 27

7.1 Erfarenheter . . . 27
7.2 Slutsatser . . . 27

Referenser . . . 29

A. Programkod . . . 30

B. Tillämplingsexempel . . . 32

1. Inledning

Vid Institutionen för Reglerteknink, LTH, bedrivs forskning inom realtidsystem och robotstyrning. En målsättning är att utveckla flexiblare inbyggda regulatorer. För att åstadkomma detta är en skiktad regulatorstruktur lämplig. Mellan skikt kan data sändas och funktionsanrop göras. En av idéerna är att även exekverbar kod kan överföras. Då kan den övre nivån bestämma vilka funktioner som ska användas samt när, om och var de skall kopplas in. Den undre nivån kan tillhandahålla inkopplingspunkter samt bestämma vad som får användas.

Överföring av exekverbar kod skiljer sig endast delvis från överföring av data. På den sändande nivån är de lika, men på den mottagande nivån behövs inkopplingspunkter för koden. Dessa inkopplingspunkter krävs för att kunna anropa koden. En sådan kan t.ex. vara en konstant parameter som kan bytas mot en funktion. Detta kan då svara mot att en linjär regulator under drift byts mot en olinjär. Mottagaren tillhandahåller inkopplingspunkter samt bestämmer vilka och hur variabler och funktioner får användas. Sändaren bestämmer vilka funktioner som *ska* anropas. Vidare bestämmer sändaren när och om de ska kopplas in samt till vilken inkopplingspunkt de skall kopplas.

Idag är objektorientering den paradigm som oftast används vid programutveckling i industrin. Därför har jag valt att göra implementationen objektorienterad i programspråket C++. Vid institutionen används en realtidskärna skriven i Modula-2. Mot denna finns även ett funktionsgränssnitt i C. Detta gränssnitt har jag delvis anpassat till objektorientering och C++.

Syftet med detta examensarbete är att visa att överföring av exekverbar kod är genomförbar i praktiken. Detta skall demonstreras genom implementering av principen för några testexempel.

Tyngdpunkten i rapporten ligger på implementationen för att kunna överföra kod, men även anpassning av institutionens realtidskärna till C++ beskrivs. Följande kapitel ingår:

- Kapitel 2, Bakgrund, beskriver idéerna bakom att överföra kod. Vidare beskrivs experimentplattformen, tre testexempel presenteras och en del begrepp definieras.
- Kapitel 3, Dynamiska metoder, beskriver implementationen av examensarbetets centrala delar. Det är de delar som hanterar mottagning och organisering av nedladdad kod i målsystemet.
- Kapitel 4, Klient/Serverprogram, beskriver implementationen av de delar som är specifika för den miljö huvuddelen av arbetet är utfört i. Vidare beskrivs funktioner och hjälpmedel på värddatorn.
- Kapitel 5, Objektorienterat gränssnitt mot realtidskärnan, beskriver anpassningar av kärnans gränssnitt till objektorientering och C++.
- Kapitel 6, Vidareutveckling av actions, beskriver översiktligt en vidareutvecklad metod som löser de problem som framkommit under detta examensarbete.
- Kapitel 7, Kommentarer, beskriver slutligen erfarenheter och slutsatser från arbetet.
- I appendix A innehåller centrala delar av den skrivna källkoden.
- I appendix B visar hur typexemplen har lösts.

2. Bakgrund

2.1 Struktur

I en öppen regulatorarkitektur vill man ha möjlighet att enkelt förändra eller byta ut regleralgoritmen eller delar av den. Dessutom vill man kunna modifiera fast implementerade algoritmer utifrån tillämpningsspecifika krav, utan att ändra i den fasta delen. Då faller det sig naturligt att dela upp arkitekturen i lager. Sådana strukturer är vanliga inom datakommunikation. För inbyggda styrsystem finns dock fall då inte bara data behöver kunna överföras mellan lagren, utan även effektivt exekverbara program. Detta examensarbete går ut på att experimentellt undersöka möjligheterna att överföra data i form av kompilerad kod mellan lagren. För en noggrannare beskrivning av den öppna regulatorarkitekturen se [4].

2.2 Experimentplattform

Den experimentplattform som har använts har tre fysiska nivåer. Överst finns värddatorn, en SUN arbetsstation. På den sker all filhantering och kompilering. Här finns även användargränssnitt med kontrollpanel och plotting. På nivån under finns ett VME-system med en 68040 processor. Till denna finns en realtidskärna utvecklad på institutionen. Den tillåter samplingshastigheter upp till 1 kHz. Mellan dessa system finns två kommunikationskanaler. Dels en seriell förbindelse, dels en förbindelse via ethernet. Genom den seriella förbindelsen skickas enklare kommandon och utskrifter. Via ethernet skickas all övrig information. Det kan vara programkod, plotvärden, signaler från kontrollpanelen etc. I VME-systemet finns även den tredje fysiska nivån. Den består av DSP32C-signalprocessorer. Sådana är optimerade för snabb beräkning av typiska filter- och regleralgoritmer.

2.3 Exempel

Nedan presenteras tre väl avgränsade exempel. Dessa exempel visar på problem som är svåra att lösa med traditionella implementeringsmetoder.

Enkel regulator

Under utvecklingen av en enkel P-regulator vill man prova att ersätta konstanten med olika funktioner. För att använda dessa krävs att de kompileras in i regulatorn från början. Ett trevligare sätt vore att byta konstanten utifrån. Funktionerna som ersätter konstanten skulle vara helt skiljda från regulatorn vid kompileringen. Med denna metod skulle regulatorn *inte* behöva omkompileras om en ny funktion ska provas.

Plottning

En regulator utvecklas för ett system. Vid olika typer av laststörningar vill man plotta regulatorns initiala beteende då störningen läggs på. Under dessa

korta tider vill man spara och plotta värden med en så hög hastighet att kontinuerlig loggning inte är möjlig. Svårigheten ligger i att sätta igång loggningen vid rätt tillfälle. En intressant metod vore om man kunde specificera externa triggfunktioner. Dessa skulle kunna avläsa tillstånd, in- och utsignaler kontinuerligt. Då dessa uppfyller något villkor sätts loggningen igång.

Avancerad regulator

En avancerad regulator ska användas för att reglera en ny olinjär process. Olinjäriteterna är inte stora och man försöker använda en linjär regulator. Det visar sig att det inte är tillräckligt och man tvingas börja experimentera med funktioner i matrisen för tillståndsåterkoppling. Detta kan vara mycket tidsödande då regulatorn måste kompileras om mellan varje försök. Det är dessutom omöjligt att använda en färdig regulator. En attraktivare lösning vore att ha en regulator vars parametrar kan bytas mot kompilerad kod. Inifrån denna kod kommer man endast åt förutbestämda variabler och funktioner i den inbyggda regulatorn, vilket skiljer metoden från vanlig dynamisk länkning.

2.4 Kompilerade dynamiska metoder

När man programmerar är det ofta en modell av verkligheten man beskriver. Verkligheten är uppbyggd av objekt. På dessa kan man applicera metoder. Vid objektorienterad programmering bygger man klasser. Man kan sedan skapa objekt av dessa klasser. I kompilerande språk måste metoderna vara definierade vid kompileringen. Interpreterande språk har inte denna begränsning, men de kan i allmänhet inte exekvera kod lika effektivt som kompilerande språk. Däremot tillhandahåller kompilerande språk för objektorienterad programmering både statisk och dynamisk bindning av metoder. Ett centralt begrepp i denna rapport är *kompilerade dynamiska metoder*. De definieras dynamiskt och binds dynamiskt till ett körande system under drift. De behöver inte finnas i systemet från början utan kan laddas medan det kör. En kompilerad metod kan vara allt från en funktion som returnerar en konstant till ett komplett system med modeller av verkligheten som ska regleras. En kompilerad metod kan också skapa en process när den installeras.

Målsättningen har varit att en kompilerad metod ska ha samma prestanda som om målsystemet hade kompilerats med funktionaliteten från början. Vidare kan det vara stränga minneskrav i målsystemet, t.ex. kan det vara en signalprocessor med allt minne integrerat med processorn och inget externt minne. Med denna bakgrund har följande metodik valts:

- All kompilering, länkning etc. skall ske i värddatorn.
- Kompilatorn måste generera PC-relativ, positionsoberoende kod.
- Länkaren kan inte länka koden till en viss position.
- En kompilerad metod anropas som en funktion med två inparametrar. Den första parametern är en pekare på en struktur som innhåller variabler och funktionspekare som ibland behövs i den kompilerade metoden. Den andra är en pekare på data som metoden själv kan allokera.
- En kompilerad metod skall kunna kopplas in i en körande regulator.

Efterhand visade det sig att kompilatorer inte alltid kan generera PC-relativ, positionsoberoende kod. I kapitel 6 presenteras därför kort ett modifierat angreppsätt som kräver en enkel länkare. Denna kan användas för fortsatt arbete.

2.5 Omgivning och inkopplingspunkter

Den första inparametern till en kompilerad metod är en pekare på en struktur. Denna innehåller variabler och pekare på funktioner som man eventuellt vill använda och anropa. Utan länkare är detta enda sättet att komma åt systemet. Denna struktur kallas omgivningsstruktur i denna rapport.

Det måste finnas mottagare för kompilerade metoder. Dessa mottagare kan använda metoden på olika sätt. Exempelvis kan en mottagare exekvera en metod en gång när den laddas ned och sedan aldrig mer. Då kan t.ex. metoden starta processer och själv se till att exekveras. En annan typ av mottagare ser till att en metod exekveras varje sampel och ytterligare en annan ser till att en metod exekveras varje gång en parameter används.

3. Dynamiska metoder

Dynamiska metoder och actions används synonymt i denna rapport. Med detta menas en avgränsad bit nedladdningsbar, kompilerad kod. En action kan variera avsevärt i omfattning.

3.1 Översikt

Exemplen i avsnitt 2.3 visar att nedladdningsbara, färdigkompilerade metoder verkar lösa vissa typer av problem. Denna rapport beskriver en implementation av actions. Ett system där sådana metoder medför stora fördelar har typiskt egenskaper som:

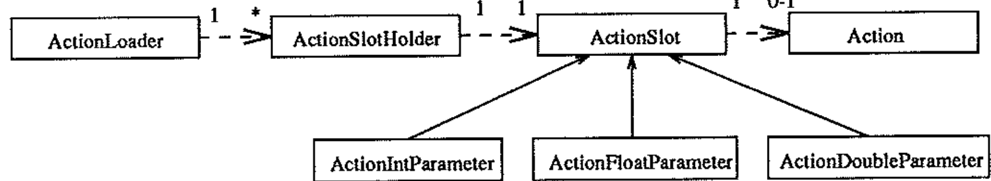
- Det är distribuerat.
- Det är heterogent.
- Det är maskinnära.
- Det är inbyggt.
- Det finns tidskrav (realtid).
- Det finns ibland stränga minneskrav.
- Det finns krav på effektiv exekvering.
- Det finns säkerhetskrav därför att maskiner och människor kan skadas om systemet fungerar felaktigt.

Två grundläggande problem måste lösas. Dels ska kod kunna tas emot, dels ska den placeras någonstans. Den metod som valts för att placera koden är att det finns en generell typ av behållare för nedladdad kod. Strukturen är dels lämpad för parametrar som kan ersättas med en action, dels för mera generella metoder. Att parametrar behandlas separat beror på att det är svårt att få en parameter att fungera både som konstant och som mottagare av actions. En alternativ metod vore att actions samlas gruppvis i objekt. Objektet sköter sedan om att exekvera respektive action, men det är svårt att få lösningen så generell att den blir lättanvänd.

Då det ska vara enkelt att anpassa konstruktionen till ny hårdvara har den delats i två delar. Den första delen beskrivs i detta kapitel och den är ej bunden till specifik hårdvara. Den andra delen innehåller allt hårdvaruspecifikt och är beskriven i kapitel 4. Den behandlar kommunikationen med värddatorn. Därför finns endast begränsad funktionalitet för kodmottaging i detta kapitel, tonvikten ligger istället på organisering av den mottagna koden i målsystemet. Utifrån dessa överväganden har nedstående implementation gjorts.

Den implementerade metoden för att ladda ned kod består av ett antal klasser i C++. Se figur 3.1. I figuren beskriver fyrkanter klasser, heldragna pilar betyder ärvning och streckade pilar betyder "använder". En heldragen pil från klass A till klass B betyder att klass A är härledd från klass B. En streckad pil från klass A till klass B betyder att klass A använder klass B. Vid varje ände av en streckad pil finns en siffra eller en asterisk. Siffrorna visar antalet i förhållande till varandra. Asterisk betyder noll eller flera.

Betydelsen av figuren är alltså att klassen **ActionLoader** använder noll eller flera **ActionSlotHolder**. I varje **ActionSlotHolder** finns en **ActionSlot** och i varje **ActionSlot** finns ingen eller en **Action**. **ActionIntParameter**, **ActionFloatParameter** och **ActionDoubleParameter** är alla

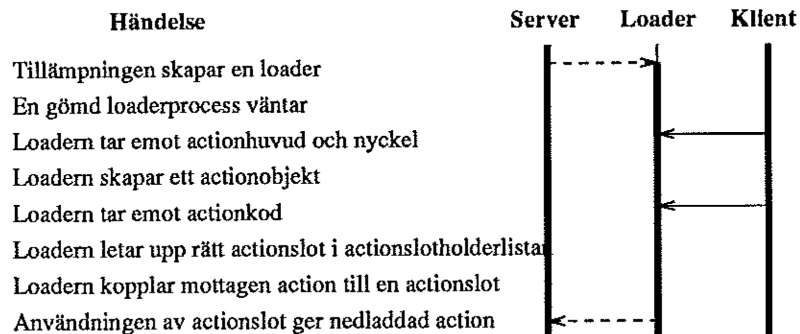


Figur 3.1 Klasstruktur för actionklasser.

härledda från **ActionSlot**. Det finns endast en basklass för kodmottaging, **ActionLoader**. Denna är ej bunden till någon kommunikationskanal utan är främst ett hjälpmedel vid nedladdning. Konstruktionen är sådan att godtyckligt antal kodmottagare kan finnas i systemet samtidigt. Från **ActionLoader** härleder man lämpligen en klass för mottagning, se avsnitt 4.2.

Klasstrukturen är resultatet av flera avvägningar. I princip kan de flesta klasser där förhållandet mellan dem är ett till ett sammanslås. Anledningen att **ActionSlotHolder** är en separat klass är att den innehåller en nyckel. Denna nyckel används av **ActionLoader** för att finna rätt **ActionSlot**objekt. Man vill däremot inte att de härledda klasserna **ActionIntParameter** m.fl. ska innehålla någon nyckel. En nyckel hör inte ihop med det sätt dessa parameterklasser används.

I figur 3.2 visas ett interaktionsdiagram för laddning av en action. I denna figur används begreppen klient/server om värddator/målsystem. Märk att målsystemet är server och att värddatorn är klient.



Figur 3.2 Interaktionsdiagram för laddning av en action.

Ett enkelt exempel på hur dessa klasser kan användas kan se ut som nedan:

```
class SpecialActionLoader : ActionLoader {
  // implementerar en actionloader kopplad till hårdvara
};
typedef struct {
  double (*sqrt_fp)( double );
  float uc;
  float y;
} Env;
void main {
  Env env;
  env.sqrt_fp = sqrt;
  SpecialActionLoader loader;
  ActionFloatParameter K( 2.0, &env, "K", &loader, 0 );
```

```
  for (;;) {
    ...
    P = K * (env.uc - env.y);
  }
}
```

Detta exempel börjar med att härleda en klass för mottagning kopplad till hårdvara. Därefter definieras omgivningsstrukturen. I huvudprogrammet fylls omgivningsstrukturen i, en mottagare skapas och en parameter K skapas. K initialiseras till 2.0 men den kan närsomhelst ersättas med en action. K kan hela tiden användas som en variabel trots att det kan dölja sig en actionfunktion bakom den.

Ett exempel på en action som använder samma omgivningsstruktur som ovan presenteras kort. Den ersätter konstanten K i en P-regulator med en olinjär funktion utom i ett intervall runt origo.

```
float K( Env* env, void** data )
{
  float e = (env->y - env->uc);
  if (e > 0.0625) {
    return 0.5/((*env->sqrt_fp)(e));
  } else if (e < -0.0625) {
    return 0.5/((*env->sqrt_fp)(-e));
  } else {
    return (2.0);
  }
}
```

Ovanstående action fungerar utmärkt vid reglering av ett servo anslutet till experimentplattformen.

Resten av detta kapitel är implementationsbeskrivningar av klasserna som hanterar kodmottagning, lagring och exekvering av actions. Av prestandaskäl har **inline**-deklarationer gjorts på alla funktioner där detta varit möjligt.

3.2 Action

Objekt av klassen **Action** fungerar som behållare för nedladdad kod. Huvudbeståndsdelen är en pekare på allokerat minne. I detta ligger den nedladdade koden. Klassen har ett antal funktioner för att underlätta nedladdning av kod. Vidare finns det exekveringsfunktioner som exekverar actionobjektets kod. Dessa exekveringsfunktioner skiljer sig åt endast i fråga om typen på returvärdet.

Nedan följer deklarationen av klassen **Action**. I rapporten används en notation där **m_** indikerar en variabel som är attribut i en klass. Vidare betyder **sm_** statisk variabel, dvs den delas av alla objekt av den klassen.

```
class Action {
public:
  inline Action( unsigned int id, char *name,
    unsigned int virtualstart,
    unsigned int blocksize,
    unsigned int virtualentry);
  inline ~Action();
  inline void ReturnVoid( void* env );
```

```
  inline int ReturnInt( void* env );
  inline float ReturnFloat( void* env );
  inline double ReturnDouble( void* env );
  inline unsigned int GetVirtualStart();
  inline unsigned int GetSize();
  inline char* GetPtr();
  inline unsigned int GetID();
protected:
  unsigned int m_id;
  char m_name[32];
  unsigned int m_virtualstart;
  unsigned int m_blocksize;
  unsigned int m_virtualentry;
  char* m_block;
  void* m_data;
};
```

Publika attribut: Funktionen **ReturnVoid** anropar den nedladdade koden utan att returnera något värde. **ReturnInt** gör samma men returnerar ett heltal. Detsamma gäller **ReturnFloat** och **ReturnDouble** men de returnerar ett reellt tal respektive ett reellt tal med dubbel precision. Övriga funktioner är hjälpfunktioner vid nedladdningen. Funktionen **GetVirtualStart** ger den minnesposition nedladdningsprogrammet har angett som första position. **GetSize** ger det allokerade minnesområdets storlek. **GetPtr** ger en pekare till början av det allokerade minnesområdet. **GetID** ger actionfunktionens ID. Detta ID kommer från nedladdningsprogrammet. Detta kan användas på olika sätt men i resten av denna implementation används det uteslutande som identifiering, nyckel, för var en nedladdad action ska kopplas in.

Skyddade attribut: Skyddade attribut är endast tillgängliga i härledda klasser. Variabeln **m_id** lagrar den nedladdade actionfunktionens ID. Variabeln **m_name** är en sträng som lagrar actionfunktionens namn. Variabeln **m_virtualstart** innehåller den minnesposition nedladdningsprogrammet har angett som första position. Variabeln **m_blocksize** innehåller storleken på det allokerade minnesområdet. Variabeln **m_virtualentry** är den minnesposition som skall anropas i den nedladdade koden. Denna är angiven i samma adressrymd som **m_virtualstart**. Variabeln **m_block** är en pekare på det allokerade minnesområdet. Variabeln **m_data** är en pekare som en nedladdad action kan använda hur den vill. Den kan t.ex. allokera dynamiskt minne för egna statiska variabler.

3.3 ActionSlot

Klassen **ActionSlot** är huvudsakligen en basklass. Man härleder från den då man vill ha objekt som kan innehålla en action. T.ex. kan man härleda en parameter som har ett konstant värde men kan ta emot en actionfunktion. Då ersätter actionfunktionen värdet då parametern används. Se vidare actionparameterklasserna, avsnitt 3.6 – 3.8.

```
class ActionSlot {
public:
  inline ActionSlot();
```

```
  inline void SetAction( Action* action );
protected:
  Action* m_action;
};
```

Publika attribut: Funktionen **SetAction** används för att associera en action med ett **ActionSlot**objekt.

Skyddade attribut: Variabeln **m_action** är en pekare på en action. Om denna pekare är noll betyder det att ingen action har blivit nedladdad till detta objektet.

3.4 ActionSlotHolder

Klassen **ActionSlotHolder** associerar ett **ActionSlot**objekt med en nyckel. Objekt av denna klass finns i en lista i ett **ActionLoader**objekt, se avsnitt 3.5. När en action med en viss nyckel laddas ned hamnar den på rätt ställe med hjälp av associationerna i objekt av denna klass. Basklassen **ListableObject** finns beskriven i avsnitt 3.9.

```
class ActionSlotHolder : public ListableObject {
public:
  inline ActionSlotHolder( ActionSlot* actionslot, int key );
  inline ActionSlot* GetActionSlot();
  inline int GetKey();
protected:
  ActionSlot* m_actionslot;
  int m_key;
};
```

Publika attribut: Funktionen **GetActionSlot** ger en pekare på **ActionSlot**objektet. Funktionen **GetKey** ger den nyckel som är associerad till **ActionSlot**objektet.

Skyddade attribut: Variabeln **m_actionslot** är en pekare på det associerade **ActionSlot**objektet. Variabeln **m_key** är den (heltals-) nyckel som är associerad till **ActionSlot**objektet.

3.5 ActionLoader

Klassen **ActionLoader** är en basklass som inte gör något själv. Den hanterar gemensam funktionalitet för kodladdare. Dess huvuduppgift är att hantera en lista av **ActionSlotHolder**objekt. Den innehåller även en pekare på ett **Action**objekt om ett sådant håller på att tas emot. Klassen **ListableObjectList** finns beskriven i avsnitt 3.10.

```
class ActionLoader {
public:
  inline ActionLoader();
  inline ~ActionLoader();
  inline void RegisterSlot( ActionSlot* actionslot, int key );
protected:
```

```
  Action* m_currentaction;
  ListableObjectList m_actionslotlist;
  inline void SetAction();
};
```

Publika attribut: Funktionen **RegisterSlot** registrerar ett **ActionSlot**-objekt och associerar det med en nyckel.

Skyddade attribut: Variabeln **m_currentactionslot** är en pekare på det **Action**objekt som håller på att mottagas. Om mottagning inte pågår är denna pekare noll. Variabeln **m_actionslotlist** är en pekare på den lista som innehåller positioner för nedladdad kod (**ActionSlot**objekt). Funktionen **SetAction** flyttar den action som precis blivit färdignedladdad till rätt actionslot.

3.6 ActionIntParameter

Klassen **ActionIntParameter** är härledd från klassen **ActionSlot**. Objekt av denna typ är heltalskonstanter som kan laddas med actions så att de blir funktioner. Detta kan användas för att bygga en linjär regulator som kan göras om till en olinjär. Vid nedladdning ersätter actionfunktionens resultat värdet parametern gavs vid initeringen då den används.

För att uppfylla kravet på typsäkerhet behöver man härleda en ny klass från **ActionIntParameter**. Den härledda klassen behöver ingen egen funktionalitet utan fungerar endast som typsäkring.

```
class ActionIntParameter : public ActionSlot {
public:
  inline ActionIntParameter( int initialvalue, void* env,
    char* name, ActionLoader* actionloader, int key );
  inline operator int();
protected:
  int m_value;
  void* m_env;
  char m_name[32];
};
```

Publika attribut: Operatorn **int** returnerar ett värde. Om ingen action finns laddad är värdet det som angavs vid initialiseringen. Annars exekveras actionfunktionen och resultatet av den returneras.

Skyddade attribut: Variabeln **m_value** är det konstanta värde parametern gavs när den skapades. Variabeln **m_env** är en pekare på en omgivningsstruktur, se avsnitt 2.5. Variabeln **m_name** är namnet på parametern.

3.7 ActionFloatParameter

Klassen **ActionFloatParameter** är härledd från klassen **ActionSlot**. Den är identisk med klassen **ActionIntParameter** förutom att värdet den hanterar är reellt (float).

3.8 ActionDoubleParameter

Klassen **ActionDoubleParameter** är härledd från klassen **ActionSlot**. Den är identisk med klassen **ActionIntParameter** förutom att värdet den hanterar är reellt med dubbel precision (double).

3.9 ListableObject

Klassen **ListableObject** är en mycket enkel basklass för objekt som ska kunna placeras i en lista.

```
class ListableObject {
public:
  ListableObject *m_next;
  inline ListableObject();
};
```

Publika attribut: Variabeln **m_next** är en pekare på nästa element i en lista. När elementet skapas initialiseras denna pekare till noll.

3.10 ListableObjectList

Klassen **ListableObjectList** är en mycket enkel listklass. Ett objekt av denna typ innehåller en pekare på första elementet i listan samt några metoder att använda på listan. Godtyckliga objekt som har klassen **ListableObject** som basklass kan sättas in i listan blandat. Det finns ingen metod implementerad för att ta bort element ur listan. Detta på grund av tidsbrist.

```
class ListableObjectList {
public:
  ListableObject *m_first;
  inline ListableObjectList();
  inline void Insert( ListableObject* listableobject );
  inline int NumElems();
};
```

Publika attribut: Variabeln **m_first** är en pekare på första elementet i listan. När listan skapas initialiseras denna pekare till noll. Funktionen **Insert** stoppar in ett objekt sist i listan. Funktionen **NumElems** ger antalet element i listan.

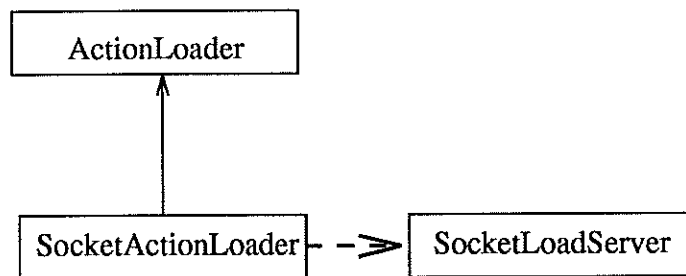
4. Klient/Serverprogram

Detta kapitel beskriver implementationen av de delar som är specifika för den miljö arbetet är utfört i. Vidare beskrivs funktioner och hjälpmedel på värddatorn.

4.1 Översikt nätverk

Den grundläggande implementationen för att ladda actions är oberoende av hårdvara. Detta gäller även kommunikationen och därför ingår endast en basklass för mottagarobjekt i grunden. Denna behöver kompletteras med specifika härledda klasser för olika typer av kommunikationskanaler. Nedan presenteras en sådan klass med hjälpklass för mottagning via experimentplattfomens ethernetkommunikation. Konstruktionen är sådan att flera mottagningsobjekt kan finnas samtidigt.

Systemet för att ladda ned kod över nätverket består av ett antal klasser i C++. Se figur 4.1. De olika symbolerna har samma betydelse som i figur 3.1.



Figur 4.1 Klasstruktur för socketimplementationen.

Betydelsen av figuren är alltså att klassen **SocketActionLoader** är härledd från **ActionLoader**. **SocketActionLoader** använder sig av klassen **SocketLoadServer**. Klassen **SocketLoadServer** beskrivs i avsnitt 5.8 och **ActionLoader** beskrevs i avsnitt 3.5. Betrakta även figur 3.1 och figur 4.1 tillsammans.

4.2 SocketActionLoader

Objekt av klassen **SocketActionLoader** fungerar som mottagare för nedladdning av kod över ett nätverk. Klassen är nära kopplad till sockethanteringen i realtidkärnan. Den använder statiska attribut i klassen **SocketLoadServer**. **SocketLoadServer** och underliggande kod finns i ett tillämpningsprogram som inte ingår i realtidskärnan.

```
class SocketActionLoader : public ActionLoader {
public:
  SocketActionLoader( char* socketname );
private:
  long m_packets;
  long m_lastpacket;
  void Handler( SocketLoadServer::Loader l );
```

```
    void GetHeader( SocketLoadServer::Loader l );
    void GetData( SocketLoadServer::Loader l );
    static void context2this( SocketLoadServer::Loader l,
      void* context );
};
```

Publika attribut: Konstruktorn (den funktion med samma namn som klassen och som initierar ett objekt i C++) **SocketActionLoader** skapar en socket med ett namn associerat till den. Denna socket kan man sedan skicka data till över nätverket.

Privata attribut: Variabeln **m_packets** räknar antalet mottagna paket för den action som håller på att laddas för tillfället. Variabeln **m_lastpacket** sätts när huvudet laddas och anger sista paketet för aktuell action. Funktionen **Handler** anropas från socketloadservern, dvs det är en s.k. callbackfunktion. Tyvärr kan inte ett objekts attribut vara callbackfunktion, varför en speciallösning har gjorts. Den verkliga callbackfunktionen är istället **context2this** som är en statisk funktion (dvs den tillhör inget objekt, utan är endast en del av klassen). Denna kan sedan anropa objektets **Handler**. Den underliggande koden i socketloadservern är komplicerad. Kort kan principen beskrivas som att en eller flera mottagarprocesser skapas. Dessa lyssnar på nätet och om det kommer något anropas en callbackfunktion.

Vidare finns funktionen **GetHeader** som anropas från handlern. Denna tar emot information om den action som kommer direkt efter, bland annat information om storleken, ID, namn och antal paket. Den skapar ett **Action**objekt med minne reserverat för nedladdning av kod. Funktionen **GetData** tar emot ett paket eller en del av ett paket och placerar den i det aktuella **Action**objektets allokerade minne.

4.3 Översikt värddatorn

På värddatorns sida måste det finnas något som förbereder koden för nedladdning. Detta är uppdelat i tre steg.

1. Koden korskompileras och länkas.
2. Den exekverbara koden omvandlas till en ström av data. I denna dataström finns information om namn, antal paket, längder etc.
3. Dataströmmen skickas via nätverket till målsystemet.

4.4 Kompilering och länkning

Det här examensarbetet bygger på att man kan generera PC-relativ, positionsoberoende kod. Detta visade sig svårare än väntat. Med det system som presenteras här har man stora begränsningar. De är:

- En action får endast innehålla *en* funktion. Några lokala funktioner får inte finnas.
- En action får inte innehålla några statiska eller globala variabler.
- C++ runtimesystemet med bl.a. operatorerna new och delete kan inte användas.

Detta begränsar en action till en funktion.

Kompilering och länkning till 680x0 processorn

Den kompilator som används är GNU-C kompilator för 680x0. Till denna används C-Front som kompilerar C++ till C. De kompilatoroptioner som anger att den genererade koden ska vara positionsoberoende fungerar endast delvis. Om man använder globala variabler, funktioner etc. och andra konstruktioner som länkaren ska lösa upp adressreferenserna till, blir det fel redan vid kompilering.

Länkaren klarar däremot att generera positionsoberoende kod. Detta under förutsättning att indata till den inte innehåller fasta adresser. Som inparameter till länkaren är det viktigt att ange entry-punkt, dvs namnet på actionfunktionen.

Kompilering och länkning till signalprocessorer

På signalprocessorerna, DSP32C, finns samma problem. Skillnaden är att kompilatorn inte utger sig för att kunna generera positionsoberoende kod. Koden blir trots allt ganska positionsoberoende, endast globala och statiska variabler samt funktionsanrop är positionsberoende.

4.5 Paketering

Omvandling och sändning behandlar bara kommunikationen värddator till 68040. På värddatorn ligger den kompilerade koden lagrad i en fil i **coff**-format, C Object File Format. Innan den skickas ned till målsystemet måste den avkodas och omvandlas till en dataström. Under denna omvandling får man ut information om bland annat storlek och entry-punkt. Vidare kan man räkna hur många paket som ska skickas. Ett problem är att en del av informationen inte finns tillgänglig förrän hela filen har lästs. Därför har jag valt att göra omvandlingen i två pass. I pass ett hämtas informationen. Därefter kommer informationen ut och efter den, genom pass två, datastömmen. Programmet (som exekveras på värddatorn) som gör omvandlingen heter **coff2action**.

coff2action

På institutionen finns ett klassbibliotek för att läsa **coff**-formatet. Detta har använts. Vidare används ett enkelt protokoll för att skicka data. Det ser ut som nedan.

Namn	*Storlek*	*Värde*
taglength	1	(=1)
tag	1	'h' = *huvud*, 'd' = *data*
endmark	1	(=0)
datalength	1	(count+4)
start	4	startadress
data	count	rå dataström
length	1	(count)
special	count	varierande datatyp
endmark	1	(=0)

Programmet skickar först en huvud. Detta innehåller följande:

Info	*Storlek i bytes*	*Kommentar*
1	1	taglength
'h'	1	nu kommer ett huvud
0	1	endmark
4	1	length
ID	4	special, ID använd som nyckel
32	1	length
namn	32	special, namn
4	1	length
Start	4	special, virtuella startadressen
4	1	length
Längd	4	special, kodens längd
4	1	length
Entry	4	special, kodens anropsadress
4	1	length
Paket	4	special, antalet paket som kommer att sändas
0	1	endmark

Därefter kommer data i paket. Varje paket ser ut som följande:

Värde	*Storlek i bytes*	*Kommentar*
1	1	taglength
'd'	1	nu kommer ett datapaket
0	1	endmark
4	1	length
ID	4	special, ID använd som nyckel
4	1	length
Paket	4	special, paketnummer
4	1	length
Start	4	special, startadressen för detta paket
datalängd	1	längd
data	datalängd	kod
datalängd	1	längd
data	datalängd	kod
...		
0	1	endmark

Programmet skriver till **stdout**. I allmänhet skickar man detta direkt till sändning.

4.6 Sändning

På värddatorn kan man öppna en **pappipe** som är ett kommunikationsprogram utvecklat vid institutionen. Det man skriver på **stdin** skickar detta program ut på nätet. Som argument till **pappipe** anges adressen, dvs namnet på den socket (se avsnitt 4.2) som skall ta emot denna action.

5. Objektorienterat gränssnitt mot realtidskärnan

5.1 Översikt

Vid institutionen finns en realtidskärna skriven i Modula-2. Denna kärna kompileras till C för att sedan kompileras till exekverbar kod. Tidigare utvecklade realtidsapplikationer i C++ använder funktionsgränssnittet i mellankoden, dvs den automatgenererade C-koden. Detta gränsnitt känns ofta stökigt. Anledningarna till det är flera. Bland annat använder Modula-2 till C-översättaren typer som överensstämmer med Modula-2. Detta tvingar programmeraren att göra manuella typkonverteringar (type casts) vid funktionsanrop. Förutom att detta försämrar säkerheten (lätt att göra fel) så gör det programkoden både ful och svårläst. I kärnan finns dessutom flera typiska objekt. Jag har därför skapat C++ objekt för att kapsla en del av de funktioner jag har använt. Observera att detta inte är ett komplett C++-interface till kärnan utan endast de grundläggande funktionerna. Detta kan däremot vara ett bra utgångsmaterial för en mer komplett C++-anpassning av RT-kärnan.

Målsättningen har hela tiden varit att inte försämra prestanda. Både hastigheten vid exekvering och minneskrav är viktiga aspekter. Detta har medfört att nästan alla funktioner är inline-deklarerade, dvs kompilatorn ansvarar för att optimera bort anropet och koden i funktionen hamnar på anropets plats. Detta kräver i sin tur att RT-kärnans deklarationer finns hela tiden. Användaren av RT-klasserna måste ha disciplin nog att inte använda C-gränssnittet.

5.2 Semaphore

Objekt av klassen **Semaphore** är semaforer för att synkronisera processer. Klassen kapslar alla semaforfunktionerna i RT-kärnan.

```
class Semaphore {
public:
  inline Semaphore( int initialvalue, char *name );
  inline ~Semaphore();
  inline void Wait();
  inline void Signal();
private:
  Semaphores_Semaphore m_sem;
};
```

Publika attribut: Eftersom semaforer är objekt behöver man inte initialisera dem explicit utan det görs i konstruktorn. Vidare ansvarar destruktorn för att ta bort semaforen. Funktionerna **Wait** och **Signal** gör wait respektive signal på semaforen.

Privata attribut: Variabeln m_sem är den variabel som svarar mot semaforen i C och Modula-2.

Exempel

Nedan följer ett exempel på användningen av klassen **Semaphore**.

```
Semaphore sem( 1, "Testsemafor" );
void process1()
{
  ...
  sem.Wait();
}
void process2()
{
  sem.Signal();
  ...
}
```

I exemplet ovan är semaforen en statiskt allokerad variabel så den skapas innan programmet har startat och försvinner inte förrän programmet har slutat. Man behöver inte komma ihåg att initialisera den eller ta bort den. Inte heller finns det någon risk att man gör det två gånger.

5.3 Monitor

Objekt av klassen **Monitor** är monitorer för att synkronisera processer och skydda variabler med ömsesidig uteslutning. Klassen kapslar endast en begränsad delmängd av monitorfunktionerna i RT-kärnan på grund av tidsbrist. Events finns *inte* implementerat.

```
class Monitor {
public:
  inline Monitor( char *name );
  inline ~Monitor();
  inline void Enter();
  inline void Leave();
private:
  Monitors_Monitor m_mon;
};
```

Publika attribut: Eftersom monitorer också är objekt behöver man inte heller initialisera dem explicit. Med funktionerna **Enter** och **Leave** går man in respektive ut ur ett monitorskyddat område. Observera att programmeraren fortfarande explicit måste ange detta, dvs man kan inte ärva in monitoregenskaper som i [1].

Privata attribut: Variabeln **m_mon** är monitorn.

Exempel

Nedan följer ett exempel på användningen av klassen **Monitor**. Först klassdeklarationen i en headerfil.

```
class Test {
public:
  Test( int start );
```

```
  void SetVar( int var );
  int GetVar();
private:
  Monitor mon;
  int m_var;
};
```

Sedan implementationen i en c-fil.

```
Test::Test( int start )
  : mon( "Testmonitor" )
{
  m_var = start;
}
void Test::SetVar( int var )
{
  mon.Enter();
  m_var = var;
  mon.Leave();
}
int Test::GetVar()
{
  int ret;
  mon.Enter();
  ret = m_var;
  mon.Leave();
  return ret;
}
```

I exemplet ovan skyddar monitorn variabeln **m_var**.

5.4 Mailbox

Objekt av klassen **Mailbox** är brevlådor för att synkronisera processer och skicka meddelanden mellan dem. Klassen kapslar alla mailboxfunktionerna i RT-kärnan.

I Modula-2 kan man, till skillnad från i C, i ett underprogram ta reda på en vektorparameters storlek. Där detta görs lägger Modula-2 till C översättaren med en extra parameter som anger indexets gräns. Detta gör att man vid implementationen av en **Mailbox**klass måste komma ihåg en storleksvariabel för att användas vid efterföljande anrop till C-koden. Observera att den verkliga storleken på aktuellt meddelande inte är känd.

```
class Mailbox {
public:
  inline Mailbox( int messagecount, int messagesize,
    char* name );
  inline ~Mailbox();
  inline void Transmit( void* message );
  inline int TryTransmit( void* message );
  inline void Receive( void* message );
  inline int TryReceive( void* message );
private:
```

```
  Mailboxes_Mailbox m_box;
  int m_messagesize;
};
```

Publika attribut: Funktionen **Transmit** postar ett meddelande. Om brevlådan är full blockeras den anropande processen. Funktionen **TryTransmit** försöker posta ett meddelande. Om brevlådan är full ger den noll tillbaka. Funktionen **Receive** tar emot ett meddelande. Om brevlådan är tom blockeras den anropande processen. Funktionen **TryReceive** försöker ta emot ett meddelande från brevlådan. Om brevlådan är tom ger den noll tillbaka.

Privata attribut: Variabeln **m_box** är här motsvarande brevlåda på C-nivå. Variabeln **m_messagesize** är en intern variabel för att hålla reda på storleken på ett meddelande mellan anropen till C-koden.

Exempel

Nedan följer ett exempel på användningen av klassen **Mailbox**.

```
class Point {
public:
  int x;
  int y;
};
Mailbox mbox( 100, sizeof(Point*), "Point mailbox" );
void process1()
{
  Point* p;
  for (;;) {
    ...
    p = new Point;
    GetMouseX( &p->x );
    GetMouseY( &p->y );
    mbox.Transmit( &p );
  }
}
void process2()
{
  Point* p;
  for (;;) {
    mbox.Receive( &p );
    ...
    delete p;
  }
}
```

I exemplet ovan skickar process1 muskoordinater asynkront till process2.

5.5 Time

Objekt av klassen **Time** har två olika uppgifter. Den ena är att synkronisera processer med hjälp av waituntil. Den andra är att ge användaren verktyg för tidmätning. Klassen kapslar alla tidsfunktionerna i RT-kärnan och ger dessutom några till.

```
class Time {
public:
  inline Time();
  inline void Update();
  inline void Inc( int ms );
  inline void WaitUntil();
  inline void ResetNow();
  inline int GetCurrent();
  inline static void WaitTime( int ms );
private:
  Kernel_Time m_time;
  int m_t0;
};
```

Publika attribut: När tidsobjektet skapas hämtas aktuell tid in. Objektets tidmätare sätts till noll. Funktionen **Update** laddar återigen in tiden, men objektets tidmätare sätts *inte* till noll. Funktionen **Inc** ställer fram objektets tid ett antal millisekunder i framtiden men påverkar inte tidmätningen. Funktionen **WaitUntil** blockerar anropande process tills den verkliga tiden är större än eller lika med objektets tid. Funktionen **ResetNow** sätter tidmätaren till noll. Funktionen **GetCurrent** ger tidmätaren i millisekunder. Funktionen **WaitTime** är inte ett egenligt attribut till objekt av klassen **Time** utan kan anropas utan objekt. Dess verkan är att fördröja anropande process ett antal millisekunder.

Privata attribut: Variabeln **m_time** är tidsvaribeln på C-nivå. Variabeln **m_t0** är tidmätaren.

Exempel

Nedan följer ett exempel på användningen av klassen **Time**.

```
void process()
{
  Time::WaitTime( 2000 );
  Time time;
  int t;
  for (;;) {
    time.Inc( 100 );
    time.WaitUntil();
    t = time.GetCurrent();
    ...
  }
}
```

I exemplet ovan väntar först processen två sekunder. Därefter går den in i en loop som exekveras var 100:e millisekund. Variabeln t ges värdet av tidmätaren och kan t.ex. skickas till plottning.

5.6 RTProcess

Klassen **RTProcess** är en basklass. Syftet med klassen är att kapsla processer i objekt samt att kunna skicka parametrar till en process. Det senare är inte

möjligt med Modula-2 RT-kärnan som används. Principen är ganska enkel. I klassen **RTProcess** finns funktionalitet för att starta en ny process. När ett nytt objekt skapas av en härledd klass börjar man att sätta dess attribut (det kan göras med parametrar). Därefter anropas en funktion i klassen **RTProcess** som startar den nya processen.

Uppstarten av en ny process *måste* vara en tvåstegsoperation, huvudsakligen på grund av basklassens konstruktor anropas före den härledda klassens. Om man försökter starta processen i basklassens konstruktor skulle processen rulla igång innan attributen var tilldelade några värden. Om man istället startar den i den härledda klassens konstruktor går det bra men man förhindrar vidare ärvning. Därför bör man starta processer i två programsteg. I denna klass finns inbyggt skydd mot att starta processer två gånger. Däremot måste man själv komma ihåg att starta den. Med den implementation som gjorts kan man ärva i flera nivåer utan problem. För utförligare information om detta problem se [1].

```
class ProcessBase {
public:
  inline ProcessBase();
  inline ~ProcessBase();
  void start( char *name );
protected:
  virtual void Process() = 0;
  inline void SetPriority( int prio );
private:
  static Semaphore sm_sem_startup;
  static ProcessBase* sm_process_this;
  int m_process_state;
  void SignalAndRun();
  static void StartUpProcess();
};
```

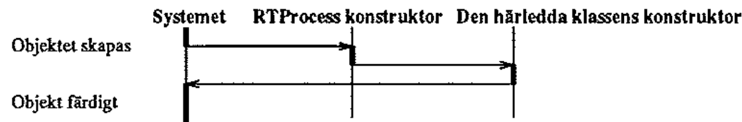
Publika attribut: Destruktorn ger ett felmeddelande om processen fortfarande är igång. Den kan tyvärr inte förhindra att objektet tas bort eller avsluta processen. Detta ger exekveringsfel. Funktionen **start** är den funktion som ska anropas när objektet är skapat och processen ska sättas igång.

Skyddade attribut: Den virtuella funktionen **Process** är den funktion som anropas i den nya processen. I normala fall går den i en loop tills man vill att processen skall avslutas. Funktionen **SetPriority** ställer processens prioritet till angivet värde.

Privata attribut: Variabeln **sm_sem_startup** används för att skydda statiska variabler under uppstarten. För alla klasser som härleds ur denna klass och för alla objekt finns endast en uppsättning statiska variabler. Den statiska variabeln **sm_process_this** används i uppstarten för att hålla reda på vilket objekt som håller på att startas. Variabeln **m_process_state** håller reda på processens tillstånd, dvs om den är ostartad, körande eller avslutad. Funktionen **SignalAndRun** används under uppstarten. Den statiska funktionen **StartUpProcess** är den nya processens startadress. Den i sin tur använder **sm_process_this** för att anropa rätt objekt.

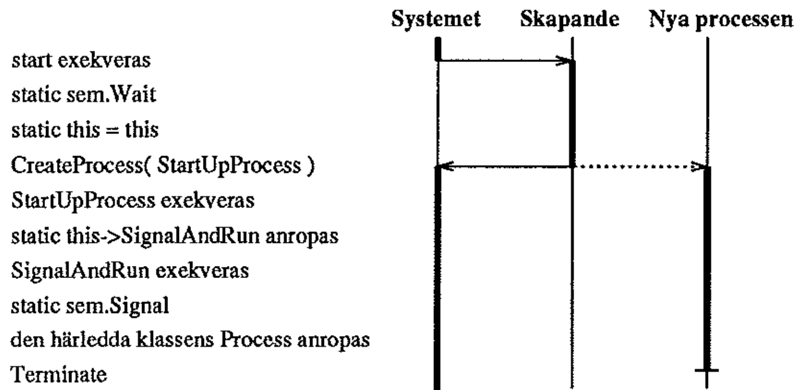
Interaktionsdiagram

En ny process skapas i två steg. I första steget skapas objektet. Se figur 5.1.



Figur 5.1 Interaktionsdiagram för första steget.

I andra steget sker en del synkronisering med andra processer som eventuellt samtidigt vill starta processer. Man måste åstadkomma ömsesidig uteslutning under delar av uppstarten. Därefter startas den nya processen och går asynkront med den gamla. Se figur 5.2. Om man följer sekvensen kan man se att semaforen skyddar den statiska this-pekaren. För att komma åt this-pekaren måste en annan process köra samma sekvens och den är utestängd från området mellan semaforens **Wait** och **Signal** då det är upptaget.



Figur 5.2 Interaktionsdiagram för andra steget.

Exempel

Nedan följer ett exempel på användningen av klassen **RTProcess**.

```
class Proc : public ProcessBase {
private:
  int m_var;
  void Process();
public:
  Proc( int var );
};
Proc::Proc( int var )
{
  m_var = var;
}
void Proc::Process()
{
  int x;
  for (;;) {
    ...
    x = m_var;
  }
}
main()
```

```
{
  Proc* proc = new Proc( 23 );
  proc->start();
  ...
}
```

I exemplet ovan skapas en process. Först skapas ett processobjekt. Därefter startas processen. Klassen **Proc** har ett attribut som sätts när objektet skapas. Den nya processen kan sedan använda detta.

5.7 MatCommSocket

Objekt av klassen **MatCommSocket** sköter grundläggande kommunikation med matlab.

```
class MatCommSocket {
public:
  inline MatCommSocket( char* name,
    int infinitereopen = 1 );
  inline ~MatCommSocket();
  inline void Send( int nrows, int ncols,
    MatComm_DataType dtype, void* data,
    int sizeinbytes );
  inline int GetNextType( int* nrows, int* ncols,
    MatComm_DataType* dtype );
  inline int Receive( int* nrows, int* ncols,
    MatComm_DataType* dtype,
    void* data, int maxsize );
private:
  MatComm_Socket m_socket;
  char* m_name;
  int m_infinitereopen;
};
```

Publika attribut: Funktionerna har samma funktion som de i RT-kärnan, men de är typsäkra. Dessutom ser man en socket som ett objekt. Vidare finns det viss felhantering inbyggd som försöker öppna en socket igen om förbindelsen har brutits.

Privata attribut: Variabeln **m_socket** är matcommporten. Variabeln **m_name** är dess namn och det används vid ethernetkommunikationen. Variabeln **m_infinitereopen** beskriver hur objektet ska bete sig om förbindelsen bryts.

5.8 SocketLoadServer

Klassen **SocketLoadServer** kapslar in C-funktioner för att ladda kod. Detta är ett statiskt objekt så att samtliga funktioner kan anropas direkt (utan föregående instantiering) som **SocketLoadServer::funktion**.

```
class SocketLoadServer {
public:
```

```
  typedef LoadServer_Loader Loader;
  typedef void (*Handler)( Loader, void* );
  inline static void Create( Handler h, char* kind,
    void* context );
  inline static int GetCommand( Loader l, char* c );
  inline static int GetData( Loader l, void* d,
    long maxsize );
protected:
  inline static int GetDataRange( Loader l, void* d,
    long first, long last );
};
```

Publika attribut: Funktionerna har samma innebörd som C-funktionerna, men de är typsäkra.

6. Vidareutveckling av actions

Under arbetets gång har en del problem uppstått och lösts. Tyvärr finns det olösta problem kvar på grund av att kompilatorerna är otillräckliga. Vid institutionen har en lösning på dessa problem provats. Den presenteras kort här. I målsystemet finns information om adresser och i värddatorn sker länkningen. Händelsekedjan vid nedladdning ser ut ungefär så här:

1. När en action ska laddas ned meddelar värddatorn målsystemet den kompilerade kodens storlek.
2. Målsystemet allokerar minnesutrymme och svarar med adress. Vidare sänder den över en symboltabell.
3. Värddatorn länkar den kompilerade koden till adressen och använder symboltabellen för att lösa upp adressreferenser.
4. Den länkade koden sänds över till målsystemet.
5. Målsystemet börjar med att köra statiska objekts kontruktorer. Sedan anropas en installationsfunktion. Installationsfunktionen är ansvarig för att koden blir inkopplad och blir inkopplad på rätt ställe. Den kan vidare skapa dynamiskt allokerade objekt etc.
6. När den kompilerade koden inte ska finnas längre exekveras en avinstallationsfunktion.

Genom att målsystemet tillhandahåller symboltabellen med de symboler en action får använda erhålls fortfarande kontroll över vad en action kan göra. Detta skiljer metoden från vanlig dynamisk länkning. Preliminära tester har visat att metoden fungerar bra. För vissa maskinvaror/kompilatorer kan dock den tidigare metoden fungera minst lika bra. Båda metoderna kan därför vara användbara, t.ex. för olika CPU:er i ett och samma styrsystem.

7. Kommentarer

7.1 Erfarenheter

Det allvarligaste problemet som har dykt upp är kompilatorernas oförmåga att generera positionsoberoende kod. Detta gäller för båda målsystemen i experimentplattformen. Detta ställer följande (delvis orimliga) krav på en action:

- Den får endast innehålla *en* funktion.
- Den får inte innehålla några statiska eller globala variabler.
- C++ runtimesystemet med bl.a. operatorerna new och delete kan inte användas.

Detta begränsar i princip en action till en funktion. Vidare krävs att standardfunktioner (t.ex. sqrt) ligger i omgivningsstrukturen. Detta ger upphov till en extra indirekt adressering vid funktionsanrop. Preliminära tester visar dock att alla dessa problem går att lösa såsom beskrevs i kapitel 6.

Att implementera actions kan delas upp i två problem. Det första är att implementera en hårdvaruoberoende struktur för att ta emot actions. Det andra problemet är att prova actions i ett verkligt system. För att lösa det andra problemet behöver man ha en god förståelse för systemet. Då utvecklingsmiljön och experimentplattformen är komplexa system tar det tid och är svårt att sätta sig in i dem. Detta gör tröskeln hög för att implementera actions trots att den grundläggande implementationen endast är medelsvår.

För att implementera actions behöver man ett maskinnära språk, huvudsakligen på grund av att man behandlar programkod som data. För att samtidigt kunna göra en struktur som enkelt kan återanvändas passar C++ utmärkt. Om Modula-2 hade använts istället hade det troligen blivit mera komplicerat att använda den färdiga implementationen av actions.

Övriga tänkvärda saker är bland annat att det finns realtidsaspekter att ta hänsyn till. Om en action har startat en process kan problem uppstå då actionfunktionen ska tas bort. Här finns ingen skyddsmekanism. Ett likartat problem uppstår då en action exekveras samtidigt som den ska tas bort. Att ha tillgång till en egenutvecklad realtidskärna underlättar därför att man då har full insyn.

Annat man behöver tänka på då man använder actions är det minne en action kan allokera. När actionfunktionen ska tas bort måste den meddelas om detta för att kunna frigöra minnet. Detta måste åstadkommas från applikationen med hjälp av flaggor i omgivningsstrukturen.

7.2 Slutsatser

Med traditionella metoder kan man inte tillhandahålla en riktigt generell regulator för inbyggnad i ett system. Exempelvis kan man inte göra en generell regulator på tillståndsform där parametrarna beror på andra tillstånd. Den traditionella typen av regulatorer kan man inte installera fast i ett system och låta parameterstrukturen bero på användningsområdet. Detta examensarbete visar att det är möjligt att åstadkomma generella regulatorer även för denna typ av problem.

I kapitel 2, avsnitt 2.3, presenterades tre exempel. Det första exemplet var en generell P-regulator. Med detta examensarbete kan generaliteten utökas så att regulatorn blir olinjär. Se appendix B. Det andra exemplet behandlade triggfunktioner. Med modifierbara triggfunktioner kan loggningen bli väsentligt mera generell. Det tredje exemplet kan, med examensarbetets resultat applicerade, ge en generell regulator på tillståndsform. Denna kan byggas in i ett system. Elementen i matrisen kan ändras till funktioner vilkas värden kan bero på godtyckliga tillstånd i regulatorn.

Styrkt av testexemplen ser man att den implementerade metoden verkligen kan ge flexiblare inbyggda system. Man ser att man kan modifiera fast implementerade algoritmer utifrån tillämpningsspecifika krav, utan att ändra i den fasta delen.

Referenser

[1] Peter A. Buhr, Richard A. Stroobossher, *μC++ Annotated Reference Manual*, Version 3.4.3, 1992

[2] Sean M. Dorward, Ravi Sethi, Jonathan E. Shopiro, *Adding New Code to a Running C++ Program*, USENIX C++ Conference Proceedings, San Francisco, California 1990

[3] Lars Nielsen, *Computer Implementation of Control Systems*, Dept. of Automatic Control, Lund Institute of Technology, December 1992

[4] Klas Nilsson, *Application Oriented Programming and Control of Industrial Robots*, Dept. of Automatic Control, Lund Institute of Technology, Second edition, 1992

[5] Klas Nilsson, *Object Oriented DSP Programming*, International Conference on Signal Processing Applications and Technology, Santa Clara, California 1993

[6] Karl J. Åström, Björn Wittenmark, *Computer Controlled Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1984

A. Programkod

Nedan följer vissa centrala delar av källkoden. Målet är att visa hur klassen **ActionFloatParameter** är implementerad. **ActionFloatParameter** är en inkopplingspunkt. Objekt av denna klass används som konstanter, men när en action blivit kopplad till objektet är det egentligen en funktion som anropas.

```
class Action {
  // This class is the container for executable code. It is also
  // responsible for executing it.
  ...
public:
  typedef float floatAF( void*, void** );
  inline float ReturnFloat( void* env );
  // Executes the action returning a float.
  ...
protected:
  char* m_block;
  // Pointer to the memory allocated containing the executable code.
  ...
};

inline float Action::ReturnFloat( void* env ) {
  Action::floatAF *action;
  action = (Action::floatAF *)(m_block + (m_virtualentry - m_virtualstart) );
  return ( action( env, &m_data ) );
}
...

class ActionSlot {
  // This class is a container for an action. This class is used
  // as baseclass when an object is to be able to receive downloded code.
public:
  inline ActionSlot();
  // Default contructor. No action associated to the object.
  inline void SetAction( Action* action );
  // Associate action with this object.
protected:
  Action* m_action;
  // Pointer to the associated action.
};

inline ActionSlot::ActionSlot() {
  m_action = 0;
}

inline void ActionSlot::SetAction( Action* action ) {
  if (m_action) {
    delete m_action;
  }
  m_action = action;
}

class ActionFloatParameter : public ActionSlot {
  // Constant parameters of float type that can be changed with a
  // action to a function should be objects of this type.
public:
  inline ActionFloatParameter( float initialvalue, void* env, char* name,
         ActionLoader* actionloader, int key );
```

```
  // Constructor. Until the action is loaded references to this
  // object returns initalvalue.
  inline operator float();
  // When this object is used the float() operator is used. This
  // returns the initialvalue if no action has been downloaded.
  // Otherwise it executes the actionfunction and returns the
  // returnvalue.
protected:
  float m_value;
  // The initialvalue is stored in this variable.
  void* m_env;
  // Pointer to the environment structure used in calls to the
  // actionfunction.
  char  m_name[32];
  // The name of the parameter is stored in this variable.
};

inline ActionFloatParameter::ActionFloatParameter( float initialvalue,
void* env, char* name, ActionLoader* actionloader, int key ) {
  m_value = initialvalue;
  m_env = env;
  strcpy( m_name, name );
  actionloader->RegisterSlot( this, key );
}

inline ActionFloatParameter::operator float() {
  if (m_action) {
    return ( m_action->ReturnFloat(m_env) );
  } else {
    return m_value;
  }
}
```

B. Tillämplingsexempel

Nedan följer centrala delar av källkoden för ett typexempel. Det som visas är en P-regulator vars konstant K kan ersättas med en funktion. Samtidigt visas hur klassen **RTProcess** används för att skapa processobjekt.

```
// Headerfile
typedef struct {
  float uc;
  float y;
} Controller_struct;
typedef struct {
  double (*sqrt_fp)( double );
} Math;
typedef struct tagRegulEnv {
  Controller_struct controller;
  Math              math;
} RegulEnv;

class Regulfloatpar : public ActionFloatParameter {
  // Actionparameters using the Regul environment should be
  // objects of this class rather than the base class. This is
  // due to type safety. The base class also does not know about
  // the specialized SocketActionLoader, only the ActionLoader.
public:
  Regulfloatpar( float initialvalue, RegulEnv* env, char* name,
  SocketActionLoader* actionloader, int key )
    : ActionFloatParameter( initialvalue, env, name, actionloader, key )
  {}
  // Constructor. This automaticly calls the ActionFloatParameter
  // contructor.
};

class Controller;
class Regul {
private:
  Monitor m_mon;
  // Monitor protecting the classmember.
  float m_ref;
  // Monitorprotected reference value.
public:
  RegulEnv m_regulenv;
  // Environmentpointer containing variables and pointers to functions
  // accesible from an action.
  Controller* m_cntr;
  // Pointer to an object that is a process.
  SocketActionLoader m_loader;
  // A Loader using the socket (ethernet) interface for downloading
  // actions.
  Regul();
  // Constructor.
  ...
};

class Controller : public ProcessBase {
  // This class is a process class. The constructor sets the member
  // variables. Then a function in the base class is called, start(),
  // spawn a new process. This separation in two function calls are
  // necessay if it should be possible to derive from a class
```

```
  // such as this one. The constructor of a base class is executed
  // before the derived class. For a more thourogh explanation of
  // this see [Peter A. Buhr, Richard A. Stroobosscher,
  // uC++ Annotated Reference Manual, Version 3.4.3, 1992, page 29].
private:
  Regul* m_regul;
  // Pointer to the regul-object.
  int m_priority;
  // Priority for the process.
  void Process();
  // The Process function that is executed at start().
  Regulfloatpar m_Kc;
  // Parameter constant at first, but an action can be downloaded in
  // its place.
public:
  Controller( Regul* regul, int priority, RegulEnv* regulenv,
     SocketActionLoader* loader );
  // Constructor using the parameters.
};

// Implementationfile
Controller::Controller( Regul* regul, int priority, RegulEnv* regulenv,
       SocketActionLoader* loader )
     : m_Kc( 2.0, regulenv, "Regul Kc", loader, 0 )
{
  m_regul = regul;
  m_priority = priority;
}
void Controller::Process()
{
  int sampleinterval = 200;
  float P;
  float u;
  float v;
  SetPriority( m_priority );
  Time time;
  for (;;) {
    time.Inc( sampleinterval );
    time.WaitUntil();
    m_regul->m_regulenv.controller.uc = m_regul->GetRef();
    m_regul->m_regulenv.controller.y = AnalogIO_ADIn( 1 );
    P = m_Kc * ( m_regul->m_regulenv.controller.uc -
m_regul->m_regulenv.controller.y );
    v = P;
    if ( v > 0.9 ) { u = 0.9; }
    else if ( v < -0.9 ) { u = -0.9; }
    else { u = v; }
    AnalogIO_DAOut( 0, (REAL)u );
  }
}
// The constructor for the Refgenclass calles the contructor for
// the SocketActionLoader with the name for the socket as parameter.
Regul::Regul()
     : m_mon( "Regul monitor" ),
     m_loader( "RegulSocket" )
{
  m_regulenv.math.sqrt_fp = sqrt;
  m_cntr = new Controller( this, 20, &m_regulenv, &m_loader );
  m_cntr->start( "Controller thread" );
}
...
```