Sammanfattning

Denna rapport är ett försök för att förklara hur man programmerar en Sony AIBO ERS-210 med OPEN-R. Till detta ändamål har mjukvaruresurserna från OPEN-R SDK använts. Rapporten beskriver roboten från tre olika aspekter, dvs. hårdvaran, mjukvaran och programmeringsprocessen. Olika steg vid mjukvaruutvecklingen tas upp och förklaras, nämligen till exempel: kompilering, körning och felsökning. Rapporten som en laborationshandledning ger konkreta lösningsförslag i form av ett antal väldefinierade steg till de vanligaste programmeringsproblemen som en OPEN-R programmerare kan stötta på.

Abstract

This rapport is an effort to create a laboratory manual for programming of a Sony AIBO ERS-210 with OPEN-R. For this purpose the software packages from the OPEN-R SDK have been used. We have tried to look at the robot from three different aspects such as the hardware, the software and the task of programming. Different steps have been described in the process of programming, ie compilation, execution and debugging. This report as a laboratory manual offers straightforward and concrete solutions to the most common programming problems that an OPEN□R programmer can be faced with.

Förord

Resan av tusen mil börjar alltid med ett litet steg. Det har varit en lång men intressant väg. Det har förstås funnits problem på vägen men det har varit mycket givande. Vi vill tacka speciell vår handledare Jacek Malec, som under tiden har väglett oss med stor intresse, arrangement, kompetens och tålamod. Vi skulle även få tacka ett antal studenter som har hjälpt oss med att komma igång med arbetet och förståelse om systemet. Tack till Daniel Påhlstorp, Dannie Ronnovius, David Pettersson och Michael Green.

Innehållsförteckning

Kapitel 1

Introduktion

Denna rapport är resultatet av ett 20-poängsexamensarbete i datalogi vid Institutionen för datavetenskap vid Lunds Universitet.

Förslaget till detta examensarbete kommer från docent Jacek Malec, ansvarig för kursen AI för robotar vid Lunds Universitet. Malec uttryckte behovet av en dokumentation som skulle fungera som en handledning för programmering av en AIBO ERS-210 med OPEN-R.

Grunden till detta arbete ligger i det faktum att det saknades en relativt kortfattat beskrivning om hur man kan programmera en AIBO ERS-210, omfattningen och utspridningen av de existerande materialen har gjord det svårt för de intresserade.

Denna rapport kan användas som en laborationshandledning till programmering av en AIBO ERS-210. I rapporten sammanställs och presenteras de mest väsentliga materialen och kunskaperna om programmering av en AIBO på ett sammanhängande sätt för att göra det lättare för läsaren att förstå programmeringsprocessen av en ERS-210. Rapporten kommer att ge även förklaringar till hur man kan kompilera, köra och söka fel i ett färdigt program.

Kapitel 2

Systembeskrivning

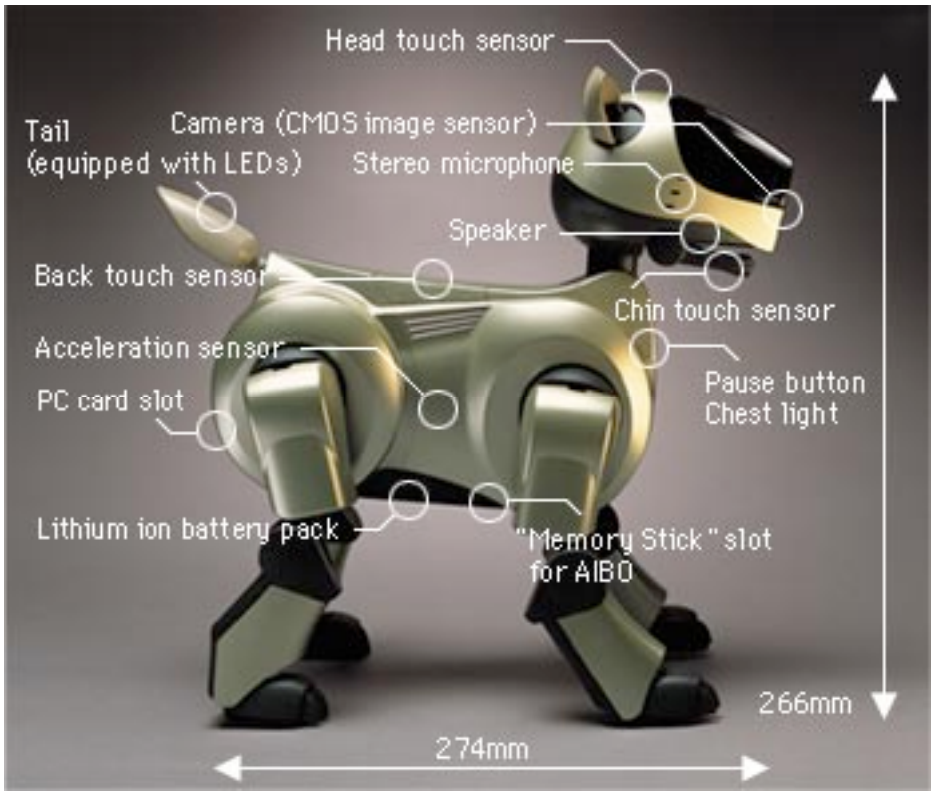
2.1 Hårdvaran

Hårdvaran hos en ERS-210 AIBO består av följande huvuddelar:

- Roboten ERS-210
- Datorn
- Minnesticksenheten
- Sladdlöskommunikation
- Avlusningslådan

2.1.1 Roboten ERS-210

ERS-210 är en robot utrustad med ett antal sensorer och effektorer. Roboten är visad på Figur 2.1.



Figur 2.1: ERS-210 AIBO.

Tabellen visar det mest väsentliga om hårdvaran hos en ERS-210.

Vikt	1,5 kg inklusive batteri och minnessticket
Dimensioner	Bredd: 152 mm Höjd: 266 mm Längd: 274 mm
Processorn	64-bitar RISC processor tillverkad av MIPS Klockfrekvens: 192 MHz Minne: 32 MB RAM
Komponenter	Kroppen, Huvudet, Svansen, 4 ben
Motorer (rörliga delar)	Munnen: 1 frihetsgrad (degree av freedom) Huvudet: 3 frihetsgrader Benen: 3 frihetsgrader (per ben) x 4 = 12 Öronen: 1 frihetsgrad (per öra) x 2 = 2 Svansen: 2 frihetsgrader Totalt = 20 frihetsgrader
Lysdioder	Ögonen: röd x 4, grön x 2 = 6 st Bröstet: röd x 1, grön x 1 = 2 st Svansen: blå x 1, orange x 1= 2 st Totalt = 10 st
Sensorer	CCD Kamera med 10^6 pixel i upplösning Mikrofon x 2 Trycksensorer: vid huvudet, hakan, ryggen och benen Accelerationssensor, Rörelsesensorer, Temperatursensor, Infraröd avståndssensorn och Vibrationssensorer
Temperaturbegränsning	5° C till 35° C (Celsius)
Fuktighetbegränsning	10% till 80%

Tabell 2.1: ERS-210 Specifikationer.

Varje motor rör ett led i endast en riktning. Flera motorers samverkan möjliggör komplexa rörelser. Till exempel de tre motorer vid nacken har till uppgift att få roboten att nicka, vrida och luta huvudet. Varje motor har en viss rörlighetsbegränsning dvs. har ett maximal och minimal vridningsgrad.

2.1.2 Datorn

Det minimala kravet på en persondator eller en arbetsstation, som används vid applikationsutveckling till en ERS-210 med OPEN-R SDK är:

- Processorn: minst 233MHz i klockfrekvens.
- Primärminne: minst 64 MB RAM.
- Hårddisk*:* minst 200 MB fritt utrymme.

Dessa är tillverkarens krav för användning av OPEN-R SDK. Man kan koppla en AIBO till en dator och kommunicera med den. Dessa minimala krav är avgörande vid sådana tillfället när man vill kommunicera eller göra felsökning via fysiska förbindelsen.

2.1.3 Minnesticksenheten

Minnesstick används för att flytta data till och från ERS-210. För att kunna göra detta så måste för det första datorn vara utrustad med minnesstickläsaren MSAC-USI och för det andra tillgång till ett minnesstick (MS). Minnessticken som får användas har beteckningen ERS-MS016, 16 MB och är rosa.

OBSERVERA ATT: Båda dessa enheter måste vara tillverkade av Sony, annars säkerhetsfunktionerna sätts igång och låser minnestickläsaren!
Anledningen att systemet låser sig är i själva verket närvaron av en osynlig fil i det rosa programmeringsminnessticket. Varje gång roboten bootas (startas), så finns det en säkerhetsrutin som kollar om det är rätt minnesstick som används. Om minnessticket är OK, så fortsätter systemet att köras, annars säkerhetsrutinen låser systemet och körningen misslyckas.

2.1.4 Sladdlöskommunikation

Vad är sladdlöskommunikation?

Sladdlöskommunikation eller Wireless LAN på engelska är ett lämpligt sätt att skapa en kommunikationskanal mellan olika aktörer i systemet. Den använder sig av radiovågor för att överföra data mellan aktörer. Det finns olika kopplingssätt i ett nätverk för sladdlöskommunikation.

Figur 2.2: Koppling1.

Koppling 1: En bärbar dator med sladdlöskort har kopplats till en kopplingspunkt (Access Point). En kopplingspunkt är i själva verket en radiosändare/mottagare. Genom kopplingspunkten etableras radiokontakten med roboten, som i sin tur måste innehålla en AIBO-sladdlöskommunikationskort (ERA-201D1).

Figur 2.3: Koppling 2.

Koppling 2: Koppling är mellan en stationär dator och en AIBO via en fysisk länk till en kopplingspunkt. Här behövs även en Hub för att slutföra kopplingen.

Figur 2.4: Koppling 3.

Koppling 3: Kopplingen åstadkoms med hjälp av en bärbardator och en AIBO, utan någon kopplingspunkt.

2.1.5 Avlusningslådan

Avlussningslådan är ett alternativ till sladdlöskommunikation som har till uppgift att ange vilket tillstånd datorn och roboten befinner sig i. Informationen från ERS-210 kan visas via en serie- och en parallellkoppling i realtid på ett terminalfönster. Vid systemavbrott kan roboten startas om med en *Reset*-knapp, placerad på avlusningslådan.

2.2 Mjukvara

2.2.1 Aperios

Aperios är Sonys egentillverkade operativsystem (OS), som är skapat specifikt för smarta inbyggda systemen. Aperios är kärnan (kernel) i ett realtidsoperativsystem.

Aperios har en storlek på 100 kilobytes (kb) vilket gör den mer lämpligt att användas i små digitala apparater som kräver ett operativsystem. Windows eller Linux med en storlek på flera miljoner bytes är mer lämpliga till större apparater, som stationära eller bärbara datorer.

Aperios är ett distribuerat OS, utan någon central kontrollenhet, vilket innebär att när Aperios bildar ett nätverk av digitala komponenter (noder), processkraften och minneskapaciteten hos hela nätet blir summan av de enskilda komponenters kraft och kapacitet.

Aperios är uppbyggt så att den känner av sitt nätverk och uppdaterar eller förnyar delar av OS så fort som det kommer nyare version genom olika komponenter som kopplas till nätverket. Detta är möjligt eftersom Aperios är byggt av ett antal moduler och saknar central kontrollenhet vilket gör att en modul kan ersättas av en nyare modul i nätverket utan några svårigheter.

Det finns tyvärr också en del brister i Aperios i jämförelse med andra konventionella OS som t.ex. Unix och Windows. Det finns t.ex. inga pipe, socket, signaler, fork och exec, som är vanliga i en Unix-kärna. Det finns även svagheter när det gäller filhantering, flertrådig system (multi-threading), parallellism och felsökning.

Det är viktigt att tänka på dessa svagheter och styrkor hos systemet när man programmerar en apparat som innehåller Aperios.

2.2.2 OPEN-R

OPEN-R eller ”OPEN aRchitecture” är Sonys standard programmeringsgränssnitt (Application Programming Interface API) för att kunna programmera inbyggda system. OPEN-R är en samling av biblioteksrutiner som optimerar kontrollen över och utnyttjandet av en AIBOs resurser. Dessa rutiner är representerade endast i form av biblioteksfunktionsanrop och detaljerna i koden är inte tillgängliga för användaren. OPEN-R fungerar även som en länk mellan System och Applikations lager.

Man kan säga att OPEN-R:

- förenklar utnyttjande och utvecklingen av mjukvaran.
- ger möjligheter till att modulisera mjukvaran.
- möjliggör modulisering av hårdvaran (reservdelar).
- skapar möjligheter till kommunikation mellan olika mjukvarumoduler.
- ger stöd för sladdlöskommunikation och nätverksprogrammering.

De mest väsentliga begreppen i OPEN-R är objekt och interobjektkommunikation (IOK).

- **Objekt i OPEN-R**

  En applikation i OPEN-R består av ett eller flera objekt. Varje objekt utför en specifik uppgift. Ett objekt i OPEN-R existerar endast vid exekvering (run-time) och liknar mer en process i Unix än objekt i C++ eller Java. Det innebär att ett OPEN-R objekt är ett självständigt och körbart program, dvs. har sin egen exekveringstråd och kan kommunicera med andra objekt i systemet.

  En typiskt livscykel för ett OPEN-R objekt är en oändlig loop:

  1. Objektet laddas och startas av systemet
  2. Objektet väntar på meddelanden
  3. När meddelande mottagits exekveras metoden som motsvarar begäran.
  4. När metoden upphör sin körning, så går objektet tillbaka till vänttillståndet. (punkt 2).

  OBSERVERA ATT: Ett objekt kan inte avsluta sig själv, utan det existerar så länge systemet är på gång. Ett program (applikation) kan avslutas bara genom att trycka på knappen som sitter på framsidan av roboten.

- ## Interobjektkommunikation

InterObjektKommunikation (IOK) är den viktigaste metoden att introducera kommunikation mellan objekten i en applikation. Genom att skicka och ta emot meddelande via en gemensam minnesplats, kan alla objekt i OPEN-R kommunicera med varandra. Varje objekt kan skapas separat och kopplas till andra objekt i applikationen med hjälp av IOK.

Alla OPEN-R objekt kan vara både sändare (Subject) och mottagare (Observer). IOK fungerar på följande sätt:

1. Sändaren skickar meddelandet till den gemensamma minnesplatsen med hjälp av `subject[index]->SetData()`. Sändaren talar om för alla mottagare att meddelandet är skickat genom att anropa `subject[index]->NotifyObservers()`.

2. Mottagaren tar emot meddelandet från den gemensamma minnesplatsen i form av en händelse `event` av typen `ONotifyEvent&`. Själva meddelandet packas upp med `event.Data(0)`. När meddelandet är mottaget meddelar mottagaren sin beredskap för att ta emot nästa meddelande genom att skicka `AssertReady()` till alla sändare.

3. Om Mottagare inte är redo, så skickar Sändaren inte något meddelande.

2.2.3 OPEN-R SDK

En samling av programutvecklingsverktyg, baserat på programmeringsspråket C++ och biblioteksrutiner i OPEN-R kallas för OPEN-R SDK. Det används som en plattform vid utveckling av program till Sonys AIBO robotar ERS-210, ERS-220, ERS-210A och ERS-220A. OPEN-R SDK förenklar programmeringsprocessen avsevärt genom att separera systemskiktet ifrån applikationen. Vid OPEN-R SDKs officiella hemsida finns ett antal programmeringsexempel, manualer, minnessticksavbilder och program tillgängliga. Enligt Sonys licensvillkoren får man inte utveckla program med OPEN-R SDK för kommersiella syften.
En detaljerad instruktion till installationen av OPEN-R SDK ges detaljerad i ”OPEN-R SDK Installation Guide”.

Kapitel 3

Programmering med OPEN-R

Programmering med OPEN-R innebär att man skapar ett antal självständiga objekt och inrättar kommunikationskanalerna mellan objekten.

3.1 Arkitekturen hos ett OPEN-R program

Ett OPEN-R program består av ett antal filer placerade i några kataloger.:

- Ett ram
- Minst en objektkatalog
- En MS katalog
- En Makefile

Ett ram

Hela applikationen måste ligga inuti en katalog som fungerar som ett ram, skal eller behållare för hela programmet. Detta förenklar arbetet med att söka efter eller att hänvisa till ett program. Denna katalog kallas också för ”applikationskatalogen”.

Varje ram innehåller:
- Minst en objektkatalog
- En MS katalog
- En Makefile (Ram-Makefile)

Objektkatalogen

Varje objekt inuti en applikation måste ha en egen objektkatalog. Antal objektkataloger är alltid lika med antal objekt i applikationen.

Varje objektkatalog innehåller 5 filer:
- En *.h-fil
- En *.cc-fil
- En *.ocf-fil
- En stub.cfg fil
- En Makefile (Objekt-Makefile)

MS katalogen

Katalogen ”MS” innehåller ett antal kataloger och filer som tillsammans med ett antal andra kataloger och filer från **/usr/local/OPEN_R_SDK/OPEN_R/MS** kopieras till minnesticket vid körningen. Dessa kataloger kallas för ”MS-avbilder” eller ”MS-konfigurationer”. MS-avbilder är tillgängliga i laborationsdatorer i katalogen /usr/local/AIBO/MS/.

Allt som en AIBO kommer slutligen att ha tillgång till vid körningen skall finnas med på minnessticket. Minnessticket är endast 16 MB stor och är alltså hela AIBOs hårddisk! I så litet utrymme skall det rymmas alla de körbara filerna (*.BIN), hela operativsystemet Aperios och direktiven om kommunikation mellan objekten!

Här presenteras endast den del av katalogen MS som skall finnas med i applikationen.

Katalogstrukturen:

MS/OPEN-R/MW/OBJS/
MS/OPEN-R/MW/CONF/OBJECT.CFG
MS/OPEN-R/MW/CONF/CONNECT.CFG

MS/OPEN-R/MW/OBJS/
I denna katalog samlas alla de körbara objektfilerna (dvs. *.BIN-filer). Detta skapas vid kompileringen.

MS/OPEN-R/MW/CONF/OBJECT.CFG
Adressen till de körbara filerna (*.BIN) som finns i minnessticket måste vara tillgänglig för resten av systemet vid körningen. Dessa adresser finns samlade här i filen OBJECT.CFG.

CFG står för ConFiGuration.

Ett exempel:
AIBO/sample/HelloWorld/MS/OPEN-R/MW/CONF/OBJECT.CFG:

```
/MS/OPEN-R/MW/OBJS/POWERMON.BIN
/MS/OPEN-R/MW/OBJS/HELLO.BIN
```

Detta betyder att det finns två objekt i applikationen, objektet ”POWERMON.BIN” och ”HELLO.BIN”.

OBSERVERA ATT: /MS/OPEN-R/MW/OBJS/POWERMON.BIN. refererar till ett objekt som är inte kopplat till något annat objekt i applikationen och arbetar helt separat. Det har till uppgift att stänga av AIBO på ett säkert sätt när strömknappen trycks ned. Utan det här objektet kan AIBO inte stängas av. Den kommer att befinna sig i ett mellanläge som liknar pausläget och lysdioden ovanför strömknappen förblir röd även efter nedtryckning.

MS/OPEN-R/MW/CONF/CONNECT.CFG
Varje rad i filen CONNECT.CFG beskriver en kommunikationkanal mellan sändare mottagaren.

En Makefile (Ram-Makefile)

En Makefile innehåller ett antal kommandon till det underliggande operativsystemet. Filen implementerar tre kommandon dvs. `all`, `install` och `clean` i tur och ordning för varje angiven objekt i applikationen. Variabeln COMPONENTS tilldelas objektens namn. Variabeln `INSTALLDIR` visar var *.BIN-filer skall hamna. Resten av filen kommer att se alltid exakt likadant ut, och garanterar att kommando make anropas med rätta parametrar för varje komponent i applikationen.

```
COMPONENTS= MittObjekt
INSTALLDIR=$(shell pwd)/MS
TARGETS=all install clean

.PHONY: $(TARGETS)

$(TARGETS):
for dir in $(COMPONENTS); do \
(cd $$dir && $(MAKE) INSTALLDIR=$(INSTALLDIR) $@) \
```

Ram-Makefile för en enkel applikation med ett objekt.

3.2 Objekt i OPEN-R

Mjukvarumodulerna i OPEN-R kallas för ”Objekt”. En applikation eller ett OPEN-R program består av minst ett objekt. Ett objekt i OPEN-R liknar mer en process i Unix än objekt i C++ eller Java. Detta innebär att ett OPEN-R objekt är ett självständigt och körbart program.

Varje objekt i OPEN-R:

1. motsvarar en körbar fil
2. har en egen katalog (objektkatalogen).
3. har sitt eget körningstråd (Thread).
4. har flera ingångspunkter (entry points).
5. kan kommunicera med sin omgivning och utbyta information.(IOK)
6. motsvarar en uppgift.

1. Ett objekt motsvarar en körbar fil.

Ett objekt i OPEN-R existerar som en enhet endast vid exekvering (run-time). Denna enhet representeras av en binär fil, dvs. en *.BIN fil. Innan dess, dvs. vid kompileringen består ett objekt av ett antal filer (5 st).

2. Ett objekt har en egen katalog (objektkatalogen).
Katalogen fungerar som en behållare. Den innehåller 5 filer:

a. Kärnklassfilerna (*.h + *.cc)

Med kärnfilerna menas två filer. En *.h-fil där man deklarerar sina metoder och variabler och en *.cc-fil, vilken innehåller definitionen av de deklarerade metoderna i objektet, enligt regler i språket C++.

b. en Makefile (Objekt-Makefile)

Direktiven till operativsystemet vid kompileringen och länkningen finns här. Det är Makefile som bestämmer vilka hjälpfiler skall skapas, flyttas och tas bort under kompileringsfasen. Det är Makefile som säger till stubgeneratorn vad den skall göra. Den här filen är en samling av adresser i form av konstanter och även kommandon till underliggande operativsystemet. Den skapar de körbara filerna dvs. *.BIN-filer och flyttar de till katalogen /MS (här: `HelloWorld/MS/OPEN-R/MW/OBJS/`). Användaren kan även ta bort alla de hjälpfiler som har skapats under kompileringen och länkningsprocessen med hjälp av kommandot: `make clean`.

Här visas Makefile i AIBO/sample/ERA201D1Info/ERA201D1Info:

```
OPENRSDK_ROOT?=/usr/local/OPEN_R_SDK
INSTALLDIR=../MS
CXX=$(OPENRSDK_ROOT)/bin/mipsel-linux-g++
STRIP=$(OPENRSDK_ROOT)/bin/mipsel-linux-strip
MKBIN=$(OPENRSDK_ROOT)/OPEN_R/bin/mkbin
STUBGEN=$(OPENRSDK_ROOT)/OPEN_R/bin/stubgen2
MKBINFLAGS=-p $(OPENRSDK_ROOT)
LIBS= \
     -L$(OPENRSDK_ROOT)/OPEN_R/lib \
     -lObjectComm \
     -lOPENR \
     -lInternet \
     -lantMCOOP \
     -lERA201D1
CXXFLAGS= \
     -O2 \
     -g \
     -I. \
     -I$(OPENRSDK_ROOT)/OPEN_R/include/R4000 \
     -I$(OPENRSDK_ROOT)/OPEN_R/include/MCOOP \
     -I$(OPENRSDK_ROOT)/OPEN_R/include

.PHONY: all install clean

all: era201d1Info.bin
```

```
%.o: %.cc
     $(CXX) $(CXXFLAGS) -o $@ -c $^

ERA201D1InfoStub.cc: stub.cfg
     $(STUBGEN) stub.cfg

era201d1Info.bin: ERA201D1InfoStub.o ERA201D1Info.o
     era201d1Info.ocf
     $(MKBIN) $(MKBINFLAGS) -o $@ $^ $(LIBS)
     $(STRIP) $@

install: era201d1Info.bin
     gzip -c era201d1Info.bin > $(INSTALLDIR)/OPEN-
     R/MW/OBJS/ERA201D1.BIN
clean:
     rm -f *.o *.bin *.elf *.snap.cc
     rm -f ERA201D1InfoStub.h ERA201D1InfoStub.cc def.h entry.h
     rm -f $(INSTALLDIR)/OPEN-R/MW/OBJS/ERA201D1.BIN
```

c. en *.ocf

”ocf” står för ”object configuration” och filen har till uppgift att konfigurera och skapa objektet i OPEN-R vid länkningen. Lägg märke till att filnamnet börjar alltid här med en liten bokstav. Filen innehåller endast en enda rad som består av 8 ord. De ord markerade med fetstil i tabellen kan man välja mellan för att fylla i raden i .ocf-filen.

Tabell .1. Innehållet i en *.ocf-fil

Ord 1	Ord 2	Ord 3	Ord 4	Ord 5	Ord 6	Ord 7	Ord 8
object	Objektnamn här **helloWorld**	Stackstorlek i byte,t.ex **3072**	Heapstorlek, t.ex **16384**	Prioritet **0** till **128**	Cache **cache** eller **nocache**	Minne **tlb** eller **notlb**	Mode **user** eller **kernel**

Exempel: ”/AIBO/sample/HelloWorld/HelloWorld/helloWorld.ocf”:

```
object helloWorld 3072 16384 128 cache tlb user
```

För mera information om detaljerna om vad varje fält betyder se: AIBO/manualer/Programmersguide_E.pdf”->kapitel3.1->.ocf sida 17.

d. en stub.cfg

En `stub.cfg` fil beskriver antal och typ av kommunikationskanaler hos ett objekt. Detta kommer att beskrivas i detalj i avsnitt om interobjektkommunikation.

3. **Ett objekt har sin egen körningstråd.**
   Detta gör att ett objekt kan köras samtidigt (i realtid) med andra objekt. Å andra sidan ett objekt kan köras bara i en tråd. Det finns inga möjligheter att fördela objektens beräkningar i flera trådar.

4. **Ett objekt har flera ingångspunkter (entry points).**
   Till skillnad från de flesta andra konventionella programmeringsspråk som tillåter endast en enda ingångspunkt i varje program (ofta metoden `main()`), så finns det flera ingångspunkter i ett OPEN-R objekt. Det finns åtminstone 5 ingångspunkter till varje objekt. De motsvarar metoderna `DoInit()`, `DoStart()`, `DoStop()`, `DoDestroy()`, och `Prologue()`. De fyra första metoderna kallas för "**Do-metoderna**" och måste finnas med i kärnklassen till alla OPEN-R objekt. Anledningen är att alla objekt i OPEN-R ärver från klassen OObject. In i klassen Oobjekt är Do-metoderna deklarerade som abstrakta och anropar "**de riktiga ingångspunkterna**", dvs. metoderna: `Init()` `Start()`, `Stop()`, `Destroy()`.

   `DoInit()`
   Anropas först i början av programmet. Den registrerar kommunikationstjänsterna. Mer om detta i avsnittet om IOK.

   `DoStart()`
   Anropas efter `DoInit()` och innehåller koden som beskriver vad objektet skall utföra vid körningen.

   `DoStop()`
   Anropas vid avslutning av programmet.

   `DoDestroy()`
   Anropas strax efter `DoStop()` innan programmet skall upphöra sin körning helt.

   `Prologue()`
   En speciell ingångspunkt som inte befinner sig i kärnklassen. Den anropas endast en gång vid laddningen av varje objekt. Den initierar variabler och anropar konstruktörer till globala attribut.

   OBSERVERA ATT: Ett objekt kan inte avsluta sig själv, utan det existerar så länge systemet är på gång. Ett program avslutas genom att stänga roboten, dvs. trycka på knappen som sitter på framsidan av roboten.

5. **Ett objekt kan kommunicera med sin omgivning och utbyta information.**
   Mer om detta vid avsnitt 3.3 nedan.

3.3 Interobjektkommunikation

Interobjektkommunikation eller "IOK" är en av de viktigaste och mest användbara principerna i programmering med OPEN-R. Det sker både i form av kommunikation mellan

egenskrivna objekt och även med och mellan systemobjekt. IOK påminner mycket om klient/server principen, dvs. att tjänster erbjuds på efterfrågan.

Alla objekt kan skicka så väl som ta emot meddelanden, dvs. kan vara både sändare och mottagare, men inte i samma ögonblick! Sändarobjektet i OPEN-R kallas för ”Subject” medan mottagarobjekten kallas för ”Observer”.

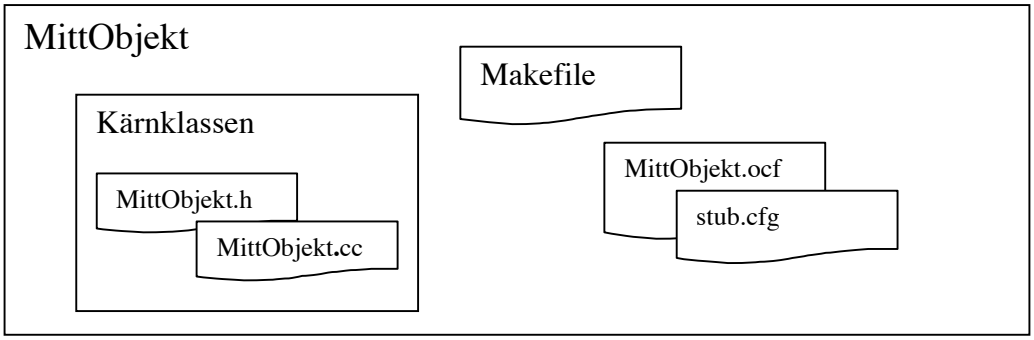
Vid IOK:

1. alla sändare och mottagare meddelar att de är redo att skicka respektive ta emot meddelande.
2. sändaren skickar meddelandet till mottagaren genom att köra metoden `Ready()`.
3. sändaren meddelar alla berörda mottagare via metoden `NotifyObservers()` att nu finns meddelandet.
4. mottagaren får tillgång till innehållet i minnesplatsen med metoden `Notify()`.
5. efter att at emot meddelandet kör mottagaren metoden `AssertReady()` som talar nu om att mottagaren är redo att ta emot nästa meddelande. (tillbaka till steg 1)

Denna process pågår så länge programmet körs.

3.4 Att skapa ett Objekt.

Mjukvarumoduler i OPEN-R SDK kallas för objekt. Ett objekt i OPEN-R skiljer sig från ett objekt i andra objektorienterade programmeringsspråk som t.ex. Java och C++. Varje objekt i en applikation i OPEN-R SDK har sin egen katalog och måste innehålla ett antal filer.



Figur 3.4: Objekt i OPEN-R.

Processen att skapa ett objekt med OPEN-R SDK:

Steg 1**.** byt katalog till katalogen MittObjekt (skapa den om det behövs).

Steg 2**.** Katalogen MittObjekt skall innehålla:

1. Kärnklassen
2. MittObjekt.ocf filen
3. Stub.cfg filen
4. Makefile filen

1 Kärnklassen

Med kärnklassen menas filerna *.h och *.cc som beskriver en klass i C++.

MittObjekt.h

En h-fil enligt programmeringsspråket C++ regler innehåller alla deklarationer av samtliga variabler och metoder som förekommer i en klass.
Koden nedan visar hur en enkel. *.h-fil i OPEN-R SDK ser ut:

```
#ifndef MittObjekt_h_DEFINED
#define MittObjekt_h_DEFINED

#include <OPENR/OObject.h>
#include <OPENR/OSubject.h>
#include <OPENR/OObserver.h>
#include "def.h"

class MittObjekt: public OObject {
public:
MittObjekt ();
virtual ~MittObjekt () {}

OSubject*    subject[numOfSubject];
OObserver*   observer[numOfObserver];

virtual OStatus DoInit   (const OSystemEvent& event);
virtual OStatus DoStart  (const OSystemEvent& event);
virtual OStatus DoStop   (const OSystemEvent& event);
virtual OStatus DoDestroy(const OSystemEvent& event);

void Notify(const ONotifyEvent& event);
void Ready(const OReadyEvent& event);
};

#endif // MittObjekt_h_DEFINED
```

Alla klasser i OPEN-R ärver från klassen OObject. Det innebär att metoderna `DoInit`, `DoStart`, `DoStop`, `DoDestroy` som är deklarerade som virtuella metoder, måste finnas i de klasser som ärver OObject. Alla dessa metoder returnerar ett värde av typen `OStatus`, som innehåller en samling av konstantvärden för att ange ett objekts tillstånd.

Varje klass i OPEN-R måste ha tillgång till ett antal sändare och mottagare som finns definierat i applikationen. `OSubject*` och `OObserver*` är två data typer som kommer från standardklasser i OPENR med samma namn. Antalet sändare respektive mottagare anges av `numOfSubject` och `numOfObserver` som kommer från filen ”`def.h`”. `def.h` skapas i sin tur av stubgen2.

Interobjektkommunikation i OPEN-R genomförs med metoderna `Notify()` och `Ready()`.
Metoden `Ready()` måste vara med i alla sändarobjekt (Subject). Den har till uppgift att lägga ett meddelande i bufferten.
Metoden `Notify()` måste vara med i alla mottagarobjekten (Observer). Den tar emot data bufferten.

MittObjekt.cc

```
#include <OPENR/OSyslog.h>
#include <OPENR/core_macro.h>
#include "MittObjekt.h"

MittObjekt:: MittObjekt()
{
        OSYSDEBUG(("MittObjekt::MittObjekt()\n"));
}

OStatus
MittObjekt::DoInit(const OSystemEvent& event)
{
        OSYSDEBUG(("MittObjekt::DoInit()\n"));

        NEW_ALL_SUBJECT_AND_OBSERVER;
        REGISTER_ALL_ENTRY;
        SET_ALL_READY_AND_NOTIFY_ENTRY;
        return oSUCCESS;
}

OStatus
MittObjekt::DoStart(const OSystemEvent& event)
{
        OSYSDEBUG(("MittObjekt::DoStart()\n"));

        ENABLE_ALL_SUBJECT;
        ASSERT_READY_TO_ALL_OBSERVER;
        return oSUCCESS;
}

OStatus
MittObjekt::DoStop(const OSystemEvent& event)
{
        OSYSDEBUG(("MittObjekt::DoStop()\n"));
        OSYSLOG1((osyslogERROR, "Bye Bye ..."));

        DISABLE_ALL_SUBJECT;
        DEASSERT_READY_TO_ALL_OBSERVER;
```

```
        return oSUCCESS;

}

OStatus
MittObjekt::DoDestroy(const OSystemEvent& event)

{
        DELETE_ALL_SUBJECT_AND_OBSERVER;
        return oSUCCESS;
}

void
MittObjekt::Notify(const ONotifyEvent& event)
{
        const char* text = (const char *)event.Data(0);
        OSYSPRINT(("MittObjekt::Notify() %s\n", text));
        observer[event.ObsIndex()]->AssertReady();
}
void
MittObjekt::Ready(const OReadyEvent& event)
{
        OSYSPRINT(("MittObjekt::Ready() : %s\n",
        event.IsAssert() ? "ASSERT READY" : "DEASSERT READY"));

        strcpy(str, "!!! Hello !!!");
        subject[sbjSendString]->SetData(str, sizeof(str));
        subject[sbjSendString]->NotifyObservers();
}
```

`OSYSDEBUG` ger utskrift på Terminalfönstret vid körning med debuggflaggan på.
`OSYSLOG1` ger utskrift på Terminalfönstret när programmet upphör att exekvera.
`OSYSPRINT` ger utskrift på Terminalfönstret vid programmets gång.
Alla dessa metoder kommer från klassen OPENR/OSyslog.h.

Metoden `AssertReady()` anropas alltid från metoden `Notify()` i ett mottagarobjekt för att meddela alla sändare att nu är mottagning av data genomfört.
Metoden `NotifyObservers()` anropas alltid från metoden `Ready` i ett sändarobjekt för att meddela alla mottagare att nu är meddelande utsänd och det kan hämtas från bufferten.

2 mittObjekt.ocf

En ocf-fil specificerar konfigurationen av ett objekt och används vid länkningen.

```
object MittObjekt 3072 16384 128 cache tlb user
```

En.ocf-fil består av en enda rad som innehåller:

1. `object`
   ett reserverat ord

2. **Objektets namn**

3. **Stackstorleken:**
   I bytes. Rekommenderat värde: `3072`

4. **Heap storleken**
   I bytes. Rekommenderat värde: `16384`

5. **Schemaläggnings prioritet**

   Är en 8-bitars integer som visar schemaläggningsprioritet av ett objekt i processtabellen. Rekommenderat värde: `128,` som garanterar att alla processer skall ha samma prioritet i början.

   De 4 första bitarna (mest signifikanta) visar ”prioritetsklassen” hos ett objekt. Ett objekt med lägre värde körs aldrig så länge ett objekt med högre värde körs.

   De sista 4 bitarna visar ”tidsförhållandet” mellan objekt med samma prioritetsklass. Ju högre värde desto mer tid åt objektet.

   Observera att $(1000\ 0000)_2 = (128)_{10}$.

6. **Cache-minne**

   Det kan stå ”cache” eller ”nocache”. `cache` rekommenderas.
   Användningen av cache-minnet gör datornsarbete snabbare.

7. **TLB**

   Det kan stå ”tlb” eller ”notlb”.
   tlb =translation lookaside buffer
   tlb anger användningen av det virtuella minnet medan ” notlb” anger användningen av det fysiska minnet.
   `tlb` rekommenderas tillsammans med ”memprot”.

8. **Mode**

   Kan väljas mellan ”kernel” och ”user”. `user` rekommenderas.
   En process i en dator kan köras i två olika tillstånd. Ett övervakartillstånd som heter kernel och ett användartillstånd, user. Vid övervakartillstånd kan användaren inte ha kontroll över minnesanvändning hos en process.

3 stub.cfg

Endast en stub.cfg-fil skall finnas med för varje objekt. Den har som uppgift att etablera en ”telefonlinje” mellan sändare och mottagare. Den berättar för systemet hur många sändare och mottagare finns i just detta objekt och dessutom vilka funktioner kommunicerar med varandra.

```
ObjectName : MittObjekt
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : " MittObjekt.SendString.char.S", null, Ready()
Service : " MittObjekt.ReceiveString.char.O", null, Notify()
```

En stub.cfg-file består av:

1. ObjectName:
   Det ska vara samma som objektets namn.

2. NumOfsubject
   Antal sändare

3. NumOfObserver
   Antal mottagare

4. Service

Första termen

Sändarensnamn för en sändaredeklaration eller mottagarensnamn för en mottagardeklaration.

Andra termen

Ett unikt funktionsnamn som bestäms utan några begränsningar.

Tredje termen

Datatypen hos de utsända eller mottagna data.

Fjärde termen

”S” står för sändare (Subject) och ”O” står för mottagare (observer).

Femte termen

När kopplingsresultat mottagits så körs denna metod som finns med i kärnklassen. Med en koppling menas en kanal mellan mottagare och sändare. Kopplingsresultat definieras i kärnklassen och anropas varje gång ett meddelande mottagits eller ett meddelande sänds. I fall man ej behöver en sådan metod, så sätter man det till `null.`

Sjätte termen

Det måste alltid finnas åtminstone en sändare och en mottagare i varje objekt.
Här skall stå namnet på en metod som finns i kärnklassen. Denna metod är den enda metoden i hela programmet som har exklusivt tillgång till den kanal som beskrivs med Serviceraden. Om det inte finns någon sändare eller /och mottagare i programmet, så skriver man `null` vid platsen för sjättetermen.
I fall man inte har någon kanal alls, defaultvärdet gör objektet till en sändare.

I det här fallet har vi ett objekt, dvs. MittObjekt, som är både sändare och mottagare. Annars om man önskar att bara ha ett objekt som antingen är mottagare eller sändare, så kan man ändra den oönskade serviceraden till:

```
Service : " MittObjekt.DummySubject.DoNotConnect.S", null, null
```

4 Makefile

Den här är objektkatalogens Makefile. Här definieras de kommandon som utför kompilering, stubgenerering och länkning. Innehållet i filen är känslig för små och stora bokstäver och måste exakt vara som det är.

Frågetecknet framför variabeln `OPENRSDK_ROOT` vid första raden i Makefilen nedan är i själva verket en if-sats som kontrollerar existensen av katalogen till `OPENR_SDK` vid adressen `/usr/local/OPEN_R_SDK` och det ser ut som:

if (finns_adressen(`/usr/local/OPEN_R_SDK`)) then
`OPENRSDK_ROOT=/usr/local/OPEN_R_SDK`
`else`
(”ett fel har inträffat, kan ej hitta `adressen`”);

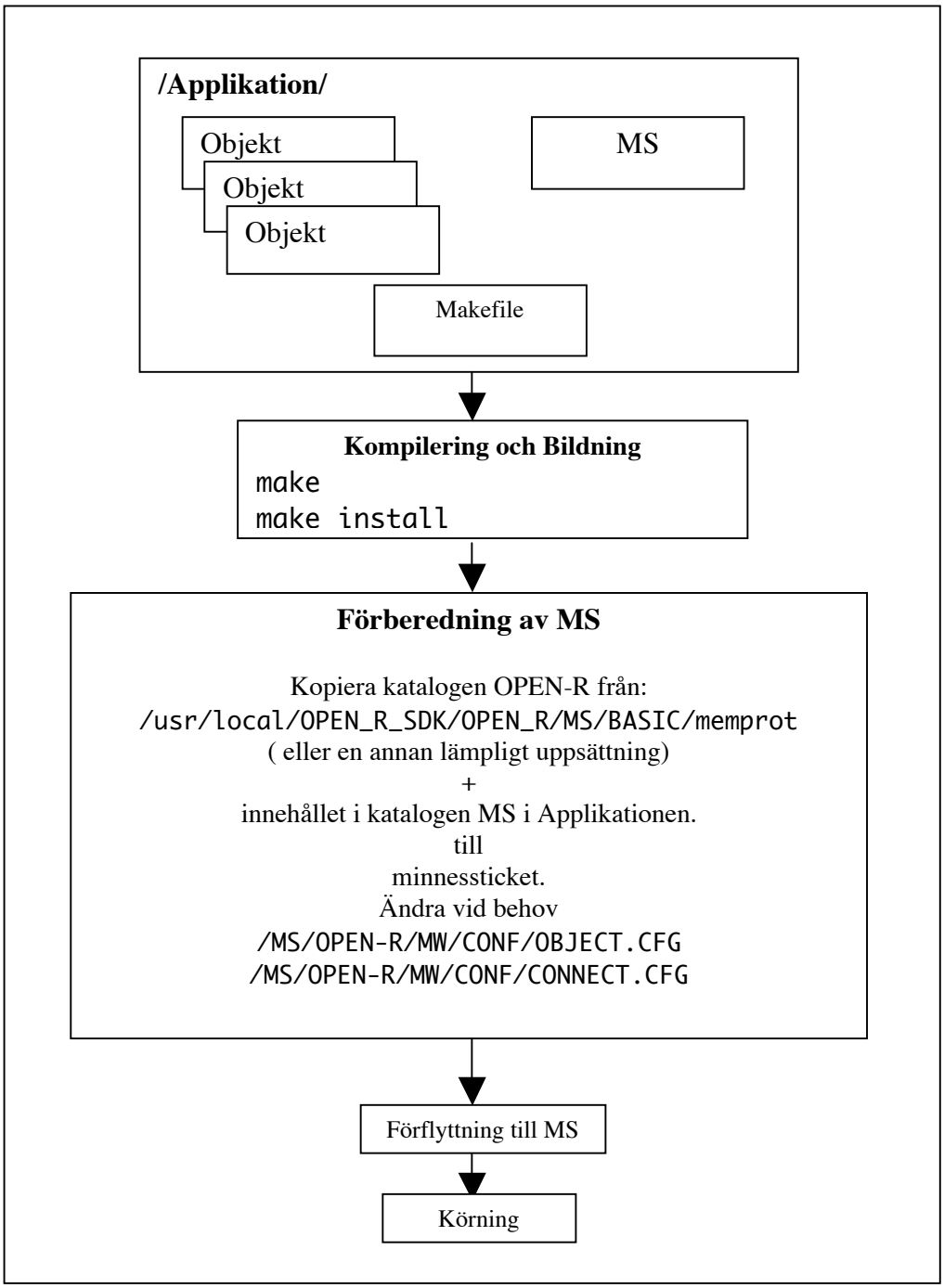
```
OPENRSDK_ROOT?=/usr/local/OPEN_R_SDK
INSTALLDIR=../MS
CXX=$(OPENRSDK_ROOT)/bin/mipsel-linux-g++
STRIP=$(OPENRSDK_ROOT)/bin/mipsel-linux-strip
MKBIN=$(OPENRSDK_ROOT)/OPEN_R/bin/mkbin
STUBGEN=$(OPENRSDK_ROOT)/OPEN_R/bin/stubgen2
MKBINFLAGS=-p $(OPENRSDK_ROOT)
LIBS=-L$(OPENRSDK_ROOT)/OPEN_R/lib -lObjectComm -lOPENR
CXXFLAGS= \
          -O2 \
          -g \
          -I. \
          -I$(OPENRSDK_ROOT)/OPEN_R/include/R4000 \
          -I$(OPENRSDK_ROOT)/OPEN_R/include

CXXFLAGS+=$(DEBUGFLAGS)
```

```
.PHONY: all install clean

all: mittObjekt.bin

%.o: %.cc
	$(CXX) $(CXXFLAGS) -o $@ -c $^

MittObjektStub.cc: stub.cfg
	$(STUBGEN) stub.cfg

mittObjekt.bin: MittObjektStub.o MittObjekt.o mittObjekt.ocf
	$(MKBIN) $(MKBINFLAGS) -o $@ $^ $(LIBS)
	$(STRIP) $@

install: mittObjekt.bin
	gzip -c mittObjekt.bin > $(INSTALLDIR)/OPEN-R/MW/OBJS/
MITTOBJEKT.BIN

clean:
	rm -f *.o *.bin *.elf *.snap.cc
	rm -f MittObjektStub.h MittObjektStub.cc def.h entry.h
	rm -f $(INSTALLDIR)/OPEN-R/MW/OBJS/MITTOBJEKT.BIN
```

Kapitel 4

Kompilering, körning och felsökning

Hela processen med att förbereda ett färdigskrivet program fram till körningen kan betraktas i bilden nedan:



Figur 4.1: Kompileringsprocess.

4.1 Kompilering med OPEN-R SDK

För att kompilera ett program med OPEN-R SDK:

1. Byt katalog till katalogen som innehåller applikationen. För katalogbyte används kommandot ***cd*** följd av adressen till applikationskatalogen, t.ex.

   ```
   cd XXX
   ```

OBSERVERA ATT: med XXX menas adressen till den katalog som applikationen ligger i.

2. Kompileringen och länkningen genomförs med kommandon:

   ```
   make
   make install
   ```

Resultatet av kompileringen är binärafiler (.* BIN) som hamnar i katalogen MS som finns i applikationskatalogen.

4.2 Att välja rätt konfigiration för en minnesstick.

För att ett OPEN-R program skall kunna köras så måste hela operativsystemet och ett antal andra program laddas in i systemet. Detta görs genom att kopiera en katalog med förberedda filer till roten av minnessticket. De fyller systemdelen av MS med en katalog med samma namn. Det finns 6 MS konfigurationer att välja mellan. Valet av dessa konfigurationer baseras på applikationen som skall köras.
Alla dessa konfigurationer finns att hämta från adressen
`usr/local/OPEN_R SDK/OPEN_R/MS/.`

Dessa 6 kataloger är:

- `BASIC/memprot/OPEN-R`
- `BASIC/nomemprot/OPEN-R`
- `WLAN/memprot/OPEN-R`
- `WLAN /nomemprot/OPEN-R`
- `WCONSOLE/memprot/OPEN-R`
- `WCONSOLE /nomemprot/OPEN-R`

Tabell 4.1 MS konfigurationer.

Begrepp	**Betydelse**	**Fördelar**
`BASIC`	Utan sladdlöskommunikation	Sparar på minnet lämpligt vid körning av mycket enkla program.
`WLAN`	Med sladdlöskommunikation, Utan terminalfönster	Sparar på minnet.
`WCONSOLE`	Med sladdlöskommunikation, Med terminalfönster	Ger större kontrollmöjligheter om vad som händer inuti programmet vid körningen.
`nomemprot`	Utan minnesskydd	Minnet tilldelas i lagom mängd vid behov. Programmet körs snabbare.
`memprot`	Med minnesskydd	Lättare att hitta fel med. Mer utnyttjande av minnet.

4.3 Körning

Det finns ett antal steg som måste följas vid förflyttningen av filer till minnessticket:

1. Montera minnessticket: med kommandon:

```
mount /mnt/memory_card
```

Kopiera katalogen med den valda konfigurationen av systemet till roten av ett tomt minnesstick:

```
cp /usr/local/OPEN_R_SDK/OPEN_R/MS/WCONSOLE/memprot/OPEN-R  /mnt/memory_card
```

I stället för WCONSOLE kan stå BASIC eller WLAN och i stället för memprot kan det stå nomemprot beroende på val av systemet.

3. Kopiera innehållet av katalogen MS/ i din applikation till roten av minnesstick.

Utför umount för att stänga av minnesstickets filsystem.

```
umount /mnt/memory_card
```

4. Sätt in minnessticket i ERS-210.

5. Tryck på startknappen.

4.4 Felsökning

Vanligen för att veta vilket fel har inträffat och var i koden felet befinner sig i, använder man olika utdatarutiner som printf och cout. Vid kompilering av ett program kan en del syntaktiska fel upptäckas av kompilatorn. Dessa fel är egentligen de enklaste att förutse och upptäcka, eftersom det handlar mest om grammatik och stavning. I AIBOn körs olika objekt samtidigt. Detta skapar risken att utdata från olika objekt blandas med varandra. Detta problem undviks med hjälp av rutinerna OSYSPRINT() och OSYSLOG() som ger utdata vid körningen av programmet. Dessa metoder fungerar i princip som cout och printf och tar samma sorts argument. De tar max 243 tecken plus ett avslutande nulltecken. Användning av debuggflagg gör det möjligt att få textmeddelande vid körningen.

```
OSYSDEBUG(("xxxxxxxxxx text meddelnde xxxxxxxxxx\n"));
```

Processoravbrott (CPU exception)

All information om processoravbrott samlas i en fil som kallas för EMON.LOG i minnessticket vid adressen /MS/OPEN-R/EMON.LOG. Denna fil skapas endast i samband med själva processavbrottet, dvs. det existerar inte vid körningen.

När ett processoravbrott sker:

1. Systemet startar avbrottsövervakaren (Exception Monitor) med syfte att stoppa alla objekt i systemet.

2. Avbrottsövervakaren kör alla kommandon som finns med i filen `MS/OPEN-R/SYSTEM/CONF/EMON.CFG` i tur och ordning.

```
play exception
play warning
exception > /MS/OPEN-R/EMON.LOG
cp0 >> /MS/OPEN-R/EMON.LOG
cp1 >> /MS/OPEN-R/EMON.LOG
objs >> /MS/OPEN-R/EMON.LOG
dstack -max 0x4000 -r 0x40 >> /MS/OPEN_R/EMON.LOG
play finish
```

Ett exempel på `EMON.CFG`.

play exception: meddelar att avbrottsövervakaren har startats.

play warning: varnar om att det skrivs på minnessticket. En melodi börjar att spelas.

exception: Kommandot beskriver informationen om det objekt som har orsakat processoravbrottet och innehållet i orsaksregistern.

cp0: Det skriver ut innehållet i koprocessor eller systemets kontroll koprocessorn (system control coprocessor) vilket arbetar parallellt och samtidigt med den centrala processorn för att upptäcka och åtgärda avbrotten.

cp1: Samma som cp0 men de gäller innehållet i register i koprocessor1, som tar hand om reella talen (floating-point unit).

objs: Det ger en lista över de existerande objekten vid processoravbrottet. Kommandot ger en beskrivning om namnen på de objekten tillsammans med sina omgivningsidentifikationer (ContextID).

stack / dstack: Dessa kommandon ger information om stacken vid processoravbrottet. dstack ger även innehållet av stacken.

3. När avslutningsmelodin är slut så är programmet terminerat (**play finish**).

För ytterliggare information se:

- Tutorial 6: Debugging OPEN-R Programs.
- Tutorial 7: Debugging by gdb using Remote Processing OPEN-R
- AIBO Programmer's Guide

Kapitel 5

Lösningförslag till programmeringsproblem

I detta avsnitt visas hur man kan sammansluta och samordna flera objekt för att skapa en kraftfullare enhet i form av en applikation eller ett program. Anledningen till att man behöver kraftfullare applikationer är för att kunna hantera och utföra mer komplexa uppgifter med en AIBO som t.ex: kartlägga omgivningen, hitta ett landmärke, hitta objekt, sparka en boll, osv. En applikation består av ett antal mindre enheter som är specialister i att genomföra en specifik uppgift. En sådan enhet kallas för ett objekt.

En fördel med att programmera med OPEN-R är att man kan sammansluta och samordna ett antal objekt i ett program på ett enkelt och effektivt sätt. Med sammanslutning av olika objekt menas att anordna interobjektkommunikation mellan dessa objekt. Medan samordning är att få ett antal objekt att samarbeta harmoniskt för att uppnå ett specifikt mål i form av en applikation.

Olika objekt kombineras i ett program för att kunna genomföra mer komplicerade uppgifter som kräver att ha flera objekt inblandade för att klaras av. De inblandade objekten skall kunna kommunicera med andra objekt i programmet medan de körs samtidigt. Ett objekt kan kommunicera antingen med andra ”användarobjekt” eller ”systemobjekt” i ett program. Systemobjekt förklaras närmare i avsnitt 5.3.

Ett exempel

Målet med detta kapitel är att förklara de inblandade stegen i utvecklingsprocessen för en komplex applikation som får en AIBO att sparka på en boll. Utvecklingsprocessen kan delas i tre huvudmoment: analys, design och implementation.

5.1 Utvecklingsprocessen

5.1.1 Analys

I exemplet ska det skapas ett program som får en AIBO att sparka på en boll. Denna uppgift kommer att kallas för huvuduppgiften. Den kan delas up i ett antal deluppgifter som är både mindre och lättare att implementera. Det hjälper programmeraren att både förstå hela uppgiften närmare och få ett grep om vilka delar uppgiften består av. Ett förslag på dessa deluppgifter är:

1. att få roboten att stå upp på sina fyra ben och behålla balansen hela tiden;
2. att hitta bollen;
3. att uppskatta sitt avstånd till bollen;
4. att gå till bollen;
5. att sparka bollen i rätt riktning.

5.1.2 Design

Här läggs betoningen på att beskriva uppgiften lite mer detaljerat, genom att dela upp varje deluppgift från analysdelen till ett antal enkla och odelbara moment som kan beskrivas med kod. Designdelen kan delas upp i 4 delmoment:

1 Bestäm de inblandade objekten i programmet.
2 Bestäm metoderna i varje objekt.
3 Bestäm sändar- och mottagarmetoder i varje objekt.
4 Organisera kommunikation mellan objekten i programmet.

1. **Bestäm de inblandade objekten i programmet.**

   Varje objekt i OPEN-R motsvarar en självständigt och körbar process. Detta innebär att exemplet ovan kan delas upp i ett antal sådana processer som tar hand om:

   I- Färgigenkänning.
   II- Avståndsmättning.
   III- Data från gravitationssensorn.
   IV- Huvudets rörelser.
   V- Gångstilar (att stå upp, sitta, ligga, gå, springa).
   VI- Övervakningen av alla objekt i systemet.
   VII- Strömbrytaren hos en AIBO (skall alltid vara med).

2. **Bestäm metoderna i varje objekt.**

Alla objekt i OPEN-R ärver från klassen `OObject`. I klassen `OObject` finns det 4 abstrakta metoder: `DoInit()`, `DoStart()`, `DoStop()` och `DoDestroy()` som är deklarerade som abstrakta. Dessa metoder kallas för Do-metoderna. Därför måste alla klasser som ärver från `OObject` innehålla Do-metoderna.

Det behövs ytterliggare ett antal metoder i varje objekt för att utföra den specifika uppgiften som varje objekt skall klara av. Till exempel i objektet som tar hand om ”färg igenkänningen” skall 2 metoder finnas:

1. En metod som tar en bild av omgivningen.
2. En metod som känner igen bollen i den bild som kommer från metod 1.

Samma princip gäller för resten av objekten i programmet.

3. **Bestäm sändar- och mottagarmetoder i varje objekt.**

I ett objekt kan det finnas både sändar- och mottagarmetoder. Därför kan ett objekt vara både sändare och mottagare, men inte i samma ögonblick. Anledningen är att varje objekt har ett enda exekveringstråd och kan utföra en uppgift i taget (antingen skicka eller ta emot information).

I vårt exempel i objektet som känner igen färgen måste Metod 1 ta emot den information som kommer från kameran. Därför är den en mottagarmetod.

4. **Organisera kommunikation mellan objekten i programmet.**

Objekt i OPEN-R måste ibland ha tillgång till information från andra objekt för att kunna genomföra sina uppgifter. Till exempel objektet som tar hand om huvudets rörelser måste ha tillgång till information från objektet som känner igen färg för att veta när bollen har hittats och när det skall avsluta sin uppgift.

Olika objekt i ett OPEN-R program kan kommunicera med varandra genom att skicka och ta emot meddelande eller information via ett gemensamt buffert.
En sändare är ett objekt som vill utnyttja en effektor eller vill skicka en order till ett annat objekt. En mottagare är ett objekt som kan ta emot data från en sensor eller utföra en tjänst genom att ta emot en order.

I vårt exempel är objektet som tar hand om gångstilar en sändare för att den skickar order till effektorer, vilka är mottagare av order. Objektet som tar hand om gravitationen är en mottagare för att den tar emot data från gravitationssensorn som är en sändare.

Kommunikation mellan objekt sker antingen mellan två användareobjekt eller ett användareobjekt och ett systemobjekt. Olika anordningar hos en AIBO kontrolleras av olika systemobjekt. Det finns endast två systemobjekt i OPEN-R. De heter: **OVirtualRobotComm** och **OVirtualRobotAudioComm**. Tillsammans har de kontroll över hela AIBOn**.** Se avsnitt 5.2.

5.1.3 Implementation

Här ska alla de objekten beskrivna i designdelen skapas. Till detta ändamål gäller det att:

1 Skapa de 5 första objekten i avsnittet om design.
2 Testa och köra varje objekt separat.
3 Skapa ett ram till hela uppgiften.
4 Sammansluta objekten.
5 Samordna objekten.
6 Testa hela systemet.
7 Köra applikationen.

5.2 Lite mer om systemobjekt

Hårdvaran i en AIBO kontrolleras av systemobjekt. Det finns två systemobjekt i OPEN-R. De heter: **OVirtualRobotComm** och **OVirtualRobotAudioComm.** Tillsammans har de kontroll över hela AIBOn**.**

`OVirtualRobotAudioComm`

är det systemobjekt som har kontroll över ljudresurser hos en AIBO. Med ljudresurser menas både högtalarna och mikrofonen. För att utnyttja dessa anläggningar skall kommunikation mellan användarobjektet och systemobjektet upprätthållas med hjälp av följande servicerader i stub.cfg-filen.

1. För att spela upp ljud med högtalaren:
`OVirtualRobotAudioComm.Speaker.OSoundVectorData.O`

2. För att ta emot ljud från mikrofonen:
`OVirtualRobotAudioComm.Mic.OSoundVectorData.S`

Högtalaren är en effektor därför den är en mottagare. Den tar emot och reagerar på data från programmet. Eftersom en effektor skall utföra en uppgift måste det finnas minst ett användarobjekt som sänder data av samma datatyp som effektorn kan ta emot. Samma gäller sensorer fast där är de sändare medan ett användarobjekt måste kunna ta emot data av den fördefinerade typen.

`OVirtualRobotComm`

`OVirtualRobotComm` kontrollerar resten av AIBOs resurser. Här delas resurserna i 3 grupper:

1. Effektorer
`OVirtualRobotComm.Effector.OCommandVectorData.O`

2. Sensorer
`OVirtualRobotComm.Sensor.OCommandVectorData.S`

3. Kamera (vilket är också en sensor, fast med annorlunda datatyp)
`OVirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S`

5.3 Primitivlokaliserare

Varje hårdvarukomponent eller anordning hos en AIBO har ett eget namn. Detta namn kallas för ”CPC Primitive Locator” eller kort för bara ett ”Locator”. CPC står för ”Configurable Physical Components” och syftar på det faktum att en AIBO är hårdvvarumoduliserad, dvs man kan byta en del av kroppen hos en AIBO utan att påverka resten av den. Varje primitivlokaliserare börjar med ”PRM:/” och resten av namnet ser ut som en fil- eller katalogadress i Linux.

Här presenteras alla Primitiven som finns i en AIBO ERS-210:

Huvudet

Nacken

`PRM:/r1/c1-Joint2:j1`	lutning till sidorna
`PRM:/r1/c1/c2-Joint2:j2`	nicka (fram och tillbaks)
`PRM:/r1/c1/c2/c3-Joint2:j3`	vridning till höger och vänster

Munnen (stänga och öppna)

`PRM:/r1/c1/c2/c3/c4-Joint2:j4`

Trycksensorer

PRM:/r1/c1/c2/c3/f1-Sensor:f1	på baksidan av huvudet
PRM:/r1/c1/c2/c3/f2-Sensor:f2	på framsidan av huvudet
PRM:/r1/c1/c2/c3/c4/s5-Sensor:s5	strömavbrytaren

PSD (Position Sensitive Detector) - infraröd avståndmätare

PRM:/r1/c1/c2/c3/p1-Sensor:p1

Ljud

PRM:/r1/c1/c2/c3/m1-Mic:M1	Mikrofonen
PRM:/r1/c1/c2/c3/s1-Speaker:S1	Högtalaren

Kameran

PRM:/r1/c1/c2/c3/i1-FbkImageSensor:F1

Öronen

PRM:/r1/c1/c2/c3/e1-Joint3:j5	vänstra öronen
PRM:/r1/c1/c2/c3/e2-Joint3:j6	högra öronen

LED (Light Emitting Diodes) - Lampor

PRM:/r1/c1/c2/c3/l1-LED2:l1
PRM:/r1/c1/c2/c3/l2-LED2:l2
PRM:/r1/c1/c2/c3/l3-LED2:l3
PRM:/r1/c1/c2/c3/l4-LED2:l4
PRM:/r1/c1/c2/c3/l5-LED2:l5
PRM:/r1/c1/c2/c3/l6-LED2:l6
PRM:/r1/c1/c2/c3/l7-LED2:l7

Benen

Vänstra frambenet

PRM:/r2/c1-Joint2:j1	Ledet (lyfter benet på sidorna från höften)
PRM:/r2/c1/c2-Joint2:j2	Axeln (rör på benet fram och tillbaks från höften)
PRM:/r2/c1/c2/c3-Joint2:j3	Knäet

Vänstra bakbenet

PRM:/r3/c1-Joint2:j1	Ledet
PRM:/r3/c1/c2-Joint2:j2	Axeln
PRM:/r3/c1/c2/c3-Joint2:j3	Knäet

Högra frambenet

`PRM:/r4/c1-Joint2:j1`	Ledet
`PRM:/r4/c1/c2-Joint2:j2`	Axeln
`PRM:/r4/c1/c2/c3-Joint2:j3`	Knäet

Högra bakbenet

`PRM:/r5/c1-Joint2:j1`	Ledet
`PRM:/r5/c1/c2-Joint2:j2`	Axeln
`PRM:/r5/c1/c2/c3-Joint2:j3`	Knäet

Trycksensorer

`PRM:/r2/c1/c2/c3/c4-Sensor:s4`	Vänsterfram
`PRM:/r3/c1/c2/c3/c4-Sensor:s4`	Vänsterbak
`PRM:/r4/c1/c2/c3/c4-Sensor:s4`	Högerfram
`PRM:/r5/c1/c2/c3/c4-Sensor:s4`	Högerbak

Svansen

`PRM:/r6/c1-Joint2:j1`	fram och tillbaka rörelsen
`PRM:/r6/c2-Joint2:j2`	lutning
`RPM:/r6/l1-LED2:l1`	Blå lampan
`RPM:/r6/l2-LED2:l2`	Orange lampan
`PRM:/r6/t1-Sensor:t1`	Termometer
`PRM:/r6/s1-Sensor:s1`	Strömbrytaren på baksidan

Accelerations sensorer

`PRM:/a1-Sensor:a1`	y-axeln – fram och tillbaka (fram positiv)
`PRM:/a2-Sensor:a2`	x-axeln – höger och vänster (höger positiv)
`PRM:/a3-Sensor:a3`	z-axeln – upp och ner – (upp positiv)

För mer information se: Model Information for ERS-210 och Model Information for ERS-220.

5.4 Kommunikation med Systemobjekt

Om man kommunicerar med ett systemobjekt och önskar sig att få använda en resurs hos AIBOn, så skall man:

1. Deklarera **tillstånden** i programmet.

Det är vanligt att AIBOn skall ha olika tillstånd vid användning av ett systemobjekt. Det kan vara t.ex. stillastående, start, stå upp eller liggande tillstånd om man till exempel tänker på rörelsen hos en AIBO.

```
        enum States {
        State_IDLE,
        State_START,
        State_STANDUP,
        State_LAYDOWN
    };

    States state;
```

2. Deklarera de anordningar man vill använda i programmet, som konstanter med hjälp av sina **Primitivlokaliserare**. Här visas alla primitiver för alla lampor eller LED (Light Emitting Diodes) hos AIBO. De skall finnas med i de objekt som styr lamporna.

```
static const char* const DEVICE_LOCATOR[] = {
    "PRM:/r1/c1/c2/c3/l1-LED2:l1",
    "PRM:/r1/c1/c2/c3/l2-LED2:l2",
    "PRM:/r1/c1/c2/c3/l3-LED2:l3",
    "PRM:/r1/c1/c2/c3/l4-LED2:l4",
    "PRM:/r1/c1/c2/c3/l5-LED2:l5",
    "PRM:/r1/c1/c2/c3/l6-LED2:l6",
    "PRM:/r1/c1/c2/c3/l7-LED2:l7",
};
```

3. Deklarera antalet primitivlokaliserare.
   (Alla deklarationer i *.h-filer)

   ```
   static const size_t NUM_PRIMS = 2;
   ```

4. Deklarera en variabel för att ta emot och lagra anordningarnas ”primitive ID”.

   ```
   OPrimitiveID      DeviceID[NUM_LEDS];
   ```

5. Öppna alla anordningar som finns med i programmet.
   (`Objektetsnamn` syftar på namnet på det objekt som skall skapas)

```
void Objektetsnamn::OpenPrimitives()
{
        for (int i = 0; i < NUM_PRIMS; i++)
```

```
        {

                OStatus result =
                OPENR::OpenPrimitive(DEVICE_LOCATOR[i],
                & DeviceID [i]);

                if (result != oSUCCESS)
                {
                        OSYSLOG1((osyslogERROR, "%s : %s %d",
                        "Objektetsnamn::OpenPrimitives()",
                        "OPENR::OpenPrimitive() FAILED", result));
                }
        }
}
```

En anordning öppnas egentligen med:
`OPENR::OpenPrimitive(DEVICE_LOCATOR[i], &DeviceID[i]);`

De metoderna som har med hantering av hårdvaran att göra kommer från filen `OPENRAPI.h`. För att använda dem måste man ha med
`#include <OPENR/OPENRAPI.h>`
i början av filen.

6. Deklarera ett antal minnesområden i ett buffert.

   Vissa hårdvaruanordningar och processer behöver två extra gemensamma minnesplatser för att kunna arbeta med sin maximala hastighet, utan att saktas ner av andra processer eller anordningar i systemet. Dessa minnesplatser räknas som en enhet och kallas för en buffert. Man behöver en buffert för att blinka med lysdioder (LED) och ett för att kunna genomföra komplexa rörelser hos en AIBO.

   ```
   static const size_t NUM_COMMAND_VECTOR = 2;
   ```

7. Skapa order för att utföra något.

   Man kan allokera ett minnesområde med datatypen OCommandVectorData. Minnesarean är av typen MemoryRegionID.

```
void
Objektetsnamn::NewCommandVectorData()
{
    OStatus result;
    MemoryRegionID      cmdVecDataID;
    OCommandVectorData* cmdVecData;

    for (int i = 0; i < NUM_COMMAND_VECTOR; i++) {
```

```
result =
OPENR::NewCommandVectorData(NUM_LEDS,&cmdVecDataID,&cmdVecData);
        if (result != oSUCCESS) {
              OSYSLOG1((osyslogERROR, "%s : %s %d",
                    "Objektetsnamn::NewCommandVectorData()",
              "OPENR::NewCommandVectorData() FAILED", result));
        }

        region[i] = new RCRegion(cmdVecData->vectorInfo.memRegionID,
                                 cmdVecData->vectorInfo.offset,
                                 (void*)cmdVecData,
                                 cmdVecData->vectorInfo.totalSize);

        cmdVecData->SetNumData(NUM_LEDS);

        for (int j = 0; j < NUM_LEDS; j++) {
            OCommandInfo* info = cmdVecData->GetInfo(j);
            info->Set(odataLED_COMMAND2, ledID[j], 1);

            OCommandData* data = cmdVecData->GetData(j);
           OLEDCommandValue2* val = (OLEDCommandValue2*)data->value;
            if (i % 2 == 0) {
                val[0].led = (j % 2 == 0) ? oledON : oledOFF;
            } else {
                val[0].led = (j % 2 == 0) ? oledOFF : oledON;
            }
            val[0].period = 64; // 8ms * 64 = 512ms
            OSYSDEBUG(("ID[%d] %d\n", j, val[0].led));
        }
    }
}
```

5.5 Att skapa en applikation

Inledning

I detta avsnitt visas hur på ett konkret och praktiskt sätt kan en komplex applikation skapas genom att tillämpa utvecklingsprocessen. Samma exempel som inledningen av detta kapitel används i detta avsnitt, dvs. en AIBO som sparkar på en boll. För enkelhetens skull kommer vi i fortsättningen att referera till denna applikation för bara ”spelaren”.

Problemformulering

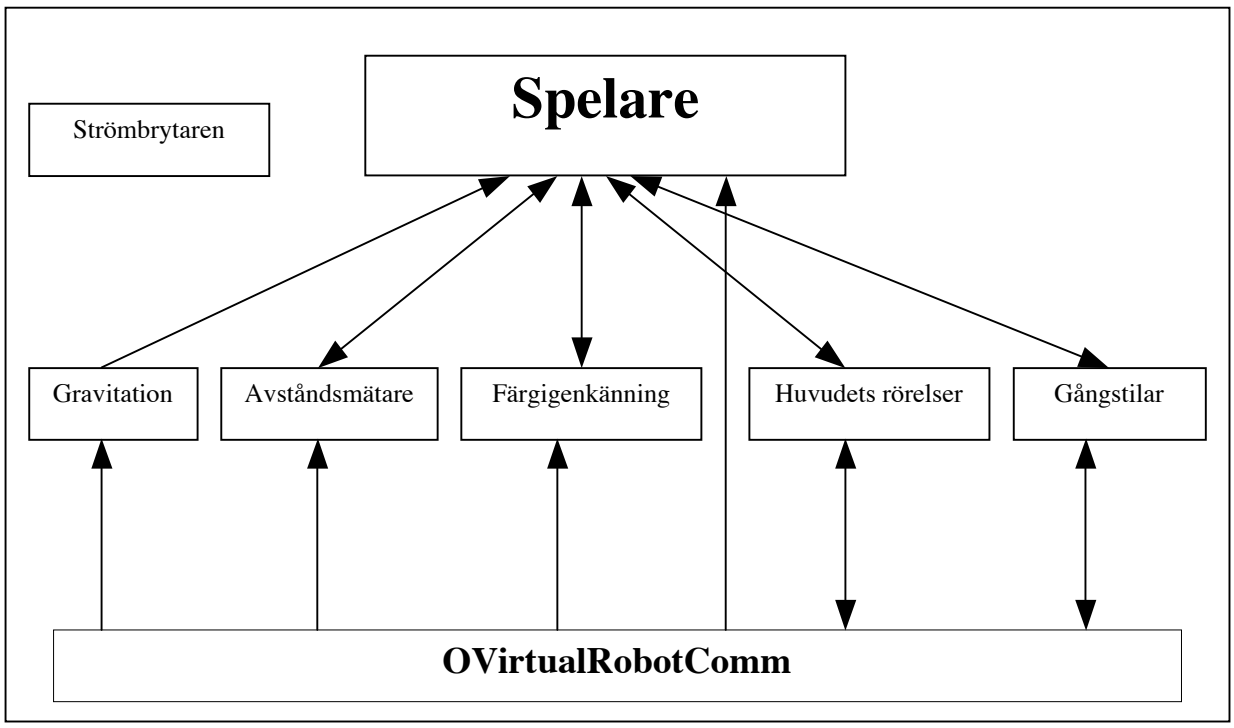
Vi vill skapa en applikation som får en AIBO att sparka på en boll. Detta innebär att:

När AIBOn sätts igång, den skall iaktta sin omgivning för att hitta bollen. Med detta menas att huvudet skall vrida sig på ett sätt att ett så stort fält i omgivningen kan fotograferas med kameran. När bollen är hittad den skall stoppa huvudetsrörelse, uppskatta avståndet och börja gå till bollen. När AIBO är tillräckligt nära bollen den skall kanske stanna och uppta ett nytt tillstånd, dvs. sparktillståndet, med 3 ben på märken och ett framben böjd. Sedan den skall sparka bollen.
Under hela processen den måste för det första kunna hålla sin balans och för den andra veta var bollen befinner sig, dvs i vilken riktning och vilket avstånd den befinner sig.

Analys

Att identifiera vilka objekt som ingår i en applikation är en av de svåraste uppgifter som en programmerare kan ställas inför. Det är en fråga av smak, förståelse om uppgiften och erfarenhet.
Ett tips är t.ex. att leta efter substantiven i problemformuleringen och identifiera dem som sina preliminära kandidater till objekt i applikationen. Sedan kan man ändra sig under design- och implementationsfasen.
Här är ett antal förslag till objekt baserad på problemformuleringen:

I- Färgigenkänning.(kameran)
II- Avståndsmättning.(avstånd)
III- Data från gravitationssensorn.(gravitation)
IV- Huvudets rörelser.
V- Gångstilar (att stå upp, sitta, ligga, gå, springa).
VI- Övervakningen av alla objekt i systemet.(ramobjektet)
VII- Strömbrytaren hos en AIBO (skall alltid vara med).
Bilden visar alla de inblandade objekten i applikationen Spelare och med sina inbördes kommunikationskanaler:



Figur 5.1 Applikationen Spelare.

Här följs stegen i implementationsfasen i Utvecklingsprocessen för att vidareutveckla applikationen Spelare. Dessa steg är:

1. Skapa de 5 grundobjekten.
2. Testa och köra varje objekt separat.
3. Skapa ett ram till hela uppgiften.
4. Sammansluta objekten.
5. Samordna objekten.
6. Testa hela systemet.
7. Köra applikationen.

1. Skapa de 5 grundobjekten.

Med de första 5 objekten menas de inre objekten dvs. Gravitation, Avstånd, Färg, Huvud och Gång. Dessa kan även ses i bilden ovan.

Att skapa ett objekt innebär att:

1 Skapa ett objekt
2. Anordna interobjektkommunikation
3. Modifiera koden efter vad man vill att objektet skall genomföra.

Dessa punkter kan man läsa om i:

- Tutorial 1, Inter-Object Communication.
- Tutorial 2, Communication with system objects.
- Tutorial 3, Get data from Camera.

Det går även att titta i exempelprogrammen från OPEN-R SDK för att hitta de exempel som liknar mest vad man vill göra. Våra förslag om vilka exempelprogram man kan analysera och metoderna som kan ingå i dessa är:

I- Färgigenkänning.
(se: AIBO/sample/ImageObserver)

1. Att ta en bild av omgivningen.
2. Att känna igen bollen.

II- Avståndsmättning (Avstånd).
(se: AIBO/sample/sensorObserver)

Att uppskatta avståndet till bollen från AIBOn genom att lokalisera sig i sin omgivning.

III- Data från gravitationssensorn (Gravitation).
(se: AIBO/sample/SensorObserver)

Att upptäcka och reagera på ändringar hos robotens balans.

IV- Huvudets rörelser (Huvud).
(se: AIBO/sample/MovingHead)

Att få huvudet att vrida på sig så länge bollen är inte hittad.

V- Gångstilar (att stå upp, sitta, ligga, gå, springa) (Gång).
(se: AIBO/sample/MoNet)

Att generera olika poseringar hos en AIBO.
Att generera, kontrollera och sköta gången på ett smidigt sätt.

VI- Övervakningen av alla objekt i systemet (Spelare).
(se: AIBO/SoccerLion200)

Att sätta igång alla objekt i systemet.
Att koppla de relevanta objekten till varandra.

2. Testa och köra varje objekt separat.

Om varje objekt inte klarar av sin uppgift så kommer naturligtvis inte hela applikationen att fungera heller. Så var säker på att varje objekt gör vad det skall göra innan ni skall sätta de ihop.

Se även Tutorial 6: Debugging OPEN-R programs och Tutorial 7 Debugging with gdb using Remote Processing OPEN-R.

3. Skapa ett ram till hela uppgiften.

Med ett ram menas en katalog som heter Spelare och innehåller 3 enheter:

1. Makefile (Ram-makefile)
2. Spelare/ (objektkatalogen)
3. MS/ (minnesstickskatalogen)

4. Sammansluta objekten.

Här gäller det att samla alla de inblandade objekt i ramkatalogen Spelare. Man gör detta genom att antingen flytta eller kopiera alla objektkatalogerna till ramkatalogen och ange objektkatalogensnamn eller modifiera komponentraden i Spelare/Makefile med rätt adress. I det senare fallet så behöver man inte ens flytta objektkatalogerna till Spelare. Man anger i stället adressen till objektkatalogerna relativt den aktuella adressen.

```
COMPONENTS= Geravitation Farg Avstand Huvud Gang Brytare
INSTALLDIR=$(shell pwd)/MS
TARGETS=all install clean

.PHONY: $(TARGETS)

$(TARGETS):
for dir in $(COMPONENTS); do \
```

```
        (cd $$dir && $(MAKE) INSTALLDIR=$(INSTALLDIR) $@) \
        done
```

Spelare/Makefile.

Nästa steg är att upprätta interobjektkommunikationen bland de inblandade objekten i applikationen. Detta genomförs genom att ändra i filen `CONNECT.CFG` i katalogen `Spelare/MS`.

Se även Tutorial4, Combine objects.

5. Samordna objekten.

Med samordna objekten menas att modifiera kärnklassen till applikationen Spelare så att den skall genomföra sin uppgift. Detta är en svår uppgift. Ofta finns det många objekt som är inblandade i en sådan applikation. Alltså applikationens storlek och dess komplexa struktur gör det svårt för programmeraren att förutse applikationens beteende fullt ut.

Se även Tutorial5: Combine objects and Coordinate them.

6. Testa hela systemet.

För att vara absolut säker på att allt fungerar tillsammans och applikationen gör vad den lovar så testa och testa igen.

Se även Tutorial 6: Debugging OPEN-R programs och Tutorial 7 Debugging with gdb using Remote Processing OPEN-R.

7. Köra applikationen.

Se Kapitel 4 avsnitt om Körning (Avsnitt 4.3).

Referenser

OPEN-R SDKs officiella hemsida

http://openr.aibo.com/openr/eng/index.php4
Här kan man bland annat hitta den senaste OPEN-R SDK.

RoboCups hemsida
http://www.robocup.org

TeamSwedens hemsida
http://www.aass.oru.se/Agora/roboCup

AIBOs hemsida
http://www.aibo.com

Sonys Manualer

Installations Guide
Internet Protocol Version 4
Level 2 Reference Guide
Model Information 210
Model Information 220
Programmer's Guide
Revision Record
Specifications

Tutorial:

Tutorial 1	Inter-Object Communcation
Tutorial 2	Communication with Objects
Tutorial 3	Get data from Camera
Tutorial 4	Combine Objects
Tutorial 5	Combine Objects and Coordinate them
Tutorial 6	Debugging OPEN-R Programs
Tutorial 7	Debugging by gdb using Remote Processing OPEN-R

Bilaga A

Exempelprogrammen

På OPEN-R SDKs hemsida finns det ett antal exempelprogram. Dessa exempelprogram finns också tillgängliga i katalogen `/usr/local/AIBO/sample/` i laborationsdatorerna. Att studera exempelprogrammen är ett utmärkt sätt att få lära sig programmering med OPEN-R SDK. Genom att montera ner varje exempel avslöjas arkitekturen och mekanismen hos ett OPEN-R program. Tabellen nedan visar alla dessa exempel program tillsammans med en kort beskrivning om vad de gör:

Tabell A.1 Exempelprogrammen.

Programmets namn	**Beskrivning**
HelloWorld	Skriver ut texten ”HelloWorld” på Terminalfönstret.
ObjectComm	IOK
BlinkingLed	Blinkar med ögonen och rör öronen.
MovingHead	Rör på huvudet.
MovingLegs	Rör på ben och sömnar.
SensorObserver	Info från sensorer.
ImageObserver	Spelar in BMP-bilder.
SoundPlay	Spelar WAV audio filer.
SoundRec	Sparar ljud på MS i WAV-fromat.
BallTrackingHead	Letar efter en rosa boll med huvudet.
MoNet	Motion Network som genererar Rörelser.
PowerMonitor	Ger batteriens tillstånd (kapacitet/temperatur).
Crash	CPU undentag tillstånd.
PIDControl	Proportionell Integrerande Differentiell styrning.
RobotDesign	AIBO typ.
EchoClient	AIBO=klienten, Datorn=servern (wireless LAN).
EchoSever	AIBO=severn (wireless LAN).
TinyFTPD	Enkel FTP server (wireless LAN).
LmasterRSlave	Höger benet imiterar vänster benets rörelser.
ImageCapture	Tar en bild med AIBO.
DNSLookUp	Ta IP adressen via DNS.
ERA201D1Info	Information Om wireless LAN kortet ERA-201D1.
W3AIBO	AIBO som en webb-kamera.

Exempelprogrammen kan också uppdelas baserad på vad de utför:

Uppgift	Exempel
Enkla program	
a. Utskrift på Terminalen	HelloWorld
b. IOK	ObjectComm
c. Batteristatus	PowerMonitor
d. CPU-Exception	Crash
e. Rörelse/gång styrning	PIDControl
Sensorer	
f. AIBO-typen	RobotDesign
g. Internetadress via DNS	DNSLookUp
h. Data från sensor till Terminalen	SensorObserver
Effektorer	
i. LED / Öronrörelser	BlinkingLED
j. Huvudrörelser	MovingHead
k. Benrörelse	MovingLegs
l. Färg-igenkänning	BalltrackingHead
m. Gångstilar	MoNet
n. Högra benet härmar det vänstra	LMasterrslave
Ljud	
o. Mikrofonen - spela in ljud	SoundRec
p. Högtalaren – spela ljud	SoundPlay
Kameran	
q. CCD-Kameran (BMP-format)	ImageObserver
r. CCD-Kameran	imagecapture
s. Kamera + webbserver	W3AIBO
Sladdlöskommunikation	
t. Klient (AIBO) / Server (PC)	EchoClient
u. AIBO eko Server	EchoServer
v. Enkel FTP sever	TinyFTPD
w. Inställningar till kortet	ERA201d1Info

A.1. HelloWorld

Exempelprogrammet HelloWorld erbjuder det enklaste programmet som kan skrivas med C++ i OPEN-R SDK. Det enda HelloWorld gör är att den skriver ut text på terminalfönstret. Trotts sin enkelhet är just HelloWorld ett utmärkt första steg för att lära sig programmera med OPEN-R SDK och förstå mekanismerna i ett OPEN-R program.

OBSERVERA ATT: de filer och kataloger som är understrukna har tidigare blivit beskrivna så kommer de inte att tas upp i resten av rapporten. I stället kommer i fortsättningen endast klassfilerna(.h + .cc) att förklaras tillsammans med de nya filerna där de dyker upp.

AIBO/sample/HelloWorld/

Katalogstruktur
(2 kataloger och en fil)
`HelloWorld/HelloWorld/`
`HelloWorld/MS/`
`HelloWorld/Makefile`

HelloWorld/HelloWorld/

Katalogstrukturen
(5 filer)
`HelloWorld.h`
`HelloWorld.cc`
`helloWorld.ocf`
`HelloWorldStub.cc`
`Makefile`

OBSERVERA ATT: filerna .h och .cc tillsammans kallas för klassfiler i programmeringsspråket C++.

HelloWorld.h

- **Include-filer**

  ```
  <OPENR/OObject.h>
  ```

  Alla objekt i OPEN-R ärver från klassen OObject. Här får man även tillgång till en del resurser i form av olika metoder, konstanter och attribut.
  (se AIBO/OPENR.hfiler)

- **Konstruktor** `HelloWorld()`
- **Destruktor** `virtual ~HelloWorld() {}`

  Destruktorn i HelloWorld är deklarerad som virtuell. Detta innebär att destruktorn utför inget och är odefinierat.

- **Publika metoder:**

  Do-metoderna = `DoInit(), DoStart(), DoStop(), DoDestroy()`

```
virtual OStatus DoInit   (const OSystemEvent& event);
virtual OStatus DoStart  (const OSystemEvent& event);
virtual OStatus DoStop   (const OSystemEvent& event);
virtual OStatus DoDestroy(const OSystemEvent& event);
```

  De här 4 metoderna kallas för ”Do-metoder”. Do-metoderna är mycket viktiga och måste finnas med i alla OPEN-R objekt. Anledningen är att dessa metoder är deklarerade som virtuella i klassen `OObject`. Detta gör att enligt reglerna i programmeringsspråket C++ och objekt-orienterad programmering, så måste klasserna som ärver ifrån klassen `OObject` (vilket i själva verket alla klasser i OPEN-R gör) innehålla minst en definition av de 4 metoderna.

  `DoInit()` genomförs när objektet skapas. Nästan samma som konstruktorn.
  `DoStart()` genomförs vid körningen av objektet.
  `DoStop()` genomförs när körningen upphör (strömknappens nedtryckning).
  `DoDestroy()` genomförs när objektet tas bort. Samma funktion som destruktorn.

  Do-metoderna returnerar ett värde av typen `OStatus` som visar objektets inre tillstånd. `OStatus` erbjuder en mängd resurser i form av ett antal fördefinerade konstanter (se `AIBO/OPENR.hfiler/OStatus.h`).

  Do-metoderna tar emot indata i form av en konstant av typen `OSystemEvent&`. Tecknet & i slutet av `OSystemEvent` talar om att den här indata är en referensvariabel. Referensvariabler möjliggör att man kan returnera fler värden än bara ett från en metod. I det här fallet kan `event` vara både indata till och utdata ifrån Do-metoderna. (för `OSystemEvent` se `AIBO/OPENR.hfiler/OPENREvent.h`)
  Do-metoderna tillsammans med destruktorn i HelloWorld.h är deklarerade virtuella. Det innebär att definitionerna till metoderna:

```
void Init   (const OSystemEvent& event);
void Start  (const OSystemEvent& event);
void Stop   (const OSystemEvent& event);
void Destroy(const OSystemEvent& event);
```

  som finns i klassen OObject används i stället. Dessa metoder kallas även för OObject-metoderna. Tyvärr definitionen av dessa metoder är inte tillgängliga!
  (se `AIBO/OPENR.hfiler/OObject.h`)

HelloWorld.cc

Include-filer

```
<OPENR/OSyslog.h>
```

ger tillgång till metodena `OSYSPRINT`, `OSYSDEBUG` och `OSYSLOG1`.
`OSYSPRINT` skriver ut på terminalfönstret vid körningen.

```
OSYSPRINT(("!!! Hello World !!!\n"));
```

`OSYSDEBUG` skriver meddelanden på terminalfönstret vid körning med debuggflaggen

```
OSYSDEBUG(("HelloWorld::HelloWorld()\n"));
```

`OSYSLOG1` skriver meddelanden på terminalfönstret när körningen upphör.

```
OSYSLOG1((osyslogERROR, "Bye Bye ..."));
```

OBSERVERA ATT: det finns två paranteser framför alla ovanstående metoder!

`"HelloWorld.h"`
enligt reglerna i programmeringsspråket C++ ,så måste .h-filen vara inkluderad i början av .cc-filen. De måste ha samma namn (klassnamnet). Deklarationerna av metoderna och attributer i en klass ligger i .h-filen, medan definitionen av samma metoder finns i .cc-filerna.

helloWorld.ocf

Se avsnitt 3.4

HelloWorldStub.cc

Vanligast är att den filen är autogenererad från stub.cfg mha programmet stubgen2 för att bland annat specificera sändar- och mottagarmetoderna i ett objekt, skapa ett antal hjälpfiler som t.ex. ”Stub.cc”. HelloWorld saknar interobjektkommunikation, därför läggs objektetmetoderna i tabellingången och startar OObject-metoderna i HelloWorldStub.cc.

Makefile

Direktiven till allt som sker under kompileringen (`make`) och länkningsprocessen (`make install`) finns i Makefile. Den beskriver hur skapas de körbara filerna dvs .BIN-filer och flyttas till katalogen /MS (här: `/HelloWorld/MS/OPEN-R/MW/OBJS/`). Användaren kan även ta bort alla de hjälpfiler som har skapats under kompileringen och länkningsprocessen med hjälp av kommandot: `make clean`.

/HelloWorld/MS/

Katalogstruktur
`/MS/OPEN-R/MW/CONF/OBJECT.CFG`
`/MS/OPEN-R/MW/OBJS/`

Katalogen `/MS/` innehåller ett antal kataloger och filer som tillsammans med ett antal andra kataloger och filer från `/usr/local/OPEN_R_SDK/OPEN_R/MS` fyller i minnesticket. Dessa kataloger är även tillgängliga i laborationsdatorer under katalogen `AIBO/MS`.

För mera detaljer se: AIBOs Programmer’s Guide, kapitel 4.

Allt som slutligen en AIBO kommer att ha tillgång till vid körningen av ett program skall finnas med i minnessticket som är endast 16 MB stor! Det finns alla de körbara filerna (.BIN) och även hela operativsystemet Aperios med i minnessticket.

OBJECT.CFG

Adressen till alla de körbara filerna (.BIN) som finns i minnessticket måste vara tillgängliga i filen OBJECT.CFG för systemet. Varje körbarfil som finns med i katalogen: ”/MS/OPEN-R/MW/OBJS/” i minnessticket refereras med en egen adresserad i OBJECT.CFG.

```
/MS/OPEN-R/MW/OBJS/POWERMON.BIN
/MS/OPEN-R/MW/OBJS/HELLO.BIN
```

OBSERVERA ATT: Ibland finns det även en rad till i filen `OBJECT.CFG`, som refererar till en .BIN-fil med namnet: `MS/OPEN-R/MW/OBJS/POWERMON.BIN`. Detta objekt är inte kopplat till något annat objekt i applikationen och arbetar helt separat. Det har till uppgift att stänga av AIBO på ett säkert sätt när strömknappen trycks ned. Utan det här objektet så stängs inte av AIBOn helt, utan den kommer att befinna sig i ett mellanläge som liknar paus-läget och lysdioden ovanför strömknappen förblir röd även efter nedtryckning.

HelloWorld/Makefile

Det finns en Makefile in i klass-katalogen som har redan presenterats. Det finns en Makefile till som ligger utanför klasskatalogen, men fortfarande inom applikationen. Den här är mycket enklare och har till uppgift att starta kompileringen, länkningen och flyttningen av de körbara filerna, katalog för katalog för varje objekt som är definierad i applikationen.

Det finns även en Makefile som kompilerar och länkar alla exempelprogrammen på en gång! (se `AIBO/sample/Makefile`”)

A.2 Crash

Detta är ett exempelprogram som skapar avbröt (exception) hos processorn, för att förklara hur sådana ser ut och kan tas hand om.

OBSERVERA ATT: de filer och kataloger som är understrukna har tidigare blivit beskrivna så kommer de inte att tas upp i resten av rapporten. I stället kommer i fortsättningen endast klassfilerna(.h + .cc) att förklaras tillsammans med de nya filerna där de dyker upp.

AIBO/sample/Crash/

Katalogstruktur
(3 kataloger och en fil)
`Crash/Crash/`
`Crash/MS/`
`Crash/util/`
`Crash/Makefile`

Crash/Crash/

Katalogstruktur
(5 filer)
`Crash.h`
`Crash.cc`
`crash.ocf`
`CrashStub.cc`
`Makefile`

Crash.cc

- **Include-filer**
  `<iostream>`
  ger tillgång till c++-systemrutiner för manipulering av dataflödet.
  Var vänlig se: http://www.cplusplus.com/ref/#libs

  ```
  using std::cin;
  using std::cout;
  ```

  using deklarationer tillsammans följer namespacing i C++. De ger möjligheten att gruppera en del funktioner och attributer som om de vore globala i egna applikationer.
  Se: http://www.cplusplus.com/doc/tutorial/tut5-2.html

- **Metoder**

  Ett antal metoder som orsakar systemavbrött och skickar ett lämpligt textmeddelande vid händelsen till kommandofönstret. Dessa avbröt beskrivs bäst med hjälp av sina respektive textmeddelanden:

```
crash_func_entry crash_func_table[] = {
    { access_null_data_pointer, "access null data pointer" },
    { access_null_text_pointer, "access null text pointer" },
    { destroy_stack, "destroy stack" },
    { cause_address_miss_alignment, "cause address miss alignment"},
    { use_unusable_coprocessor, "use unusable coprocessor" },
    { jump_to_broken_text, "jump to broken text" },
    { cause_tlb_modification_error,
      "cause TLB modification error (memprot only)" },
    { overflow_on_convert, "overflow on convert" },
    { not_a_number_source, "use not-a-number source" },
    { denormalized_number_source, "use denormalized number source"},
};
```

Se filen `AIBO/sample/Crash/Crash.cc`

OBSERVERA ATT: För övriga filer och katalogerna i Crash/Crash gäller samma förklaringar som HelloWorld**.** POWERMON.BIN finns inte med i programmet Crash. Katalogen/Crash/util/innehåller endast filen ”emonLogparser”. Den visar information om processoravbrott.

A.3 ObjectComm

Vad är interobjektkommunikation? (IOK)

InterObjektKommunikation eller ”IOK” är en av de viktigaste och mest grundläggande principerna i programmering med OPEN-R. Det sker både i form av kommunikation mellan egenskrivna objekt och även mellan systemobjekt och egenskrivna objekt. IOK påminner mycket om klient/server principen, dvs. att tjänster erbjuds av olika objekt baserad på efterfrågan av andra objekt.
Alla objekt kan skicka så väl som ta emot meddelanden, dvs. kan vara både sändare och mottagare, men inte i samma ögonblick! Sändarobjektet i OPEN-R SDK kallas för ”Subject” medan mottagarobjekten kallas för ”Observer”.

Vid IOK:

1. alla sändare och mottagare meddelar att de är redo att skicka respektive ta emot meddelande.

2. sändaren skickar meddelandet genom att köra metoden `Ready().`

3. sändaren meddelar alla mottagare via metoden `NotifyObservers()` att nu finns meddelandet tillgängligt i bufferten.

4. mottagaren får tillgång till innehållet i bufferten med metoden `Notify().`

5. efter att ta emot meddelandet kör mottagren metoden `AssertReady()` som talar nu om att mottagaren är redo att ta emot nästa meddelande. (Tillbaka till steg 1)

Exemplet ObjectComm visar praktiskt hur interobjektkommunikation mellan två enkla egetskrivna objekt fungerar. Det här exemplet är betydligt enklare än det nya exemplet från OPEN-R SDK. Det finns den minimala koden som behövs för att IOK skall fungera. Om man skulle vilja skriva en applikation med IOK det är en bra ide att ta detta exempel som grund för att bygga på resten av sitt program på.

OBSERVERA ATT: de filer och kataloger som är understrukna har tidigare blivit beskrivna så kommer de inte att tas upp i resten av rapporten. I stället kommer i fortsättningen endast klassfilerna(.h + .cc) att förklaras tillsammans med de nya filerna där de dyker upp. s

AIBO/oldsample/ObjectComm/

Katalogstruktur
(3 kataloger och en fil)
`ObjectComm/SampleObserver/`
`ObjectComm/SampleSubject/`
`ObjectComm /MS/`
`ObjectComm /Makefile`

ObjectComm/SampleObserver/

Katalogstruktur
(5 filer)
`SampleObserver.h`
`SampleObserver.cc`
`sampleObserver.ocf`
`stub.cfg`
`Makefile`

SampleObserver.h

- **Include-filer**

  ```
  #include <OPENR/OObject.h>

  #include <OPENR/OSubject.h>
  ```
  (se `AIBO/includefiler`) erbjuder allt som en sändare måste ha.

  `#include <OPENR/OObserver.h>` (se `AIBO/includefiler`) erbjuder allt som en mottagare måste ha.

  `#include "def.h"`
  skapas av stubgeneratorn som finns vid adressen `/usr/local/OPEN_R_SDK/OPEN_R/bin/stubgen2` efter `make` och `make install`. Stubgeneratorn är ett program som skapar ett antal hjälpfiler. Direktivern om vad Stubgen skall göra finns med i Makefile.
  Filen def.h kan hittas i katalogen `ObjectComm/SampleObserver/` efter `make` och `make install`.

- **Attributer**

  ```
  OSubject*    subject[numOfSubject];
  OObserver*   observer[numOfObserver];
  ```

  Dessa rader talar om för objektet hur många sändare (Subject) och hur många mottagare (Observers) finns med i systemet. Antal sändare och mottagare bestäms av programmeraren genom att redigera filen `stub.cfg`.

- **Metoder**

  Do-metoderna
  (`DoInit()`, `DoStart()`, `DoStop()` och `DoDestroy()`) finns med i alla OPEN-R Objekt.

  ```
  void Notify(const ONotifyEvent& event);
  ```

  Metoden `Notify()` som har tillgång till bufferten och därför den är ”den riktiga mottagaren”.

SampleObserver.cc

- **Include-filer**

  ```
  #include <OPENR/core_macro.h>
  ```

  De makron som finns inuti OObjekt-metoderna i SampleObserver.cc kommer härifrån. `OPENR/core_macro` är en samling av makron eller kodtillägg som förbereder bland annat IOK hos objekten.
  (se `AIBO/includefiler/core_macro.h`)

- **Metoder**

  Här finns definitionen av de vanliga Do-metoderna. Vad som är viktigt är innehållet i dessa metoder. Alla rader i Do-metoderna är makron och därför finns med i `OPENR/core_macro.h`.

  `DoInit()`

  DoInit initierar objektet och gör 3 saker:

  1. `NEW_ALL_SUBJECT_AND_OBSERVER;`
     Skapar ett antal sändare och mottagare. Antalet finns i filen ”entry.h” skapad av Stubgeneratorn.

  2. `REGISTER_ALL_ENTRY;`
     Laddar in och registrerar alla sändare och mottagaren i systemet genom att tilldela de egna ID-nummer. Den erbjuder även kommunikkationssevice till aktörerna i IOK.

  3. `SET_ALL_READY_AND_NOTIFY_ENTRY;`
     Som det framgår av makrotsnamn den förbereder alla sändare och mottagare som finns med i systemet.

```
OStatus
SampleObserver::DoInit(const OSystemEvent& event)
{
    NEW_ALL_SUBJECT_AND_OBSERVER;

    REGISTER_ALL_ENTRY;
    SET_ALL_READY_AND_NOTIFY_ENTRY;

    return oSUCCESS;
}
```

DoStart()

DoStart() förbereder alla sändare och mottagare för sändning respektive mottagning av data. Den består endast av två makro som genomförs vid körningen av objektet:

1. ENABLE_ALL_SUBJECT;
   Förbereder alla sändare.

2. ASSERT_READY_TO_ALL_OBSERVER;
   Förbereder alla mottagare att ta emot data.

```
OStatus
SampleObserver::DoStart(const OSystemEvent& event)
{
    ENABLE_ALL_SUBJECT;
    ASSERT_READY_TO_ALL_OBSERVER;
return oSUCCESS;
}
```

DoStop()

Gör som DoStart(), fast tvärtom! Dvs den stoppar alla sändare och mottagare.

1. DISABLE_ALL_SUBJECT;

2. DEASSERT_READY_TO_ALL_OBSERVER;

```
OStatus
SampleObserver::DoStop(const OSystemEvent& event)
{
    DISABLE_ALL_SUBJECT;
    DEASSERT_READY_TO_ALL_OBSERVER;

    return oSUCCESS;
}
```

DoDestroy()

Till slut tas alla sändare och mottagare bort.

DELETE_ALL_SUBJECT_AND_OBSERVER;

```
OStatus
SampleObserver::DoDestroy(const OSystemEvent& event)
```

```
{
    DELETE_ALL_SUBJECT_AND_OBSERVER;
    return oSUCCESS;
}
```

`Notify()`

Den här metoden är ”den riktiga mottagaren”. Utan `Notify()` så kan inget objekt få tillgång till den gemmensamma minnesplatsen i systemmet och därmed ingen IOK.

Det mest väsentliga med `Notify` är:

1. data kommer in i metoden i form av ett händelse. I händelsen finns själva datan eller meddelandet i form av ett fält eller vektor:

   ```
   const char* text = (const char *)event.Data(0);
   ```

   (märkera även typkonverteringen)

2. slutligen meddelas alla sändare att metoden är färdig med att ta emot data genom att anropa `AssertReady()` och är redo för att ta emot nästa meddelande.

   ```
   observer[event.ObsIndex()]->AssertReady();
   ```

```
void
SampleObserver::Notify(const ONotifyEvent& event)
{
    const char* text = (const char *)event.Data(0);
    OSYSPRINT(("SampleObserver::Notify() %s\n", text));
    observer[event.ObsIndex()]->AssertReady();
}
```

---

stub.cfg

1. `ObjectName : SampleObserver`

2. `NumOfOSubject   .: 1`
   `NumOfOObserver .: 1`

   Man talar om nu för systemet att hur många tjänster man vill ha och av vilken typ man vill att de skall vara. Här innebär att man begär den minimala tjänsten som IOK kan erbjuda.

OBSERVERA ATT: `NumOfOSubject` och `NumOfOObserver` får aldrig tilldelas noll.

3. `Service : "SampleObserver.DummySubject.DoNotConnect.S", null, null`
   `Service : "SampleObserver.ReceiveString.char.O", null, Notify()`

Man talar om för systemet att vilka tjänster eller service man vill ha. Den här raden kallas för ”serviceraden”. Det är naturligt att alla aktörer kan vara både sändare och mottagare. Man kan även vara mer sändare än mottagare eller tvärtom! Hur? Se första seviceraden med ordet ”DummySubject”. Det innebär att objektet SampleObserver är endast en mottagare. Det tar endast emot data och kan inte sända någon data även om det vill!

OBSERVERA ATT: Antalet sändar- och mottagartjänsterna måste överensstämma med antalet av serviceraderna. För varje service så skall det finnas en servicerad.

Serviceraden

En servicerad består av 4 delar:

1. `Service :`

   Helt enkelt ordet `service` tillsammans med kolon.

2. ”Förbindelsen”

   Den biten i serviceraden sätts i citationstecken. Den består av 4 delar:

Del 1	Del 2	Del 3	Del 4
Objektets namn	Ett unikt service namn	En datatyp	servicetypen
Ett kärnklass namn. Samma namn som står framför **Objectname** :	Vilket namn som helst! Det måste vara unikt i hela applikationen	Den datatyp som meddelandet har.	**S** för sändare (Subject) och **O** för mottagare (Observer)

3. Bekräftelsemetoden

   Den används bara för att bekräfta att kopplingen är genomfört. Metoden kan heta vad som helst (vanligen `Control()`) och måste finnas med i kärnklassen. I de flesta fall behövs ingen sådan metod och fälten kan fyllas med `null`.

4. Meddelandemetoden

   Metoden måste finnas med i kärnklassen. Om servicen är för en:

   - Sändare (Subject)

     Metoden anropas efter `ASSERT-READY` eller `DEASSERT-READY` från en mottaggare. Dvs. efter att ett meddelande är mottaget av en mottagare. Här brukar man använda: `Ready().`

   - Mottagare (Observer)

     Metoden anropas efter att ett meddelande är skickat av en sändare. Här brukar man använda `Notify().`

```
ObjectName : SampleObserver
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : "SampleObserver.DummySubject.DoNotConnect.S", null, null
Service : "SampleObserver.ReceiveString.char.O", null, Notify()
```

`stub.cfg` för mottagarobjektet

```
ObjectName : SampleSubject
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : "SampleSubject.SendString.char.S", null, Ready()
Service : "SampleSubject.DummyObserver.DoNotConnect.O", null, null
```

`stub.cfg` för sändareobjektet

OBSERVERA ATT : Det går bra att återanvända samma bekräftelse- och meddelandemetoder i olika servicerader i samma `stub.cfg`. Var vänlig och även jämför skillnaderna i `stub.cfg` mellan mottagar- och sändarobjekten ovan.

För mer information se: Programmer's Guide, kapitel 2.3 "`Stub`", sida 11. Titta gärna i olika exempel program för att se hur olika `stub.cfg` filer ser ut.

`stub.cfg` är egentligen bara ett kopplingsregister. Systemet vill också veta vilka man vill skicka meddelande till eller ta emot meddelanden ifrån. Det vill även veta vilken typ av meddelanden det gäller! Den här informationen finns i filen: `ObjectComm/MS/OPEN-R/MW/CONF/CONNECT.CFG` som kommer lite senare.

OBSERVERA ATT: Det skall alltid finnas en `stub.cfg` för varje objekt i programmet.

ObjectComm/SampleSubject/

Katalogstrukturen
(5 filer)
`SampleSubject.h`
`SampleSubject.cc`
`sampleSubject.ocf`
`stub.cfg`
`Makefile`

SampleSubject.h

- **Metoder**

  ```
  void Ready(const OReadyEvent& event);
  ```

  Det är den här metoden skickar meddelandet.

SampleSubject.cc

- **Include-filer**
  `#include <string.h>`

  Den här filen ger tillgång till ett antal metoder som möjliggör manipulering med t.ex strängar. Denna fil finns med som standard i C++ systembiblioteken och erbjuder metoder som t.ex:
  `sizeof(str)`
  som ger längden på en sträng, och `strcpy (str, "!!! Hello world !!!");` som kopierar strängen.

- **Metoder**

  Ready()

  Metoden skickar data till det gemensamma minnesplatsen genom att:

  1. `subject[sbjSendString]->SetData(str, sizeof(str));`

  2. Om det finns flera sändarservice i ett sändarobjekt, så krävs en index som visar vilken sändarservice det gäller. Indexen är `sbjSendString.` Om det finns bara en sändarservice, så kan indexen `sbjSendString` ersättas med `0.`

  3. `Subject` är en pekare och har metoden `SetData` som är meddelandebäraren i sin klass.

  4. `subject[sbjSendString]->NotifyObservers();`

  5. Till slut meddelar sändaren till alla mottagare att nu är meddelandet avsänt med `NotifyObservers().`

     Alla ovannämnda metoder kan hittas i `AIBO/includefiler/OSubject.h`.

```
void
SampleSubject::Ready(const OReadyEvent& event)
{
    OSYSPRINT(("SampleSubject::Ready()  : %s\n",
                event.IsAssert() ? "ASSERT READY" : "DEASSERT
READY"));

    static int counter = 0;
    char str[32];

    if (counter == 0) {

        strcpy(str, "!!! Hello world !!!");
```

```
        subject[sbjSendString]->SetData(str,  sizeof(str));

        subject[sbjSendString]->NotifyObservers();

    } else if (counter == 1) {

        strcpy(str, "!!! Hello world again !!!");

        subject[sbjSendString]->SetData(str,  sizeof(str));
        subject[sbjSendString]->NotifyObservers();

    }

    counter++;
}
```

ObjectComm/MS/

Nyheten här är filen `MS/OPEN-R/MW/CONF/CONNECT.CFG`. Den här filen ”beskriver förbindelsen mellan sändar- och mottagarmetoderna i en applikation för systemet”.

För `CONNECT.CFG` eller ”förbindelse-filen” gäller:

1. Varje OPEN-R applikation med interobjektkommunikation måste ha endast en `CONNECT.CFG` i sin minnesstick på ovannämnda adressen.

2. Den består här bara av en enda rad och med denna rad berättar den för systemet om vilka ”förbindelser” mellan sändar- och mottagarmetoder skall finnas med. Utan den kan inte systemet veta var informationen skall sändas och tas emot.

3. Innehållet i CONNECT.CFG är identiskt med vad som står på förbindelsedelen av serviceraden i stub.cfg i varje objekt i applikationen. (den delen i serviceraden som står mellan citattecknen)

4. Allmänt gäller en rad per varje förbindelse som:

   ```
   Klass1.metod1.datatyp1.S   Klass2.metod2.datatyp1.O
   Klass1.metod3.datatyp2.S   Klass3.metod4.datatyp2.O
   ```

   Alltså:

En sändarservice	Ett mellanslag	En mottagarservice

OBSERVERA ATT: datatypen måste överensstämma mellan sändaren och mottagaren.

A.4 PowerMonitor

För att AIBO skall kunna avstängas utan att den känsliga elektroniken inuti den skall skadas så följer ett exempelprogram som fungerar separat men även finns med i de flesta andra exempelprogrammen. Detta exempel kallas för PowerMonitor och är i själva verket en samling av rutiner som har med strömmen i AIBO att göra.

A.5 RobotDesign

Det ser ut först att exempelprogrammet RobotDesign innehåller interobjektkommunikation (IOK) med tanke på närvaron av filen stub.cfg och ett antal rader i *.h-filen. Men faktum är att det finns ingen IOK i detta exempelprogram. Därför kan det tas med i kategorin enkla program. Det ger helt enkelt modellen av AIBOn.

Bilaga B

Ordlistan

AI	Artificiell Intelligens.
AIBO	Artificially Intelligent roBOts Detta är benämningen för en serie av Sony tillverkade fyrbenta smarta robotar. Själva ordet på japanska betyder ”vän”. En AIBO innehåller ett inbyggt system med begränsad minneskapacitet och hastighet. Den version av AIBO som ligger till grunden av denna rapport är modellen ERS-210, som tillgänglig finns vid robotlabbet vid Lunds Universitet. För mer information var vänlig se: http://www.us.aibo.com/ http://www.aibo-europe.com/ http://www.sony.net/Products/aibo/aiboflash.html/
Aperios	Sonys objekt-orienterade Realtids operativsystem.
API	Application Programming Interface – programmeringsgränssnitt.
CCD	Couple Charged Device - Kameran
Cdt	Color detection data
Cygwin	för Unix liknande kommandon som t.ex: sh,cp, … i Windows.
CPC	Configurable Physical Components
CPU	Central Processing Unit – processorn
DNS	DNS eller ”Domain Name System” används för att översätta domännamnen till IP-adresser på internet. De flesta internettjänster använder DNS för att fungera. DNS kontrollerar även emailleveransen på internet. Om DNS upphör att fungera, så kan webbadresser inte lokaliseras korrekt och emailleveransen och andra internettjänster upphör att fungera. För mer information om DNS var vänlig se: http://www.dns.net/dnsrd/.
ERS	Entertainment RobotS – underhållningsrobotar
exec	ett systemanrop i UNIX för att ladda en process in i minnet för att köras.
fork	ett systemanrop i UNIX för att dela en process.
GNU	General Public License - Gnu's Not Unix
ID	IDentification = identitet
Idle	stillastående
IOK	Interobjektkommunikation
LAN	Local Access Network
LED	Light Emitting Diodes
MB	MegaBytes (2^{20} bites = 1048576 ≈ 10^6 Bytes)
MHz	MegaHertz – datorns klockfrkvens
MIPS	Million Instructions Per Second, ett sätt att mäta datorns beräkningskapacitet. Också namnet till ett företag som tillverkar datorer,

newlib	MIPS kompilerings verktyg
OPEN-R SDK	Ett lågt nivå kompileringsanvändargränssnitt med C++
OS	Operativsystem
Pipe	en kanal som kopplar två processer samman i UNIX.
PSD	Position Sensitive Detector – infraröd avståndsmätare
RAM	Random Access Memory – primärminne
RISC	Reduced Instruction Set Computer
RGB	Red, Green, Blue
RoboCup	Den årliga internationella tävlingen för fotbollspelande robotar
Signaler	ett tecken från operativsystemet (UNIX) till en process vid systemavbrott för att avsluta sin körning.
Socket	ett gränssnitt i UNIX som tillåter användareprocessen att koppla sig till ett nätverk för att sända och ta emot data.
Terminalfönstret	ett program som tar emot kommandon från användaren
TLB	Translation Lookaside Buffer
YcrCb	Y= skärpa, Cr= avvikelsen från röd, Cb= avvikelsen från blå